# The Art of Reversing

by ApOx

# The art of Reversing

## by Ap0x

***Predgovor drugom izdanju***

Mi zivimo nase svakodnevne zivote nesvesni sitnica i malih stvari koje nam se desavaju ispred ociju. Prelazimo preko ociglednih stvari jer nam se cine jednostavnim i logicnim. Ne osecamo potrebu da zavirimo ispod jednolicne jednostavne spoljasnosti stvari, ili samo to ne zelimo? Postavimo sebi pitanje: Kada smo zadnji put pogledali neki objekat i zapitali se kako on to radi? Kakvi se to procesi kriju iza njegovog nama dostupnog lica? Zasto se nesto desava bas tako kako se desava? Kada smo neki objekat poslednji put razlozili na sastavne delove snagom svojih misli? Kada smo zadnji put videli iza ociglednog i dostupnog? Odgovori na ova pitanja leze u nama samima i predstavljaju samu osnovu reversnog inzenjeringa. Sama teznja da se pronikne u pocetke i uzroke stvari, da se do pocetnih uslova dodje od rezultata otvara neverovatne mogucnosti, samo ako promenimo tacku gledista, samo ako se sa mesta pasivnog posmatraca pomerimo u mesto reversera, samo tako cemo doci do same srzi reversnog inzenjeringa. Imajte na umu da se reversni inzenjering ne primenjuje samo na kompjuterima, svuda oko nas je reversni inzenjering, samo to treba da uocimo.

Ovo je drugo izdanje knjige *The Art Of Cracking* koja je dobila novo ime *The Art Of Reversing* jer je u nju sada utkana moja druga knjiga pod nazivom *PE and ASM for Crackers.* U ovom drugom izdanju su dodata neka poglavlja, neka su dopunjena ali veci deo ispravki se odnosio na gramaticko-semanticke greske uocene u knjizi za ovo pisac duguje posebnu zahvalnost MDHamel-u koji je uradio reviziju i lektorat knjige. Ovom prilikom bih zeleo da se zahvalim svima koji su me podrzali i jos uvek me podrzavaju da nastavim rad na ovom projektu.

Knjiga je posvecena svim ljudima koji su ostavili neizbrisiv trag u mom zivotu: porodici, najboljim prijateljima, prvoj ljubavi, mentorima, ostalim prijateljima, neprijateljima i ostalima koji nisu ovde nabrojani ali predstavljaju manje ili vise bitan deo mog zivota.

*"The more I learn, the more I realize how much I don`t know!"*
*Ap0x*

**The Book**

# 01 Intro to cracking

Kao prvo poglavlje u knjizi ovo poglavlje ima za cilj da vas uvede u reversni inzenjering i da vam pokaze samu osnovu crackovanja, nacin razmisljanja i neke osnovne trikove sa alatima za cracking. Prvo cete usvojiti neke osnovne pojmove vezane za cracking, naucicete kako se konfigurisu alati koje cemo koristiti, i konacno cemo napraviti nas prvi crack.

# What is R.C.E.?

Reverse Code Engineering je tehnika pomocu koje dobijamo pocetne vrednosti neke funkcije pocinjuci od njenih rezultata. Primeticete da sam upotrebio uopstenu definiciju RCEa, a ne onu koja bi se odnosila na RCE primenjen na kompjuterske aplikacije. RCE sam definisao ovako jer je on upravo to, bez obzira na oblast na koju se primenjuje RCE predstavlja tehniku razmisljanja, odnosno postupak resavanja nekog problema iz drugog ugla. Ali ako se ogranicimo samo na kompjutere onda cemo RCE definisati kao tehniku modifikacije nepoznatog koda, kada izvorni kod samog programa nije dostupan. Koristeci tehnike opisane u ovoj knjizi shvaticete osnovne probleme pri modifikaciji ili analizi nepoznatog koda kada nije poznat izvorni kod samog problema. Tada moramo pristupiti reversnom posmatranju problema, to jest trazenju uzroka razlicitih ponasanja programa, pocevsi od samih posledica, da bi smo stigli do pocetnik uzroka. Naravno kao i svaka druga oblast ljudskog delovanja, pa tako i u RCEu imamo razlicite probleme koji u vecini slucaja nemaju jedinstvena resenja. Ovo ujedno znaci da se vecina problema moze resiti na veliki broj nacina i da u vecini slucajeva postoje samo najlaksa, odnosno najbrza resenja, i ona koja to nisu. Posto nas u vecini slucajeva ne zanima vreme potrebno da se neki problem resi, glavni faktor u resavanju RCE problema ce biti tacnost dobijenih rezultata. Ova tacnost je surova kada se radi o RCE problemima, jer u vecini slucajeva postoje samo dva slucaja resavanja problema. Postoje samo tacno reseni problemi i oni koji to nisu. RCE problemi koji su netacno reseni mogu dovesti do nestabilnosti sistema, kao i do pucanja samog operativnog sistema koji koristimo kao osnovu za RCE. Ovo ujedno znaci da RCE nije platformski definisan jer se moze primenjivati, a i primenjuje se, na svim kompjuterskim platformama.

Iako je ova knjiga “pompezno” nazvana The Art Of Cracking ona se ne odnosi na pravo znacenje reversnog inzenjeringa, niti je tako nesto uopste moguce. Ne, ova knjiga je sebi stavila za cilj da se ogranici samo na jednu usko definisanu oblast, popularno nazvanu Cracking, tezeci da opise sto je vise moguce pojava vezanih za sam Cracking. Naravno i ovo je tesko izvodljivo ali cu se potruditi da vas kroz ovu knjigu uverim da nepostoji stvar koja se naziva “sigurna aplikacija”. Od sada pa na dalje ovaj termin izbrisite iz svoga recnika. Termin koji cete usvojiti umesto toga je: “aplikacija teska za reversing!”, sto znaci da svaka aplikacija koja moze da se pokrene moze biti “slomljena” i da ne treba verovati takozvanim komercijalnim aplikacijama za zastitu vasih ili tudjih programa. Ova knjiga ce unistiti zablude onih koji veruju da su njihovi passwordi bezbedni u bazama podataka, da su njihovi passwordi bezbedni iza “zvezdica”. Ovakve zablude ce pasti u vodu posle citanja ove knjige. Programeri spremite se, jer vase aplikacije ce biti stavljene na obiman test...

# Beginners guide to Reversing

Pre nego sto pocnete da se bavite reversnim inzenjeringom potrebno je da znate neke osnove kompjuterske hijerarhije i nacine zapisivanja/citanja podataka. Operativni sistem koji smo izabrali da na njemu savladamo osnove reversnog inzenjeringa je Windows, koji ce nam, bez obzira na verziju, pruziti uvid u arhitekturu i nacin razmisljanja koji se sprovodi prilikom reversinga.

Iako sam siguran da vec znate da je sama osnova Windows operativnog sistema niz executabilnih (*.exe*) i statickih (*.dll*) fajlova, koji predstavljaju jezgro sistema. Ono sto vecina od vas sigurno nije znala je da se sadrzaj tih .exe i .dll fajlova moze menjati na nacin na koji ce ti programi izvrsavati instrukcije koje mi zelimo. Tehnika modifikovanja tudjih executabilnih i drugih fajlova koji sadrze izvrsni kod naziva se Cracking. Imajte na umu da je reversing aplikacija za koji nemate odobrenje od njenih autora krajnje nelegalan i stoga budite obazrivi u odabiru meta koje cete reversovati.

Sada se sigurno pitate kako nam saznanje da se operativni sistem sastoji od velikog broja .exe i .dll fajlova moze pomoci u reversingu? Pa posto znamo ovo mozemo da pretpostavimo da se njihov sadrzaj na neki nacin zapisuje u sam .exe fajl! Ovde smo vec na pravom tragu jer svi .exe i .dll fajlovi imaju jedinstvene nacine zapisivanja. Ovi nacini zapisivanja su standardizovani na Windows 32bitnim sistemima (*verzije Windowsa od 98 pa na dalje*) i naziva se PE (*Portable Executable*) standard. Ovaj standard usko definise pozicije i znacenja svakog bajta (*najmanje jedinice svakog fajla, 1024b=1kb*) u jednom standardnom .exe fajlu. Sa ovim standardom cemo se kasnije upoznati ali je za sada bitno da znate da je deo standardnog PE fajla zaduzen za izrsavanje samih funkcija koje taj .exe fajl obavlja. Ove funkcije su napisane na takodje standardan masinski nacin. Ovo samo znaci da nizovi bajtova koji predstavljaju izvrsni kod imaju tacno svoje znacenje. Tumacenje niza bajtova obavlja sam procesor vaseg racunara, a ove standardne komande se nazivaju ASM (*asembler*) komande. Sa ASM standardom cemo se upoznati na samom pocetku ove knjige. Razumevanje tog dela knjige je preduslov za razumevanje svih ostalih poglavlja i stoga ono predstavlja okosnicu za celo razumevanje ove knjige.

Ovo su samo osnovni nacini razmisljanja koje bi svi pocetnici morali da imaju u vidu pre nego sto pocnu da se bave reversingom. Ostale jako bitne stvari ce biti uvodjene paralelno sa problemima sa kojima cete se sretati prilikom citanja ove knjige.

# Becoming a Reverser

Ovo je jako cesto pitanje koje svi koji hoce da se bave reversingom postavljaju sebi. Kako postati reverser? Sta to uopste znaci? Sta sve moram da znam?

Odgovori na ova pitanja su vise individualni, zavise od osobe do osobe, ali ono sto sam ja naucio tokom ovih godina baveci se programiranjem i reversingom je da je sve moguce, da se sve moze uraditi. Jedine dve stvari koje su potrebne za resavanje svakog problema su vreme i strpljenje. U zelji da postanete ono sto zelite da postanete moracete da naucite mnogo toga. Vecina stvari koje cete nauciti ce imati veze sa strukturom racunara, nacinom izvrsavanja programa, strukturom samih fajlova, strukturom Windowsa, ali pored svega ovoga moracete da naucite osnove kriptografije i matematike. Verovali ili ne ali najbolji problemi reverserskog sveta su matematicki. Stoga ma koliko mrzeli matematiku, verujte mi na rec, zavolecete je sigurno...

Kao sto sam vec rekao ne postoji tacan "recept" kako postati reverser ali postoje osnovni vodici koji ce vas voditi linijom kojom morate uciti kako biste postali reverser. Ovaj redosled bi trebalo da izgleda ovako:

- ***Osnove Windows operativnog sistema***
- ***Osnove hexadecimalno/decimalnih brojeva***
- ***Osnovni set ASM komandi***
- ***Osnovni crackerski alati: W32Dasm i Hiew***
- ***Osnovni crackerski alati: Soft ICE, Olly, SmartCheck, DeDe***
- ***Osnove PE struture***
- ***Osnovni crackerski alati: PeID, ResHacker, LordPe, ImpRec***
- ***Programski jezici: Visual Basic, Delphi, C++, MASM***
- ***Osnove kriptografije: SHA1, MD5, RSA, RC4,RC5,SkipJask***

Naravno ovaj redosled se odnosi na neki logicki redosled razmisljanja i ucenja koje bi trebalo primeniti da biste stekli konacan uvid u reverserske probleme. Naravno analogno ovome postoji i spisak vrsta meta koje bi trebalo da "razbijate". Ovaj spisak bi trebalo da izgleda ovako:

- ***Uklanjanje NAG ekrana***
- ***Menjanje skokova u cilju dolaska do poruka o tacnoj registraciji***
- ***Napredno patchovanje, ubijanje dialoga, CD zastite...***
- ***Pecanje serijskih brojeva uz pomoc debuggera***
- ***Jednostavno otpakivanje lakih meta: ASPack, UPX***
- ***Izmena samih meta u svoje keygeneratore***
- ***Pravljenje keygeneratora u nekom programskom jeziku***
- ***ASM ripovanje keygeneratora***
- ***Pravljenje bruteforcera***
- ***Odpakivanje naprednih protektora: ASProtect, ACProtect, SKVP, Armadillo***
- ***Napredni reversing meta u cilju razumevanja tudjeg algoritma***

Da li cete postati reverseri posle citanja ove knjige?
Ne, ali cete biti na dobrom putu da to postanete....

# ASM Basics

ASM je osnova svakog reverserskog problema, zbog cega je potrebno dobro znati makar osnovne ASM komande kako biste mogli da razumete kod koji se nalazi ispred vas. Osnovni i jedini alat koji ce nam trebati dalje u poglavlju je OllyDBG, ali cemo se za pocetak baviti teorijom.

# ASM for Crackers - Part I

ASM koji cu ja ovde objasniti nije ASM koji koriste programeri da bi pisali programe. Ne iako je vecina komandi ista ovde ce biti samo reci o sustini svake ASM komande sa kojom cete se sresti tokom reversovanja meta. Pocecemo sa malo matematike...

Posto se kao osnova svakog programa navode jednostavne matematicke operacije, stoga su osnovne ASM komande namenjene bas ovim operacijama.

*Dodeljivanje vrednosti* - je osnovna ASM komanda koja se koristi kako bi se nekim promenljivim (EAX,EBX,EDX,ECX,...) cija su imena definisana konstanta dodelila neka aritmeticka vrednost. Ovo bi u asembleru izgledalo ovako:

```
MOV EAX,1
```

a njeno matematicko znacenje je: EAX = 1.

*Sabiranje* - je osnovna matematicka operacija i svima je veoma dobro poznata. Siguran sam da znate da sabirate ali verovatno ne znate kako se vrsi sabiranje brojeva u asembleru. Primer sabiranja dve promenljive je:

```
ADD EAX,EBX
```

Ova jednostavna ASM komanda je ekvivalent matematickoj komandi sabiranja dva broja: EAX = EAX + EBX.

*Oduzimanje* - je takodje osnovna matematicka komanda koja bi u asembleru izgledala ovako:

```
SUB EAX,EBX
```

Ova jednostavna ASM komanda je ekvivalent matematickoj komandi oduzimanja dva broja: EAX = EAX - EBX.

*Mnozenje* - je cesto koristena komanda, a izgleda bas ovako:

```
IMUL EAX,EBX
```

Ova jednostavna ASM komanda je ekvivalent matematickoj komandi mnozenja dva broja: EAX = EAX * EBX.

Ovo su za pocetak samo neke osnovne ASM komande koje cemo iskoristiti da resimo neke jednostavne matematicke probleme. Za sada nema potrebe da brinete o tome koje cemo promenljive koristiti, za sada to necu objasnjavati jer nema potrebe, kasnije cu doci do toga, za sada je samo vazno da shvatite kako se izvrsavaju ASM komande.
Prvo cemo napisati program koji ce pomnoziti dva broja i na njihov proizvod dodati 4.

*Resenje:*
```
MOV EAX,3
MOV ECX,4
IMUL EAX,ECX
ADD EAX,4
```

Mislim da je jasno kako radi ovaj jednostavan program ali za svaki slucaj objasnicu zasto smo program napisali bas ovako. Prvo moramo dodeliti vrednost promenljivim EAX i ECX kako bismo imali sta da mnozimo. Ovo se radi u prva dva reda. Posle ovoga radimo standardno mnozenje dva broja, posle cega cemo u poslednjem redu na njihov proizvod dodati 4. Naravno rezultat izvrsenja ovog programa bice: 3 * 4 + 4 = 16.
Probacemo da uradimo modifikaciju ovog primera tako da program posle dodavanja 4 na proizvod oduzme 8 i rezultat pomnozi sa 4.

*Resenje:*
```
MOV EAX,3
MOV ECX,4
IMUL EAX,ECX
ADD EAX,4
SUB EAX,8
IMUL EAX,4
```

Kao sto vidimo rezultat ce biti: (((3 * 4) + 4) – 8) * 4 = 32. Naravno treba imati na umu da ako pisemo ASM programe svi brojevi moraju biti u heksadecimalnom obliku, stoga ce rezultat poslednjeg zadatka biti 20h a ne 32.

Posto smo uspesno savladali rad osnovnih matematickih operacija vreme je da objasnimo zasto i kako koristimo promenljive.
Kao i u matematici i u asembleru mozemo definisati promenljive kojima mozemo dodeljivati bilo koju aritmeticku vrednost. Jedino ogranicenje kada se radi sa asemblerom je da postoji vec definisani broj promenljivih koje mozemo koristiti. Ove promenljive se nazivaju registri i koriste se za sve asemblerske operacije. Neka imena ovih registara su vec pomenuta ali ceo spisak bi izgledao ovako: EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI,EIP. Ovi

registri iako se mogu koristiti za bilo koje operacije, bivaju dodeljeni specificnim tipovima operacija. Pa tako:
EAX sluzi za osnovne matematicke operacije,
EBX sluzi za osnovne matematicke operacije,
ECX sluzi kao brojac u petljama,
EDX sluzi kao registar u koji se snima ostatak deljenja i za druge stvari,
ESP sluzi kao pointer do raznih kontrola/stacka,
EBP sluzi kao pointer do delova memorije,
ESI sluzi kao slobodan registar,
EDI sluzi kao slobodan registar,
EIP sluzi kao pointer ka trenutnoj RVA adresi.
Ali ovo ne znaci da treba da shvatite ovaj spisak upotrebe registara crno-belo, ovo su samo najcesce upotrebe registara, ali se njihovo koriscenje moze menjati u zavisnosti od situacije do situacije. Videli smo da kao i promenljive u matematici, registri mogu uzimati bilo koju brojnu vrednost i koristiti se za matematicke operacije. Pre nego sto nastavimo sa objasnjavanjem ostalih ASM komandi potrebno je da shvatite razliku izmedju 32bit, 16bit i 8bit registara i da uvidite vezu izmedju njih.
Registri koje smo gore upoznali (EAX-EIP) su 32bitni registri. Ovo znaci da brojevi koje mogu sadrzati registri krecu se od 00000000 - FFFFFFFF, sto cini ove registre 32bitnim. Zbog kompatibilnosti sa starijim standardima 32bitni registri u sebi sadrze 16bitne i 8bitne registre. Ovo znaci da ako na primer EAX sadrzi heksadecimani broj FF443322 on predstavlja njegovu 32bitnu vrednost. Ali postoje drugi registri koji su u tesnoj vezi sa samim registom EAX. Ovi registri su AX (16bit), AH (8bit) i AL(8bit). Veza izmedju registara je data u sledecoj tablici:

| Registar | Prvi bajt | Drugi bajt | Treci bajt | Cetvrti bajt |
|---|---|---|---|---|
| EAX | FF | 44 | 33 | 22 |
| AX | -- | -- | 33 | 22 |
| AL | -- | -- | -- | 22 |
| AH | -- | -- | 33 | -- |

Kao sto se vidi u tablici postoje registri pomocu kojih se moze modifikovati ili samo pristupati razlicitim delovima 32bitnih registara. Iako je primer dat na registru EAX isto vazi i za ostale registre. Ovo je jako bitno jer je potrebno razumeti zasto se 32bitni registri u programima menjaju ako se promeni neki od 8bitnih ili 16bitnih registara. Primera radi objasnicu pristupanje delovima EAX registra na primeru:

```
MOV EAX,FF443322
MOV AX,1111
MOV AH,22
MOV AL,66
```

Rezultat izvrsavanja ovih ASM komandi po redovima bi bio:
EAX = FF443322, EAX = FF441111, EAX = FF442211, EAX = FF442266. Kao sto primecujete nemoguce je pristupiti prvom i drugom bajtu 32bitnog

registra EAX preko 16bitnih ili 8bitnih registara. Ovo je bitno kod pisanja keygeneratora iz razloga sto je bitno znati kako ce 16/8bitni registi promeniti konacnu vrednost nekog registra. Sada vam je sigurno jasno sta smo radili kada smo pisali jednostavne ASM programe gore. Posto za sada znamo samo cetiri ASM komande vreme je da prosirimo ovaj spisak sa dodatnim matematickim ASM komandama.

*Dodavanje +1* - je dodatna matematicka operacija koja radi dodavanje vrednosti 1 na bilo koji registar. U ASM-u to izgleda ovako:

```
INC EAX
```

Ova jednostavna ASM komanda je ekvivalent matematickoj EAX = EAX + 1. Sada se sigurno pitate zasto ovo jednostavno ne bismo uradili pomocu ASM komande ADD. Odgovor je da ASM komanda ADD zauzima od 4-8 bajtova u fajlu dok INC zauzima samo jedan. Nije velika usteda ali kompajleri tako sastavljaju exe fajlove.

*Oduzimanje -1* - je dodatna matematicka operacija koja radi oduzimanje vrednosti 1 od bilo kojeg registra. Ona u ASM izgleda ovako:

```
DEC EAX
```

Ova jednostavna ASM komanda je ekvivalent matematickoj EAX = EAX - 1. Ova komanda takodje zauzima samo jedan bajt u fajlu.

*Deljenje* - je matematicka operacija koju sam ostavio za kraj iz razloga sto je potrebno poznavanje registara kako biste razumeli kako se brojevi dele u ASMu. Primer deljenja dva broja:

```
MOV EAX,10
MOV ECX,4
MOV EDX,0
DIV ECX
```

Iako ovo izgleda malo komplikovano u stvari i nije. Ovde se desava sledece:
Prvo EAXu i ECXu dodeljujemo vrednosti, onda EDXu dodeljujemo vrednost nula jer ce EDX sadrzati ostatak pri deljenju, i na kraju delimo sa ECX. Ovo bi matematicki izgledalo ovako:

```
EAX = 16 (decimalno)
ECX = 4
EDX = 0
EAX = EAX / ECX
```

Da li je potrebno postavljati EDX na nula? Jeste jer ako ovo ne uradite izazvacete integer overflow i program ce se srusiti. Deljenje mozda izgleda malo komplikovano ali nije kada naucite da se uvek EAX deli sa nekim drugim registrom koji vi izaberete a u ovom slucaju ECXom.

Posto smo naucili sve osnovne matematicke komande provezbacemo ih na jednom primeru. Na primer napisacemo jedan ASM program koji ce sabrati dva broja, pomnoziti njihov zbir sa 4, dodati vrednost jedan na proizvod, od proizvoda oduzeti 6, podeliti oduzetu vrednost sa 3 i na kraju oduzeti jedan od rezultata.

*Resenje:*

```
MOV EAX,4
MOV ECX,3
ADD EAX,ECX
IMUL EAX,4
INC EAX
SUB EAX,6
MOV EDX,0
MOV ECX,3
DIV ECX
DEC EAX
```

Mislim da je svima jasno sta se ovde desava, ali ako nije evo matematickog resenja problema:

```
EAX = 4
ECX = 3
EAX = EAX + ECX
EAX = EAX * 4
EAX = EAX + 1
EAX = EAX - 6
EDX = 0
ECX = 3
EAX = EAX / ECX
EAX = EAX - 1
```

Razumevanje osnovnih matematickih operacija je kljucno kod resavanja osnovnih reverserskih problema. Posto smo uradili sve osnovne matematicke komande vreme je da uradimo logicke ASM komande. Ne plasite se ovim nazivom, jer logicke komande predstavljaju samo matematicke operacije sa logickim operatorima kao sto su NOT,AND,OR i slicni. Ovo je isto kao kada se u matematici koriste konjukcije, disjunkcije i slicni operatori. Rezultati ovih matematickih operacija su ili TRUE ili FALSE.

*AND* - je osnovna logicka komanda u ASMu. Koristi se kao logicki operator izmedju dva registra. Ona u ASM izgleda ovako:

```
AND EAX,ECX
```

Posle izvrsavanja ove komande EAX dobija vrednost koja korespondira matematickoj operaciji izmedju dva registra. Da biste detaljno razumeli sta radi ova logicka komanda napravicemo malu tabelu sa dva binarna broja i prikazacemo kako bi se racunao rezultat koji cemo dobiti primenom komande AND. Recimo da logicki pokusavamo da saberemo 3 i 5.

| Operacija AND | Broj decimalno | Broj binarno |
|---|---|---|
| EAX | 3 | 0011 |
| ECX | 5 | 0101 |
| Rezultat - EAX | 1 | 0001 |

Kao sto se vidi iz gornje tabele rezultat logickog sabiranja 3 AND 5 = 1. Zasto je to bas broj 1? AND komanda uporedjuje binarne brojeve bit po bit i na osnovu toga formira rezultat. Ako su bitovi 0 i 0, 0 i 1 ili 1 i 0 rezultat ce uvek biti 0, a jedino ako su dva bita koja se porede jednaka 1 onda ce rezultat za taj bit biti jednak 1. Zbog ovoga je 0011 AND 0101 = 1.

*OR* - je logicka komanda u ASMu. Koristi se kao logicki operator izmedju dva registra. Ona u ASM izgleda ovako:

OR EAX,ECX

Posle izvrsavanja ove ASM komande rezultat ce se postaviti u EAX, a vrednost koju ce dobiti registar EAX je prikazana u sledecoj tabeli:

| Operacija OR | Broj decimalno | Broj binarno |
|---|---|---|
| EAX | 3 | 0011 |
| ECX | 5 | 0101 |
| Rezultat - EAX | 7 | 0111 |

Kao sto se vidi iz gornje tabele rezultat logicke komande OR nad brojevima 3 i 5 je 7. Ali zasto je rezultat bas 7? OR komanda uporedjuje binarne brojeve bit po bit, na osnovu cega formira rezultat. Ako su nasi brojevi 0011 i 0101 rezultat ce biti 0111 jer OR komanda u rezultat stavlja 0 samo ako su oba bita jednaka 0 a ako je prvi ili drugi bit jednak 1 onda ce i rezultat biti 1.

*NOT* - je logicka komanda u ASMu. Koristi se kao logicki operator koji se primenjuje na jedan registar. Primer:

NOT EAX

Posle izvrsavanja ove ASM komande EAX dobija vrednost koja se moze procitati iz sledece tablice:

| Operacija NOT | Broj decimalno | Broj binarno |
|---|---|---|
| EAX | 3 | 0011 |
| Rezultat - EAX | 12 | 1100 |

Kao sto se vidi u tabeli NOT komanda samo invertuje bitove. To jest ako je bit jednak 0 onda ce u rezultatu biti jednak 1 i obrnuto. Ovo je krajnje jednostavna ASM komanda.

*XOR* - je jako bitna ASM komanda koja se zbog svoje reverzibilnosti koristi u svrhe enkripcije i dekriptcije. Ova komanda se primenjuje na dva registra a u ASM izgleda ovako:

XOR EAX,ECX

Posle izvrsavanja ove ASM komande EAX dobija vrednost koja se dobija XORom EAXa ECXom. Princip XORovanja se vidi iz sledece tablice:

| Operacija XOR | Broj decimalno | Broj binarno |
|---|---|---|
| EAX | 3 | 0011 |
| ECX | 5 | 0101 |
| Rezultat - EAX | 6 | 0110 |

Rezultat je 6, ali zasto? XOR je takva operacija koja uporedjuje bitove iz EAXa i ECXa tako da ako je bit iz sourcea (EAXa) razlicit od mete (ECXa) onda se u rezultat smesta 1 a ako su brojevi isti onda se u rezultat smesta 0. Zbog ovoga je 0011 XOR 0101 = 0110 to jest 3 xor 5 = 6. XOR funkcija je najbitnija zbog svoje reverzibilnosti. Ovo znaci da je 3 xor 5 = 6 ali i da je 6 xor 5 = 3 i da je 6 xor 3 = 5. Takodje bitna osobina XOR funkcija je da ako XORujemo jedan broj samim sobom uvek cemo dobiti nulu kao rezultat, to jest 3 xor 3 = 0.

Ovo su bile najbitnije logicke ASM funkcije. Da bismo utvrdili ono sto smo do sada naucili uradicemo jedan zadatak. Napisacemo ASM kod koji ce sabrati dva broja, zatim ce XORovati rezultat sa 7, na sta cemo logicki dodati 4, uradicemo negaciju, pa cemo uraditi logicko oduzimanje broja 5 od dobijene vrednosti, i na kraju cemo uraditi negaciju dobijene vrednosti.

*Resenje:*

```
MOV EAX,2
MOV ECX,3
ADD EAX,ECX
XOR EAX,7
AND EAX,4
NOT EAX
OR EAX,5
NOT EAX
```

Resenje ove jednacine bice NOT ((NOT ((2 + 3) xor 7) AND 4) OR 5) = 0. Naravno primetili ste da je u pitanju dupla negacija koja se ovde potire pa nema ni potrebe za njenim uvodjenjem u algoritam.

*LEA* - je matematicka komanda u ASMu. Koristi se kao matematicki operator za izvrsavanje vise operacija istovremeno. Ona u ASM izgleda ovako:

```
LEA EAX,DWORD PTR DS:[ECX*4+6]
```

Posle izvrsavanja ove ASM komande rezultat ce se postaviti u EAX, a vrednost koju ce dobiti registar EAX je jednaka rezultatu ECX*4+6. Naravno ovde se postuje princip matematicke prednosti operacija tako da ce se prvo izvrsiti mnozenje a tek onda sabiranje.

*SHL/SHR* – je binarna komanda koja se koristi za pomeranje bajtova u registrima u levu ili desnu stranu. Ova komanda bi u ASM izgledala ovako:

```
SHR AL,3
```

Dakle vidimo da da SHR/SHL komanda ima dva parametra: *destination* u vidu registra nad kojim se primenjuje operacija i *count* koji nam govori za koliko se brojevno pomera binarna vrednost AL registra. U praksi izvrsenje gornje komande, ako bi AL binarno bio jednak 01011011, bi izgledalo ovako:

```
01011011 – Originalni AL sadrzaj
00101101 – Pomeranje ALa u desno za jedno mesto
00010110 – Pomeranje ALa u desno za jos jedno mesto
00001011 – Pomeranje ALa u desno za jos jedno mesto
```

Svi bajtovi koji su pomeranjem dovedeni do kraja fizicke velicine registra bivaju istisnuti. Poslednji istisnuti bajt se snima kao carry-flag. Naravno ova operacija se moze primenjivati i na 8bitnim, 16bitnim i na 32bitnim registrima.

*ROL/ROR* – je binarna komanda koja se koristi za rotaciju bajtova u registrima u levu ili desnu stranu. Ova komanda bi u ASM izgledala ovako:

```
ROL AL,3
```

Dakle vidimo da da ROL/ROR komanda ima dva parametra: *destination* u vidu registra nad kojim se primenjuje operacija i *count* koji nam govori za koliko se brojevno pomera binarna vrednost AL registra. U praksi izvrsenje gornje komande, ako bi AL binarno bio jednak 01011011, bi izgledalo ovako:

```
01011011xxx – Originalni AL sadrzaj
x01011011xx – Pomeranje ALa u desno za jedno mesto
xx01011011x – Pomeranje ALa u desno za jos jedno mesto
xxx01011011 – Pomeranje ALa u desno za jos jedno mesto
11001011    – Konacan izgled posle rotacije
```

Kao sto vidite rotacija je slicna siftingu sa razlikom sto se istisnuti podaci vracaju na pocetak registra rotiranu u levu (ROL) ili desnu (ROR) stranu.

Sada smo naucili najveci deo standardnih ASM komandi sa kojima cete se sretati prilikom reversinga aplikacija, ali ovo ne znaci da smo zavrsili sa ASMom, naprotiv tek pocinjemo sa zanimljivostima vezanim za ASM programe.

# ASM for Crackers - Part II

Do sada ste naucili kako se koriste osnovne matematicke ASM komande, sada je vreme da naucite kako da koristite skokove, poredjenja i slicne programerske stvari iz ASMa.

*Zero Flag* - je jednobitna memorijska alokacija koja moze drzati samo dve vrednosti, ili 0 ili 1. Zasto se koristi zero flag? Posto postoje ASM komande koje porede dve vrednosti, kao rezultat tog poredjenja se postavlja zero flag usled cega ce se odredjene komande izvrsiti ili ne. Ovo ce vam sigurno biti jasnije kada dodjete do dela koji objasnjava uslovne skokove.

*CMP* - je osnovna komanda koja se koristi za poredjenje dve brojevne vrednosti. Poredjenje se moze vrsiti izmedju dva registra ili izmedju registra i broja. Ovako oba ova slucaja izgledaju u ASMu:

```
CMP EAX,EBX
CMP EAX,1
```

Ovde prvi slucaj poredi EAX sa sadrzajem EBXa. Ako su EAX i EBX jednaki onda ce zero flag biti postavljen na 1 a ako nisu onda ce zero flag biti jednak 0. CMP je ekvivalent komandi IF iz raznih programskih jezika.

*TEST* - je napredna komanda koja se koristi za poredjenje dve brojevne vrednosti. Poredjenje se vrsi na taj nacin sto se poredjeni registri logicki dodaju jedan na drugi a na osnovu rezultata koji se ne snima ni u jedan registar postavlja se zero flag. Ova ASM komanda je oblika:

```
TEST EAX,EAX
```

Naravno, kao i kod CMP, komande mogu se porediti ili registri medjusobno ili registri sa brojevima. U slucaju koji sam dao kao primer, ako je EAX jednak 0 onda ce zero flag biti jednak 1, a ako je EAX jednak 1 onda ce zero flag biti 0. Ovo je bitno jer se vecina provera serijskih brojeva zasniva na poredjenju registra EAX sa samim sobom.

*JMP* - je jedna od mnogih varijanti iste komande za uslovno / bezuslovne skokove. Ovi skokovi predstavljaju skakanje kroz kod od jedne do druge virtualne adrese. Ovakvi skokovi su najcesce pratioci gore opisanih funkcija za poredjenje registara i brojeva. Primer jedne bezuslovne jump komande koja se uvek izvrsava je:

```
JMP 00401000
```

Posle izvrsenja ove ASM komande program ce izvrsavanje nastaviti od adrese 00401000. Ovaj skok se zove bezuslovni jer nije bitno koja je vrednost zero flaga da bi se skok izvrsio, to jest skok se bez obzira na bilo koji parametar ili

registar uvek izvrsava. Postoje razlicite varijacije skokova koje zavise od zero flaga. JMP komande koje zavise od vrednosti zero flaga (*t.j. od njegove vrednosti zavisi da li ce se skok izvrsiti ili ne*) nazivaju se uslovni skokovi. Primeri ovih uslovnih skokova su JE *(skoci ako je zero flag jednak 1)* i JNE *(skoci ako je zero flag jednak 0)* Spisak svih varijanti ASM skokova:

| **Hex:** | **Asm:** | **Znaci:** |
|---|---|---|
| 75 ili 0F85 | JNE/JNZ | skoci ako nije jednako |
| 74 ili 0F84 | JE | skoci ako je jednako |
| EB | JMP | bezuslovni skok |
| 77 ili 0F87 | JA | skoci ako je iznad |
| 0F86 | JNA | skoci ako nije iznad |
| 0F83 | JAE | skoci ako je iznad ili jednako |
| 0F82 | JNAE | skoci ako nije iznad ili jednako |
| 0F82 | JB | skoci ako je ispod |
| 0F83 | JNB | skoci ako nije ispod |
| 0F86 | JBE | skoci ako je ispod ili jednako |
| 0F87 | JNBE | skoci ako nije ispod ili jednako |
| 0F8F | JG | skoci ako je vece |
| 0F8E | JNG | skoci ako nije vece |
| 0F8D | JGE | skoci ako je vece ili jednako |
| 0F8C | JNGE | skoci ako nije vece ili jednako |
| 0F8C | JL | skoci ako je manje |
| 0F8D | JNL | skoci ako nije manje |
| 0F8E | JLE | skoci ako je manje ili jednako |
| 0F8F | JNLE | skoci ako nije manje ili jednako |

Posto smo naucili kako se u ASMu radi takozvano programsko grananje, iskoristicemo do sada steceno znanje kako bismo resili par jednostavnih matematickih zadataka. Napisacemo program koji ce izracunati povrsinu trougla za uneta dva parametra, za unetu stranicu i visinu. Ako je povrsina trougla manja ili jednaka od 6, onda cemo dodati 3 u izracunatu vrednost za povrsinu, posle cega cemo u slucaju bilo koje vrednosti povrsine oduzeti jedan od rezultata. Povrsina trougla se racuna po obrascu: P = (a * h) / 2.

*Resenje:*

```
        MOV EAX,3
        MOV ECX,4
        XOR EDX,EDX
        IMUL EAX,ECX
        MOV ECX,2
        DIV ECX
        CMP EAX,6
        JLE tri
        JMP end
tri:
        ADD EAX,3
end:
        DEC EAX
```

Za slucaj da vam nije jasno sta se ovde desava objasnicu to u programskom jeziku C ali i matematicki. Ovako bi to izgledalo u C++:

```
#include <iostream>

using namespace std;

int main (int argc, char *argv[])
{
  int eax;                    // Definisi celobrojnu promenljivu
  int ecx;                    // Definisi celobrojnu promenljivu
  int edx;                    // Definisi celobrojnu promenljivu
  printf("Unesite bazu trougla: ");
   cin >> eax;                // Unesi EAX iz konzole
  printf("Unesite visinu: ");
   cin >> ecx;                // Unesi ECX iz konzole
   edx = 0;                   // EDX = 0
   eax = eax * ecx;           // EAX = EAX * ECX
   ecx = 2;                   // ECX = 2
   eax = eax / ecx;           // EAX = EAX / ECX
  if(eax <= 6){               // Ako je EAX <= 6
   eax = eax + 3;             // Onda EAX = EAX + 3
  }                           // Kraj uslova ako
   eax = eax - 1;             // EAX = EAX - 1
  printf("Rezultat je: ");    // Odstampaj rezultat na ekran
  printf("%i",eax);
  return 0;
}
```

Matematicko resenje ovog zadatka bi bilo:
EAX = 3
ECX = 4
EDX = 0
EAX = EAX * ECX
ECX = 2
EAX = EAX / ECX
Ako je EAX <= 6 onda EAX = EAX + 3
EAX = EAX - 1

Kao sto vidite iz ovog C++ source-a ASM JMP komande se pojavljuju na mestima gde se u C-u ali i u drugim programskim jezicima nalaze IF uslovne klauzule. Nezamislivi su programi bez poredjenja ili programi bez uslovnih skokova. Na ovakve mete necete naici prilikom reverserske prakse, stoga je izuzetno bitno da znate kako se izvrsavaju skokovi...

# ASM for Crackers - Part III

Do sada ste naucili kako se koriste osnovne matematicke operacije u ASMu, kako se radi programsko grananje,... Sada cemo nauciti kako se koristi STACK.

*STACK* - predstavlja deo memorije koji se koristi za privremeno smestanje podataka. Ovi podaci se najcesce koriste za potrebe razlicitih funkcija koje se nalaze unutar samog PE fajla. O stacku treba razmisljati kao gomili ploca naslaganih jedna na drugu. Ove ploce su poredjane tako da je ploca sa brojem 1 sa samom vrhu ove gomile dok se poslednja ploca nalazi na samom dnu. Ovo znaci da se podaci na stack salju u obrnutom redosledu, to jest da se prvo prosledjuje poslednji parametar, a tek na kraju se prosledjuje prvi. Isto kao da slazemo gomilu ploca, slazuci ploce jednu na drugu, prvo cemo postaviti poslednju plocu, pa cemo polako slagati ploce jednu na drugu sve dok ne dodjemo do prve ploce. ASM komanda koja se koristi za slanje podataka na stack je PUSH. Za slucaj da vam ovo nije jasno dacu kratak primer:

Windowsova Api funkcija GetDlgItemText zahteva sledece parametre:
(1) Handle dialog boxa
(2) Identifikator kontrole iz koje se cita tekst
(3) Adresa u koju se smesta tekst
(4) Maximalna duzina teksta
Stoga ce iscitavanje teksta u ASMu izgledati ovako:

```
MOV  EDI,ESP            ; Handle dialog boxa se smesta u EDI
PUSH 00000100           ; PUSH (4) Maximalna duzina teksta
PUSH 00406130           ; PUSH (3) Adresa u koju se smesta tekst
PUSH 00000405           ; PUSH (2) Identifikator kontrole
PUSH EDI                ; PUSH (1) Handle dialog boxa
CALL GetDlgItemText     ; Pozivanje funkcije koja vraca tekst
```

Mislim da je svima jasan ovaj primer. Ako vas buni deo koji se odnosi na handle, njega jednostavno ne posmatrajte, vazno je da razumete da svakoj funkciji koja ima ulazne parametre prethode PUSH ASM komande koje joj te parametre prosledjuju u obrnutom redosledu. Kao sto se vidi iz primera PUSH komanda ima samo jedan parametar i on moze biti ili broj ili registar.

*CALL* - sigurno ste se pitali sta radi ASM funkcija koju sam pomenuo na samom kraju dela o STACKu. CALLovi predstavljaju interne podfunkcije koje predstavljaju zasebnu celinu koda koja je zaduzena za izvrsavanje neke operacije. Ove funkcije mogu ali i ne moraju imati ulazne parametre na osnovu kojih ce se racunati neki proracun unutar same funkcije. Ako funkcija ima ulazne parametre samom CALLu ce prethoditi PUSH komande, a ako funkcija nema ulazne parametre onda CALLu nece predhoditi ni jedna PUSH komanda. Da biste shvatili razliku izmedju CALL koji ima ulazne parametre i onog koji to nema napisacu dve uopstene CALL funkcije:

- Slucaj - bez ulaznih parametara:

```
...
00403200:       CALL 00401001
00403209:       DEC EAX
...
00401001:       INC EAX
00401002:       ... neke ASM operacije
00401100:       RET
...
```

Kao sto se vidi u primeru funkciji CALL ne predhode nikakve PUSH komande iz cega zakljucujemo da CALL na adresi 00403200 nema ulaznih parametara. Sigurno ste primetili da se na prvoj adresi CALLa, 00401001 nalazi komanda INC EAX. Ova komanda kao i sve ostale komande u CALLu su proizvoljne i CALL se moze koristiti za bilo sta. Ono sto vas sigurno zanima je za sta se koristi komanda RET na samom kraju CALLa. Isto kao sto mora postojati prva komanda u samom CALLu koja zapocinje sekvencu komandi, tako mora postojati i poslednja komanda u CALLu koja nalaze programu da se vrati iz CALLa i nastavi sa izvrsavanjem programa. Postoje mnoge varijacije iste ove komande, kao sto su npr. RET 4, RET 10 i slicne, ali sve one izvrsavaju jednu te istu operaciju vracanja iz CALLa.

Kako se izvrsavaju napisane komande? Jednostavno, prvo se pozove doticni CALL, posle cega se izvrse sve komande u njemu zakljucno sa komandom RET. Posle ovoga se program vraca iz CALLa i sa izvrsavanjem nastavlja od adrese 00403209, to jest izvrsava se DEC EAX komanda.

- Slucaj - sa ulaznim parametrima:

```
...
00403198:       PUSH EAX
00403199:       PUSH EBX
00403200:       CALL 00401001
00403209:       DEC EAX
...
00401001:       PUSH EAX
00401002:       PUSH EBX
00401003:       ... neke ASM operacije
00401090:       POP EBX
00401092:       POP EAX
00401100:       RET
...
```

Kao sto se vidi u ovom primeru CALLu prethode dve PUSH komande sto znaci da funkcija ima dva ulazna parametra. Ovi ulazni parametri se nalaze privremeno na STACKu. Primeticete da se parametri funkciji predaju u obrnutom redosledu, prvo se predaje EAX, a tek onda EBX. Naravno ovo je samo tu da ilustruje kako se parametri predaju funkciji, jer u stvarnosti redosled registara nije bitan, bitan je redosled njihovog slanja na STACK. Unutar CALLa je sve isto kao i u proslom primeru ali cete na kraju CALLa primetiti nove POP komande. Sta su to POP komande? POPovi su neophodni iz razloga sto ako se podaci na pocetku funkcije salju na STACK, na njenom kraju se svi uneti parametri sa STACKa moraju skinuti pomocu komande POP. Dakle POP komanda sluzi za skidanje parametara sa STACKa.

Primecujete da se parametri sa STACKA skidaju u obrnutom redosledu od njihovog ulaska. Posmatrajte ovo kao odraz registara u ogledalu. I na samom kraju posle izvrsenja CALLa, program nastavlja sa daljim izvrsavanjem koda od prve adrese koja se nalazi ispod poziva CALLu, to jest od 00403209, DEC EAX komande.
Sada kada smo naucili kako se koriste PUSH,CALL,POP,RET komande, vreme je da napisemo jedan program. Napisacemo program koji ce pomnoziti dva broja u jednom CALLu i vratiti rezultat mnozenja.

*Resenje:*

```
            MOV EAX,3
            MOV EBX,4
            PUSH ECX
            PUSH EAX
            CALL mnozenje
            RET
mnozenje:   PUSH ECX
            PUSH EAX
            IMUL EAX,EBX
            MOV EDX,EAX
            POP EAX
            POP ECX
            MOV EAX,EDX
            RET
```

Mislim da je svima jasno zasto je CALL struktura ovako napisana. Bitno je samo da zapazite par detalja. Prvo i najvaznije je da se iste PUSH komande nalazi i ispred CALLa i u samom CALLu. Drugo da se POP komande primenjuju obrnuto od PUSH komandi. Trece da se rezultat smesta privremeno u EDX da bi se tek posle POP EAX komande vratio u EAX. Zasto? Zato sto ce POP EAX komanda vratiti vrednosti 3 u EAX pa ce rezultat mnozenja biti izgubljen. Stoga se tek posle izvrsenja POP EAX komande EAXu dodeljuje njegova prava vrednost. Kada se napokon izvrsi CALL, EAX ce sadrzati vrednost mnozenja, a sledeca komanda koja ce se izvrsiti po izlasku iz CALLa je druga RET komanda.
Naravno postoji veci broj tipova CALLova ali ovo je uopsteni primer na kome se moze razumeti sama svrha CALLova i kako se to vracaju vrednosti iz CALLova.

# ASM for Crackers - Part IV

Poslednje poglavlje o osnovama ASMa je namenjeno objasnjenju stringova i pristupanju memoriji iz ASMa.

*Stringovi* - predstavljaju niz ASCII slova koja zajedno cine jednu recenicu ili jednu rec. Duzina stringova moze biti proizvoljna ali ono sto je karakteristicno za stringove je da se svaki string mora zavrsavati sa istim 00h bajtom. Posto ovaj bajt ne predstavlja ni jedno slovo, stringovi se lako razlikuju od stalog koda. Primera radi evo jednog stringa:

```
00403040                59 6F 75 72              Your
00403048  20 6E 61 6D 65 20 6D 75                name mu
00403050  73 74 20 62 65 20 61 74                st be at
00403058  20 6C 65 61 73 74 20 66                least f
00403060  69 76 65 20 63 68 61 72                ive char
00403068  61 63 74 65 72 73 20 6C                acters l
00403070  6F 6E 67 21 00                         ong!.
```

Kao sto se vidi i svaki karakter stringa ima svoju adresu ali je adresa celog stringa adresa prvog slova. Kada se cita string koji pocinje od adrese 00403040 on se cita od tog prvog bajta pa sve do poslednjeg 00 bajta. Zakljucujemo da stringovi predstavljaju sve tekstualne poruke koje se nalaze u nekom programu.

*Memory* - Pomocu ASMa je moguce veoma lako pristupiti svim adresama exe-a koji se trenutno izvrsava. Postoji veci broj komandi i varijacija istih tako da cu ja navesti samo najcesce koriscene komande.

Postoje dve vrste manipulacije memorijom:

1) Manipulacija samo jednog bajta
2) Manipulacija niza bajtova

*BYTE PTR* - Prvo cu vam objasniti kako se koristi komanda koja se ponasa kao referenca ka zadatom bajtu. Za ovo se koristi samo jedna komanda u obliku:

BYTE PTR DS:[RVA adresa]

U ovoj komandi sve je konstanta osim RVA adrese koja moze biti ili adresa ili registar. Posto je ovo samo deo komande moze se koristiti sa svim ostalim komandama koje imaju jedan ili dva parametra. Ovo znaci da se BYTE PTR moze koristiti uz MOV,XOR,ADD,IMUL,...

*DWORD PTR* - Za razliku od prosle komande ova komanda se koristi za pristupanje nizu bajtova. Za ovo se koristi samo jedna komanda u obliku:

DWORD PTR DS:[RVA adresa]

U ovoj komandi sve je konstanta osim RVA adrese koja moze biti ili adresa ili registar. Posto je ovo samo deo komande moze se koristiti sa svim ostalim

komandama koje imaju jedan ili dva parametra. Ono sto je jako bitno da znate da ako koristite komandu DWORD PTR u obliku:

```
MOV DWORD PTR:[EAX],019EB
```

Morate da vodite racuna o tome da se bajtovi koji ce se snimiti na lokaciju na koju pokazuje EAX snimiti u obrnutom redosledu. To jest da biste na lokaciju EAX snimili recimo EB 19 morate komandu formulisati tako da prvo stavite 0 a tek onda obrnuti redosled bajtova 19 EB. Obratite paznju da se ne okrecu brojevi iz bajtova nego samo njihov redosled. Naravno ovo nije slucaj kada se koristi BYTE PTR komanda jer se ona odnosi samo na jedan bajt.
Analiziracemo jedan primer da bismo shvatili kako radi ova manipulacija memorijom.

```
00401154    |> /8A10           /MOV DL,BYTE PTR DS:[EAX]
00401156    |. |2AD1           |SUB DL,CL
00401158    |. |3813           |ADD DL,1
0040115A    |. |75 18          |JNZ 00401174
0040115C    |. |40             |INC EAX
0040115D    |. |43             |INC EBX
0040115E    |.^\E2 F4          \JNE 00401154
```

Recimo da se u EAXu nalazi neka adresa na kojoj se nalazi string ‘ap0x’, naravno bez navodnika. Posto se na adresi 00401154 koristi komanda koja ce u DL registar postaviti samo jedan bajt, vidimo da se radi o jednostavnom jednobajtnom slucaju. Primeticete takodje da se EAX registar stalno povecava za jedan, pomocu INC EAX komande. Ovo znaci da ce se za svaki prolaz ovog loopa u registar DL stavljati jedno po jedno slovo iz naseg stringa sve dok se sva slova iz stringa ne iskoriste. Kada se ovo desi program ce nastaviti sa izvrsavanjem koda koji se nalazi ispod adrese 0040115E.
Zasto je bitno razumevanje manipulacije memorijom?
Bitno je iz razloga sto se pomocu direktnog pristupa ASM kodu iz samog exe fajla delovi koda mogu polymorphno menjati ili se moze proveravati da li je sam kod na nekoj adresi modifikovan, mogu se napraviti funkcije cije ce izvrsenje zavisiti od samog sadrzaja koda i tako dalje. Medjutim ako samo zelite da se bavite reversingom stringovi i manipulacija memorijom su vam bitni jer se vecina algoritama koji se koriste za proveru serijskih brojeva zasnivaju na pristupanju delovima ili celom stringu nekog unesenog podatka. Ovaj podatak moze biti ime, serijski broj... Sto su * PTR komande veoma bitne za reversere...

# ASM for Crackers - Part V

Do sada smo se upoznali sa standardnim komandama koje se koriste za manipulaciju registrima, a sada cemo prosiriti nase vidike i naucicemo sta su FPU registri i kako se koriste.

Prvo: "Sta su to FPU registri?". Oni predstavljaju funkciju procesora za baratanje ciframa sa pokretnim zarezom. Ove cifre su za razliku od 32bitnih registara (EAX,EBX,…) predstavljene u decimalnom obliku. Izuzetno je bitno da znate da su 32bitni registri predstavljeni u heksadecimalnom obliku, a da su FPU registri predstavljeni decimalno. Bez obzira na ovu razliku FPU registri imaju dosta slicnosti sa 32bitnim registrima.

Prva slicnost je da kao i kod 32bitnih registara i FPU registri imaju memorijske lokacije koje se mogu ispunjavati brojevima, nad kojima se kasnije mogu izvrsavati matematicke operacije. Kod 32bitnih registara ove memorijske lokacije su se zvale EAX,EBX,… a kod FPU registara te lokacije se zovu ST0,ST1,ST2,…ST7. Njih mozete viditi u istom delu Ollya kao i 32bitne registre samo ako naslov tog dela prozora bude podesen na Registers (FPU), a ovo mozete uraditi klikcuci na naziv tog dela prozora. Razlika izmedju ove dve vrste registara su u velicini broja koji mogu da drze. 32bitni registri mogu da uzimaju vrednosti od 0000000 – FFFFFFFF, dok FPU registri mogu imati mnogo vece brojeve kao vrednosti.

Sledeca slicnost izmedju ove dve vrste registara je u tome da se nad obe vrste registara mogu vrsiti slicne matematicke operacije. Pocecemo od najjednostavnijih:

## Basic Math operands

*FPU inicijalizacija* - je osnovna FPU komanda koja se koristi kako bi program saopstio procesoru da sledi niz komandi koje ce pristupati FPU registrima. Iako se ova komanda, FINIT, koristi i za direktno postavljanje inicijalnih FPU flagova, ona moze biti izostavljena i FPU registrima se moze pristupati i bez nje.

*Dodeljivanje vrednosti* - je osnovna FPU komanda koja se koristi kako bi se nekim promenljivim (ST0,ST1,...) cija su imena definisana konstanta dodelila neka vrednost. Ova komanda izgleda ovako:

FLD *source - ucitavanje REAL promenljive[primer: 1,22 ili 1,44…]*
FILD *source - ucitavanje INTEGER promenljive[celobrojna, primer 1;2]*

gde je FLD komanda a source destinacija sa koje se cita niz bajtova koji se pretvara u real promenljivu i smesta po defaultu u prvi raspolozivi memorijski prostor, koji je u ovom slucaju ST0. Naravno ako se FLD izvrsi jos jednom ST1 ce dobiti novu vrednost. Primer ovakve komande u praksi bi bio:

```
FLD TBYTE PTR DS:[403197]
```

A znacenje komande je prebaci Tabelu bajtova sa adrese 00403197 u ST0 memoriski registar. Tabela bajtova je oblik slican DWORDu ali je veci jer predstavlja niz bajtova veci od cetiri.

*Sabiranje* - je osnovna FPU matematicka komanda koja se koristi kako bi se sabrale dve promenljive. Primera radi mogu se sabrati ST0 i ST1 tako da ST0 drzi rezultat sabiranja. Za ovo se koristi komanda:

FADD *destination, source – Dodavanje REAL promenljive*
FIADD *destination, source – Dodavanje INTEGER promenljive [source]*

Primer upotrebe ove komande bi bio:

```
FADD ST(0), ST(1);
```

koja u ST0 registar stavlja rezultat sabiranja registara ST0 i ST1. Ako se umesto FADD komande koristi FIADD onda se source parametar prvo pretvara u ExtendedReal a tek onda sabira sa ST0.

*Oduzimanje* - je osnovna FPU matematicka komanda koja se koristi kako bi se oduzele dve promenljive. Primera radi moze se od ST0 oduzeti ST1 tako da ST0 drzi rezultat oduzimanja. Za ovo se koristi komanda:

FSUB *destination, source – Oduzimanje REAL promenljive*
FISUB *destination, source – Oduzimanje INTEGER promenljive[source]*

Primer upotrebe ove komande bi bio:

```
FSUB ST(0), ST(1);
```

koja u ST0 registar stavlja rezultat oduzimanja ST0 – ST1. Ako se umesto FSUB koristi FISUB onda se source parametar pretvara u ExtendedReal tip a tek onda se oduzima od destination parametra.

*Mnozenje* - je osnovna FPU matematicka komanda koja se koristi kako bi se pomnozile dve promenljive. Primera radi ST0 i ST1 mozemo pomnoziti tako da rezultat mnozenja sadrzi ST0. Za ovo se koriste komande FMUL i FIMUL.

Primer upotrebe ove komande bi bio:

```
FMUL ST(0), ST(1);
```

a njen rezultat bi bio ST0 = ST0 * ST1; Ako se umesto FMUL koristi FIMUL onda se drugi parametar prvo pretvara u ExtendedReal pa tek onda mnozi sa ST0.

*Deljenje* - je osnovna FPU matematicka komanda koja se koristi kako bi se podelile dve promenljive. Primera radi ST0 i ST1 mozemo podeliti tako da ST0 dobije vrednost kolicnika. Za ovo se koriste komande FDIV i FIDIV.

Primer upotrebe ove komande bi bio:

```
FDIV ST(0), ST(1);
```

a njen rezultat bi bio ST0 = ST0 / ST1; Ako se umesto FDIV koristi FIDIV onda se drugi parametar prvo pretvara u ExtendedReal pa tek onda deli sa ST0.

*Kvadratni koren* – je matematicka operacija ciji je rezultat takav broj koji mnozenjem samog sebe daje broj iz kojeg je vadjen koren. Dakle ako je 5 * 5 = 25, kvadratni koren od 25 je 5. Ova operacija se u ASM poziva samo sa jednim parametrom koji je ujedno i destination i source. Ova komanda se zove FSQRT.

*Apsolutno* – je matetaticka operacija koja preslikava skupove negativnih vrednosti u njihove pozitivne slike. Ovo znaci da ce posle primene ove operacije na bilo koji broj vrednost tog broja uvek imati pozitivnu vrednosti. Dakle ako je broj bio -1,22 posle primenjivanja ove komande na ovaj broj rezultat ce biti 1,22. Ova komanda se zove FABS i moze se primenjivati pojedinacno na brojeve i na FPU registre.

*Promena znaka* – je ekvivalent mnozenja bilo kojeg broja sa vrednoscu -1. Dakle ako je neki broj ili registar bio negativan, postace pozitivan, i obrnuto. Ova komanda se zove FCHS i za parametar ima samo jednu vrednost, ili neki broj ili neki FPU registar.

*Sinus/Kosinus* – su osnovne trigonometrijske komande koje u ASMu izgleda ovako:

```
FSIN ST(0);
FCOS ST(0);
```

Dakle ove komande imaju samo jedan parametar koji ujedno predstavlja i source i destination izvrsenja komande. Dakle sinus/kosinus se racuna na registru ST0 (u ovom slucaju) i rezultat komande se takodje smesta u isti registar. Takodje postoji kombinovana komanda FSINCOS.

## More operands

*No Operation* – je poznata ASM komanda koja se koristi za popunjavanje praznog prostora i zove se FNOP, a funkcionalni je ekvivalent ASM komandi NOP koju smo vec upoznali.

*Test* – je poznata ASM komanda koja se koristi za logicko poredjenje vrednosti i najcesce se u ASM koristi u obliku TEST EAX,EAX gde se naravno umesto EAX moze koristiti bilo koji drugi registar. Kao rezultat ove ASM komande se dobija postavljanje zero flaga na 1 ako je EAX jednak 0 i obrnuto. Kao ekvivalent ove komande za FPU se koristi FTST.

*Zamena* – je poznata ASM komanda koja se koristi za razmenu vrednosti izmedju dva registra. Ova komanda ima svoj FPU ekvivalent i on je FXCH. Ona ce koristi na sledeci nacin:

```
IF number-of-operands is 1
  THEN
          temp <- ST(0);
          ST(0) <- SRC;
          SRC <- temp;
  ELSE
          temp <- ST(0);
          ST(0) <- ST(1);
          ST(1) <- temp;
```

*Poredjenje* – je poznata ASM komanda koja se koristi za poredjenje dve vrednosti ili dva registra. Ona ima svoj FPU ekvivalent kada se radi o poredjenju integer ili real vrednosti. Ako se porede integer vrednosti onda se koristi FICOM komanda a ako se porede real vrednosti onda se koristi FCOM vrednost.

FPU registri su vazni za matematicke operacije sa pokretnim zarezom ili za racunanje vrednosti koje su znato vece od standardnih integer vrednosti. Mada FPU registri imaju svoje prednosti i svoje primene oni se ne primenjuju cesto u reverserskoj praksi. Ova kratka tabela najcesce koriscenih FPU komandi je dodata u knjigu jer je moguce da cete se sresti sa nekim od ovih komandi prilikom reversovanja nekih crackmea i/ili nekih enkripcija.

**NAPOMENA:** Ako zelite da razumete sve sto se nalazi u knjizi razumevanje osnovnih ASM komandi je neophodno za dalje citanje knjige. Ako niste sve razumeli ili ste nesto propustili predlazem da se vratite i procitate ovaj deo knjige opet.

## Reading time table:

| Chapter | Required Level | Minimum reading time | |
|---|---|---|---|
| Intro to Cracking | newbie | 4 days | 1 week |
| NAG Screens | newbie | 1 day | 1 week |
| Cracking Serials | newbie | 2 days | 1 week |
| Making KeyGens | advanced newbie | 1 week | 2 weeks |
| CD Checking | advanced newbie | 1 day | 5 days |
| Code Hacking | advanced coder | 3 days | 1 week |
| "Getting caught" | advanced newbie | 3 days | 1 week |
| Cracking it | newbie | 2 days | 1 week |
| Decrypt me | expert coder | 1 week | 3 weeks |
| Unpacking | advanced newbie | 1 week | 3 weeks |
| Patching | advanced newbie | 1 day | 3 days |
| Nightmare | reverser | 1 week | 4 weeks |
| Tricks of Trade | newbie | 1 day | 1 day |

## Download links:

***Debugger*** – OllyDBG v1.10 http://www.Ollydbg.de
***Disassembler*** - W32dasm89 http://www.exetools.com
***PE identifikator*** – PeID 0.93 http://peid.has.it
***Hex Editor*** – Hiew 6.83 http://www.exetools.com
***Resource Viewer*** - Res. Hacker http://rpi.net.au/~ajohnson/resourcehacker
***Import rekonstrukter*** - Import Reconstructer 1.6 http://www.wasm.ru
***Process dumper*** – LordPE 1.4 http://y0da.cjb.net

**Ostali alati:**
***.ap0x Patch Creator RC3*** – http://ap0x.headcoders.net
***Olly2Table 0.1alfa*** – http://ap0x.headcoders.net
***HexDecChar 0.4alfa*** – http://ap0x.headcoders.net
***the aPE 0.0.6beta*** – http://ap0x.headcoders.net

**Napomena:** Ovo su samo neki alati koji ce biti korisceni u knjizi. Njih smatram osnovnim i pre nego sto pocnete sa daljim citanjem ove knjige trebalo bi da ih nabavite.

# Tools of Trade

Kao i za svaku drugu delatnost na kompjuteru za reversni inzenjering su vam potrebni neki alati (*programi*) kako biste mogli brze i lakse da dodjete do informacije koja vam treba. Vecina alata koje cu vam ja ovde preporuciti mogu se besplatno preuzeti sa interneta i kao freeware proizvodi koristiti i distribuirati. Pre nego sto pocnem sa listom programa koje cete morati preuzeti sa interneta prvo cu u par recenica objasniti kakvi su nam to alati potrebni i za sta oni sluze.

***Debugger*** – Ovo je osnovni alat svakog reversera ali i svakog programera koji zeli da brzo i lako eliminise greske iz svog koda. Ono sto nam debugger pruza je mogucnost da nadgledamo izvrsavanje naseg ili tudjeg koda bas onako kako ga procesor vidi. Da, to znaci da cete morati da naucite osnove assemblera (*masinskog jezika*) kako biste mogli da razumete i kontrolisete izvrsavanje koda. Ovakav tekst sam vec napisao i objavio na sajtu *www.EliteSecurity.org* a nalazi se i u ovom izdanju knjige na strani 9 (*ovo je dopunjeno izdanje*) cije je citanje neophodno radi lakseg snalazenja i razumevanja tekstova iz ove knjige.

***Disassembler*** – Ovo je dodatni alat za debugger. Naime ako vam debugger ne da dovoljno informacija o "meti" onda mozete koristiti neke od disassemblera kako biste lakse uocili informaciju koja vam treba. Sa vremenom cete sve manje koristiti ovaj alat posto cete se naviknuti na asemblerov kod tako da vam ovi alati nece biti potrebni.

***PE identifikatori*** – Ne dajte da vas ovaj naslov zbuni, PE fajlovi su samo obicni exe fajlovi koji mogu da sadrze neki dodatan kod, koji je najcesce paker koji sluzi za smanjenje velicine exe fajla. Posto postoji veliki broj ovakvih pakera i enkriptera pa potrebni su posebni programi za prepoznavanje istih. Naravno ovo se donekle moze raditi i rucno sto cu vas takodje nauciti.

***Hex Editori*** – su alati koji nam daju tacan izgled fajla na hard disku i sluze za fizicko menjanje koda za razliku od menjanja koda u memoriji sto nam omogucuju debuggeri.

***Resource Vieweri*** – Sluze za pregled, ekstrakciju, izmenu ili dodavanje exe resursa. Resursi su podaci koji su ukljuceni u sam exe fajl a mogu biti slike, dijalozi, multimedija, stringovi ili neki drugi tipovi podataka. Ovi programi u principu nisu neophodni ali nam mogu olaksati posao.

***Process dumperi*** – Sluze prevashodno kod otpakivanja zapakovanih PE fajlova i omogucavaju nam da celokupnu "sliku" nekog aktivnog programa snimimo na hard disk.

***Import rekonstrukteri*** – su programi koji sluze za popravljanje pogresnih ili nedefinisanih poziva ka windows api funkcijama.

# Configuring Tools of Trade

Vecina gore navedenih alata je vec konfigurisana kako treba i stoga samo trebamo promeniti izvesne sitnice kako biste jos vise olaksali sebi rad sa ovim alatima.

## OllyDBG v.1.10

Podesicemo sledece opcije u OllyDBG programu. Otvorite Olly i idite u meni za konfiguraciju debuggera **[Hint:** Options -> Debugging options, ili Alt + O **]** Idemo na Exeptions i tu trebamo iskljuciti sve otkacene opcije. Dalje idemo na Strings i tamo treba da otkacimo: Decode Pascal strings i Dump non printable ASCI codes as dots, od mode of string decoding opcija izaberite Plain opciju. U Disasm opciji kao sintaksu izaberite MASM, a u CPU tabu selektujte sledece: underline fixups, show directions of jumps, show jump path, show grayed path if jump is not taken, show jumps to selected command.

Sledece sto mozemo da podesimo je takozvana kolor sema koja nam omogucava da lakse uocimo odredjene komande u dissasemblovanom kodu, kao sto su skokovi i call funkcije. Idemo na Options -> Appearance i tamo u provom tabu **[Hint:** General **]** treba da iskljucimo restore window position and appearance. Dalje u defaults mozete izabrati temu koja vam odgovara. Ja preferiram Black on White kao temu, a Christmas tree kao highlighting sintakse. Vi mozete izabrati bilo koji ali su po mom misljenju ova dva najpreglednija. To je sve sto treba da se konfigurise u Olly.

**Napomena:** OllyDBG je samo jedan od velikog broja debuggera koje mozete naci na internetu. Izmedju ostalih popularnih mozete jos koristiti i SoftIce, TRW2000 ili neke debuggere koji se koriste specijalno za odredjene vrste kompajlera (programske jezike kao sto su Visual Basic, Delphi, .Net itd.), ali samo je Olly besplatan, univerzalan i jednostavno najbolji debugger za pocetnike ali i za iskusne crackere. Preporucujem da ga koristite jer ce se dalje u ovoj knjizi on veoma cesto pominjati i koristiti.

## W32Dism++ / W32Dasm 8.93

Podesicemo sledecu opciju u W32Dasmu da bismo osigurali pravilan prikaz dissasemblovanog fajla. Kliknucemo na Dissasembler -> Font -> Select font... u listi fontova izabrati Courier New, posle cega cemo izabrati opciju Save Default Font. I pored ovoga W32Dasm ima dosta bugova:

1) Ako kada otvorite fajl, W32Dasm ne radi premestite fajl u c:\
2) Ako W32Dasm ne nadje dialoge u fajlu koji ih ima skinite novu verziju
3) Ako W32Dasm ne prikaze disasemblovan kod ovo znaci da je fajl pakovan se nekim pakerom i da se stoga nemoze dissasemblovati (*ovo nije bug*).

## Numega Smart Check v.6.03

Smart Check je specijalni debugger koji se koristi za trazenje gresaka u programima koji su pisani u Visual Basicu (verzije 5 i 6). Jako je koristan a u vecini slucajeva je pregledniji i laksi za obradu podataka od Olly-a. Stoga ako je program koji pokusavate da "razbijete" pisan u Visual Basicu prvo probajte sa Smart Checkom pa tek onda sa ostalim alatima.

Idemo na *Program -> Settings -> Error detection*
Otkacite sve checkboxove osim *Report Errors immediately*.

Idemo na *Advanced*:
Otkacite samo sledece:

- *Report errors caused by other errors*
- *Report errors even if no source code available*
- *Report each error only once*
- *Check all heap blocks on each memory function call*

Sve ostalo NE treba da bude chekirano, stisnemo [ OK ] pa idemo dalje...

Idemo na *Reporting*:
Otkacimo sve osim - *Report MouseMove events from OCX*, [ OK ], zatvorimo ovaj meni za konfiguraciju.

Idemo na *Program -> Event Reporting*

Pritisnite zelenu strelicu u toolbaru da pokrenete selektovani program. Ovo moze biti bilo koji program pisan u Visual Basicu i onda samo treba da otkacimo sledece menije:

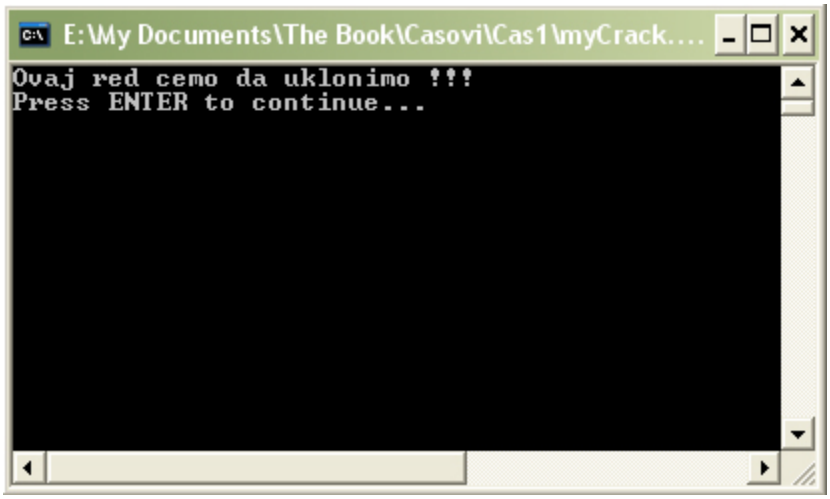
*View -> Arguments ...*
*View -> Sequence Numbers ...*
*View -> Suppresed Errors ...*
*View -> Show errors and specific…*
*Window -> [2] ... - Program Results*

## PeID v.0.93

PeID je program koji cemo dosta cesto koristiti kroz ovu knjigu a kao sto mu ime fajla kaze njegova glavna upotreba je da nam pokaze informacije o "meti" koju pokusavamo da reversujemo. Ti podaci ukljucuju verziju kompajlera kao i da li je program pakovan nekim pakerom i kojom verzijom ako jeste. Ovaj program je po defaultu podesen kako treba ali cemo samo par stvari podesiti kako bismo sebi olaksali zivot. Pokrenite program i idite na opcije. Tamo treba da selektujete *Hard Core Scan* i *Register Shell Extensions.* Kliknite na Save i program je uspesno konfigurisan.

# My first Crack

Posto smo konfigurisali alate koji su nam potrebni sada cemo krenuti sa njihovim koriscenjem. Startujte myCrack.exe koji se nalazi u folderu ...\Casovi\Cas01. Ono sto vidimo je sledeca poruka na ekranu:




Kao sto se vidi na slici ovo ce biti jednostavan primer u kome cemo samo ukloniti prvi red, tako da ce program na ekran izbacivati samo poruku koja se trenutno nalazi u drugom redu.

Ovaj zadatak mozda izgleda komplikovano pocetnicima ali uz pomoc prvih par alata necemo imati nikakvih problema da uspesno i brzo resimo ovaj problem za cije ce nam uspesno resavanje trebati samo dva alata: W32Dasm i Hiew. Ali pre nego sto pocnemo sa crackovanjem moram da vam objasnim same osnove crackovanja.

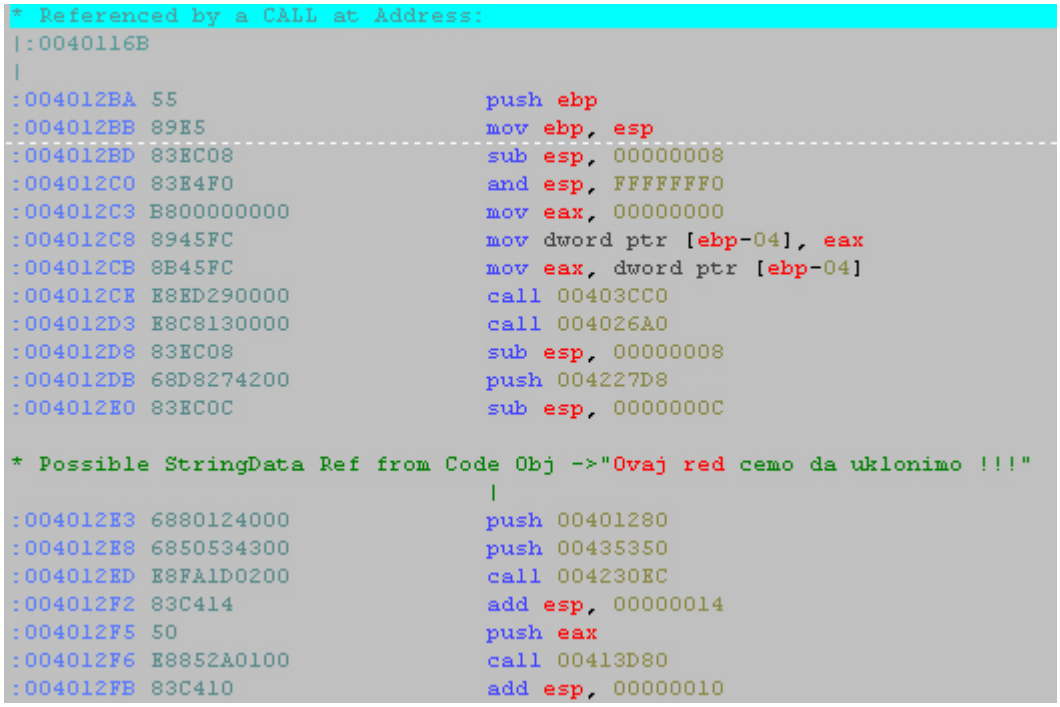
Crackovanje je tehnika menjanja vec kompajlovanih (*gotovih*) exe (*ali i drugih tipova*) fajlova. Modifikacija fajlova se radi na asemblerskom nivou, sto znaci da je pozeljno poznavati osnovne principe funkcionisanja asemblerskih komandi. Naravno ovo i nije obavezno jer mozemo da znamo sta radi neka asemblerska komanda pomocu logickog razumevanja same komande. Ovo na najjednostavnijem primeru znaci da logikom mozemo doci do zakljucka sta radi ova komanda: MOV EAX,1... Naravno pogadjate da je ova komanda ekvivalentna matematickoj funckiji EAX = 1. Osnovna komanda koja ce nam trebati za ovaj prvi cas je NOP. Ona je osnovna crackerska komanda i znaci da se na redu u kome se ona nalazi ne desava bas nista. To znaci da ce program proci preko nje i nece uraditi bas nista. Zasto nam je ovo bitno??? Prilikom crackovanja postoje komande koje zelimo da izmenimo ili da obrisemo. Posto u crackovanju nije pozeljno brisati deo fajla jer bi tako nesto ispod reda koji brisemo moglo da se poremeti, zato koristimo ovu NOP komandu da izbrisemo mesta koja nam ne trebaju. Naravno ova mesta nece biti fizicki izbrisana nego program jednostavno nece raditi nista na mestu na kojem se nalazila komanda koju smo obrisali i nastavice dalje. Prikazano asemblerski to izgleda ovako. Ako imamo sledece asemblerske komande:

```
MOV EAX,1
INC EAX
ADD EAX,5
```

mozemo da pretpostavimo sta se ovde desava. Matematicki to izgleda ovako:

**EAX = 1**
**EAX = EAX + 1**
**EAX = EAX + 5**
Mislim da je svima jasno koji je rezultat EAXa posle izvrsavanja ove tri komande. Recimo da nam taj rezultat ne odgovara i da bismo hteli da EAX na kraju izvrsavanja ove tri komande bude manji za jedan nego sto je sada. Jednostavno cemo spreciti da se EAXu doda 1 u drugom redu i resicemo problem. Sada bi to izgledalo ovako:
**MOV EAX,1**
**NOP**
**ADD EAX,5**
Vidmo da smo jednostavno NOPovali drugi red i da se on sada ne izvrsava kao INC EAX nego kao NOP, zbog cega se rezultat EAXa ne menja posle izvrsavanja tog reda.

Prvo otvorimo ovu "metu" u W32Dasmu i u njemu potrazimo tekst koji zelimo da uklonimo. Kada se program ucita u W32Dasm onda preko Find opcije potrazimo tekst poruke koju trebamo da uklonimo. Idemo na *Search -> Find -> Ovaj red -> OK.* Videcemo ovo:




Kao sto vidimo prvi red ispod nadjenog teksta je zaduzen za prikazivanje poruke na ekran. Ono sto zakljucujemo je da ako samo NOPujemo taj red onda cemo ukloniti poruku sa ekrana. POGRESNO !!! Ovde se namece jedan problem, ako NOPujemo samo taj red onda cemo dobiti gresku u programu i program ce se srusiti jer ga nismo pravilno modifikovali. Problem lezi u cinjenici da je za prikazivanje teksta na ekranu zaduzeno vise redova. Shvatite ovo kao funkciju koja ima vise parametara. Na primer:

**prikazi_tekst_na_ekran(parametar1, parametar2, parametar3,...)**

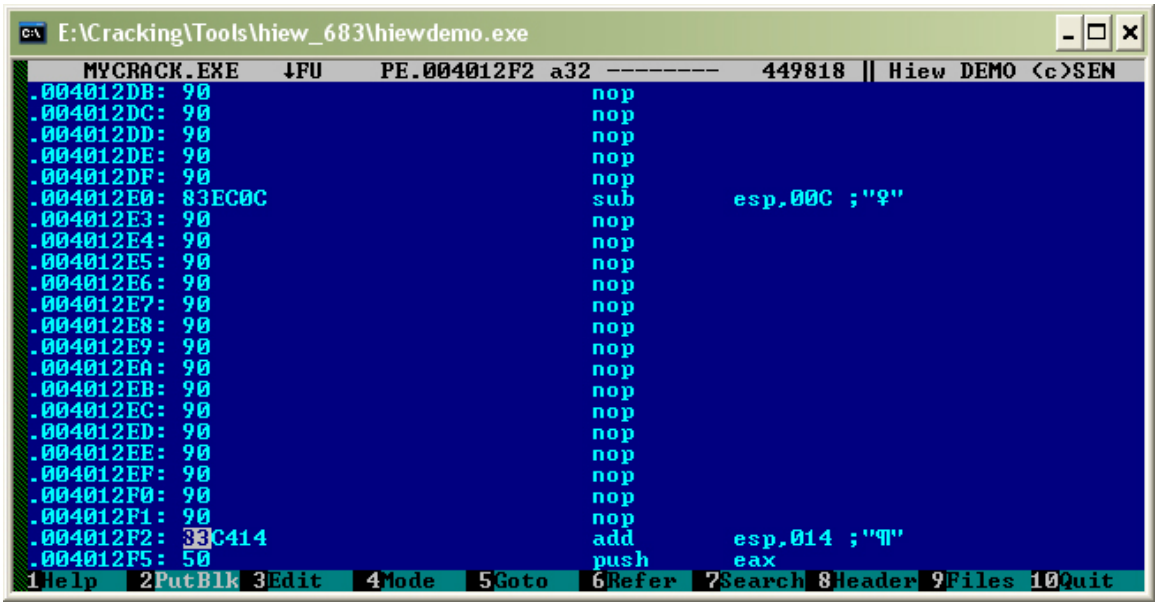
ova funkcija za prikazivanje teksta na ekranu ima tri parametra koja su joj potrebna da prikazu tekst na ekranu. Ono sto smo mi hteli da uradimo je da joj oduzmemo jedan parametar zbog cega bi se program srusio. Problem lezi u tome sto svaka funkcija mora imati dovoljan broj parametara, ovaj broj ne sme biti ni veci ni manji od potrebnog za tu funkciju. Ono sto je zgodno kod asemblera je da mi ne moramo znati koliko parametara kojoj funkciji treba. Primeticete par PUSH komandi koje prethode jednoj CALL komandi. Ove komande predstavljaju ono sto sam objasnjavao na primeru iznad, pri cemu PUSH komande predstavljaju parametre funkcije a sam CALL predstavlja funkciju. Ono sto je takodje bitno da zapamtite je da se u asembleru parametri funkcijama prosledjuju u obrnutom redosledu. Prvo se prosledjuje poslednji parametar, a pre samog pozivanja funkcije prosledjuje se i prvi parametar. Ovo znaci da ne smemo samo da NOPujemo CALL, ili samo da NOPujemo PUSH funkcije, mi da bismo sklonili ispisivanje poruke sa ekrana moramo da NOPujemo sve PUSHeve i na kraju sam CALL. Primecujete da se tri reda ispod poruke koju zelimo da uklonimo sa ekrana nalazi CALL koji sluzi za prikazivanje poruke na ekran. Pre njega se nalaze ne dva nego tri PUSHa, dva ispod teksta poruke i jedan iznad. Ne dajte da vas zbuni to sto se izmedju PUSH komandi nalaze neke druge ASM komande, one nas ne interesuju, interesuju nas samo PUSHevi i CALL. Pre nego sto pocnemo sa samim crackovanjem detaljnije cu objasniti prozor koji trenutno vidite u W32Dasmu. Ono sto vidimo je ovo:

```
* Referenced by a CALL at Address:
|:0040116B
|
:004012BA 55                    push ebp
:004012BB 89E5                  mov ebp, esp
:004012BD 83EC08                sub esp, 00000008
:004012C0 83E4F0                and esp, FFFFFFF0
:004012C3 B800000000            mov eax, 00000000
:004012C8 8945FC                mov dword ptr [ebp-04], eax
:004012CB 8B45FC                mov eax, dword ptr [ebp-04]
:004012CE E8ED290000            call 00403CC0
:004012D3 E8C8130000            call 004026A0
:004012D8 83EC08                sub esp, 00000008
:004012DB 68D8274200            push 004227D8
:004012E0 83EC0C                sub esp, 0000000C

* Possible StringData Ref from Code Obj ->"Ovaj red cemo da uklonimo !!!"
                                 |
:004012E3 6880124000            push 00401280
:004012E8 6850534300            push 00435350
:004012ED E8FA1D0200            call 004230EC
:004012F2 83C414                add esp, 00000014
:004012F5 50                    push eax
:004012F6 E8852A0100            call 00413D80
:004012FB 83C410                add esp, 00000010
```

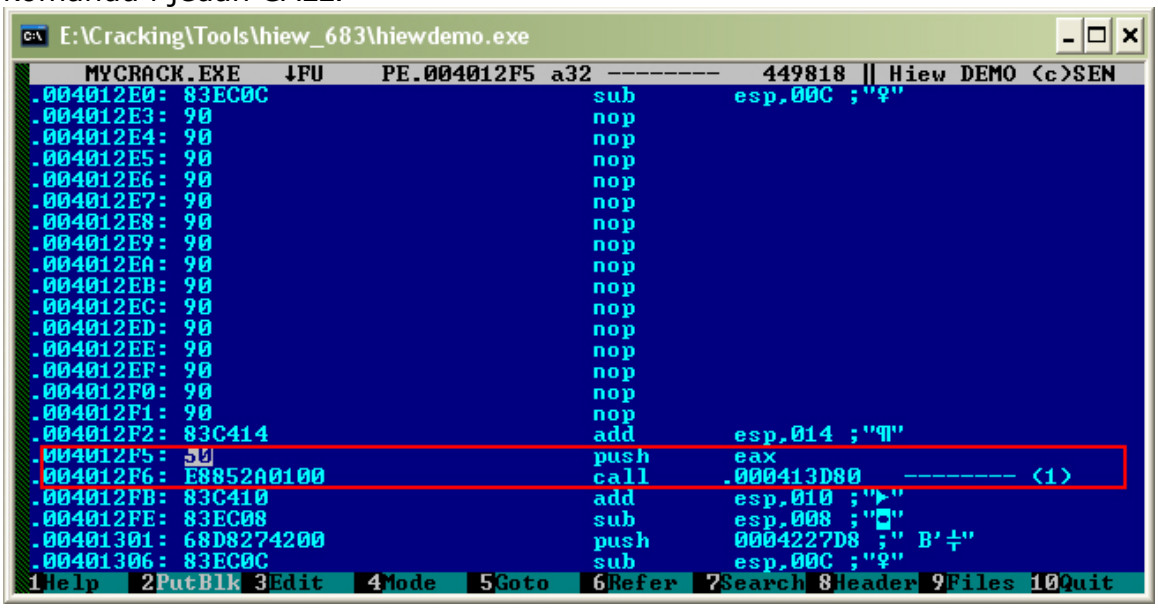
Crveno uokvireni deo koda oznacava virualne adrese na kojima se nalazi ta linija koda. Posmarajte ove brojeve kao da su brojevi 1,2,3 i da predstavljaju red izvrsavanja programa. U ovom primeru prva adresa od koje krece izvrsavanje tzv. OEP (original entry point) je 0040100, a sve ostale adrese se

povecavaju za odredjeni broj. Ovaj broj ne mora biti uvek jedan, veoma cesto i nije jedan. Taj broj direktno zavisi od sadrzaja koji se nalazi uokviren plavo na slici. Ovi hex brojevi predstavljaju komande koje su u fajlu zapisane u HEX obliku. Tako je HEXa decimalno 55 ekvivalentno ASM komandi PUSH EBP. Primeticete da se ovi HEX brojevi pisu iskljucivo u paru po dva. Samo dvocifreni HEX brojevi predstavljaju komande koje su ekvivalentne onima koje su na slici uokvirene zelenom bojom. Nama je za ovaj primer bitno da se ASM komanda NOP u HEX obliku pise kao 90. Ovo znaci da je jedno 90 ekvivalentno jednoj komandi NOP. Da bismo NOPovali jedan red moramo da zamenimo sve dvocifrene HEX brojeve iz tog reda sa 90. Na primer ako brisemo red koji se nalazi na adresi 004012E0 moracemo da zamenimo sadrzaj ove adrese (83EC0C) sa tri NOP komande (909090) jer postoje tri para dvocifrenih brojeva u toj liniji. Primetite kako se to adrese povecavaju, videcete da se recimo iza adrese 004012C0 nalazi 004012C3. Logicno je za ocekivati da se iza 004012C0 nalazi 004012C1, ali ovde se nalazi 004012C3. Ovo se desava zato sto se u ASMu svakom dvocifrenom broju dodeljuje samo jedna adresa. Dakle ako je adresa 004012C0 onda se na toj adresi nalazi samo jedan dvocifreni broj i to 83, na adresi 004012C1 se nalazi E4, a na adresi 004012C2 F0. Jedini razlog zasto su ove tri adrese spojene u jedan red je zato sto te tri adrese predstavljaju samo jednu ASM komandu, pa zbog toga je prva sledeca adresa, na kojoj se nalazi sledeca ASM komanda, 004012C3. Adrese redova odgovaraju adresi prvog dvocifrenog hex broja (bajta) koji se nalazi u toj "liniji" koda.

Posto smo naucili kako da patchujemo (*modifikujemo*) program predjimo na samu modifikaciju. Program mozete editovati u Hiewu ali i u bilo kom drugom hex editoru. Posto je Hiew najbolji za pocetnike sva objasnjenja u prvom poglavlju bice vezana za njega. Dakle startujete Hiew u otvorite myCrack.exe pomocu njega. Prvobitan prikaz u Hiewu nam je nerazumljiv pa cemo ovaj prikaz pretvoriti u prikaz koji je istovetan prikazu iz W32Dasma. Ovo radimo pritiskom na F4 i selekcijom Decode moda. Sada vec vidimo ASM komande. Ako niste zapamtili pogledajte gornju sliku ponovo i prepisite adrese koje treba da NOPujemo, te adrese su: 004012DB, 004012E3 i 004012E8 i 004012ED. Ako zelimo da odemo na te adrese u Hiewu treba da pritisnemo dugme F5 i ukucamo prvo tacku pa adresu na koju zelimo da odemo. Kada odemo na prvu adresu videcemo istu onu PUSH komandu iz W32Dasma. Posto smo naucili da treba da NOPujemo sve dvocifrene brojeve iz tog reda, zapamticemo dva zadnja bajta. Dakle zadnja dva bajta koja cemo NOPovati su 42 i 00. Sada pritisnite F3 da udjete u Edit mod i kucajte nove bajtove kojima zelite da zamenite stare, ti bajtovi su 90. Moracemo da unesemo 5x 90 da bismo NOPovali ceo red. Postavite kursor na sledecu adresu koju treba da NOPujete, to jest na sledecu PUSH komandu i sa njom uradite isto. Isti postupak ponovite i sa zadnjom PUSH komandom i sa CALLom. Kada zavrsite rezultat bi trebalo da izgleda kao na sledecoj slici:




Dakle sve adrese sa PUSH i CALL komandama su NOPovane. Da bismo snimili promene pritisnimo F9, a za izlazak iz programa pritisnite F10. Probajte slobodno da startujete program koji ste malopre patchovali i videcete da on nece raditi!!! Iako smo sve dobro uradili, patchovali smo sve PUSH i CALL komandu program pravi gresku. Ono sto nismo ocekivali, a trebalo je, da je ovo DOS program i da posle ispisivanja poruke na ekran program prebacuje kursor u novi red. Ovo se desava odmah ispod ispisivanja poruke koju smo uklonili, sto znaci da sledeci CALL sluzi bas za ovo. Otvorimo ponovo HIEW i ukucajmo adresu 004012F2. Odmah ispod ove adrese videcemo jednu PUSH komandu i jedan CALL.




Ocigledno je da i ovu CALL i PUSH komandu treba NOPovati jer program ne moze da prebaci kursor u sledeci red bez ispisivanja poruke na ekran. Znaci NOPovacemo i adrese 004012F5 i 004012F6, snimicemo fajl i pokusacemo ponovo da startujemo program. Videcemo da smo sada uspeli i da se poruka vise ne pojavljuje. Uspeli smo da crackujemo nas prvi program.

# My second Crack

Posto smo konfigurisali alate koji su nam potrebni sada cemo krenuti sa njihovim koriscenjem. Startujte myFirstCrack.exe koji se nalazi u folderu …\Casovi\Cas01. Ono sto cete videti je obican Dos prozor koji kaze da nije crackovan. Ovo cemo da promenimo. Otvorimo ovaj program u W32Dasmu **[Hint:** *Disassembler -> Open File to disassemble* **]**, sacekajmo koji trenutak da bi se pojavio disassemblovani program. Posto je ovo drugi cas necu mnogo analizirati nego cemo uraditi samo ono sto je potrebno da bismo crackovali ovaj program. Podsetimo se one poruke koju nam je program izbacio, sada bismo trebali da saznamo odakle se ona poziva. Ovo mozemo uraditi na dva nacina. Prvi je preko find opcije u programu, a drugi, nama malo korisniji, je preko opcije String Reference u W32dasmu. Znaci pritisnimo predzadnje dugme u toolbaru na kojem pise Str Ref i novi prozor ce se otvoriti. U tom prozoru cemo pronaci poruku koju je program izbacio (*Nisam crackovan :P*) i kliknucemo 2x na nju, sto ce nas odvesti do tacnog mesta u kodu ovog exe fajla odakle se ova poruka poziva. Program ce nas odvesti ovde:

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:0040128C(C)
|
:004012EF 837DFC00              cmp dword ptr [ebp-04], 00000000
:004012F3 754C                  jne 00401341
:004012F5 83EC08                sub esp, 00000008
:004012F8 6848284200            push 00422848
:004012FD 83EC0C                sub esp, 0000000C

* Possible StringData Ref from Code Obj ->"Nisam crackovan :P"
|
:00401300 6880124000            push 00401280 <- Ovde smo
:00401305 6850534300            push 00435350
:0040130A E84D1E0200            call 0042315C

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004012A8(C)
|
:0040130F 83C414                add esp, 00000014
:00401312 50                    push eax
:00401313 E8D82A0100            call 00413DF0
:00401318 83C410                add esp, 00000010
:0040131B 83EC08                sub esp, 00000008
:0040131E 6848284200            push 00422848
:00401323 83EC0C                sub esp, 0000000C
```

Vidimo poruku “Nisam crackovan :P”. Znaci odavde se poziva ova poruka. Ono sto je predmet ovog casa je da vas naucim kako da se umesto ove

poruke pojavi neka druga poruka koja se vec nalazi u istom exe fajlu. Ako pogledamo malo iznad ove poruke videcemo jedan uslovni skok:

```
:004012F3 754C                  jne 00401341
```

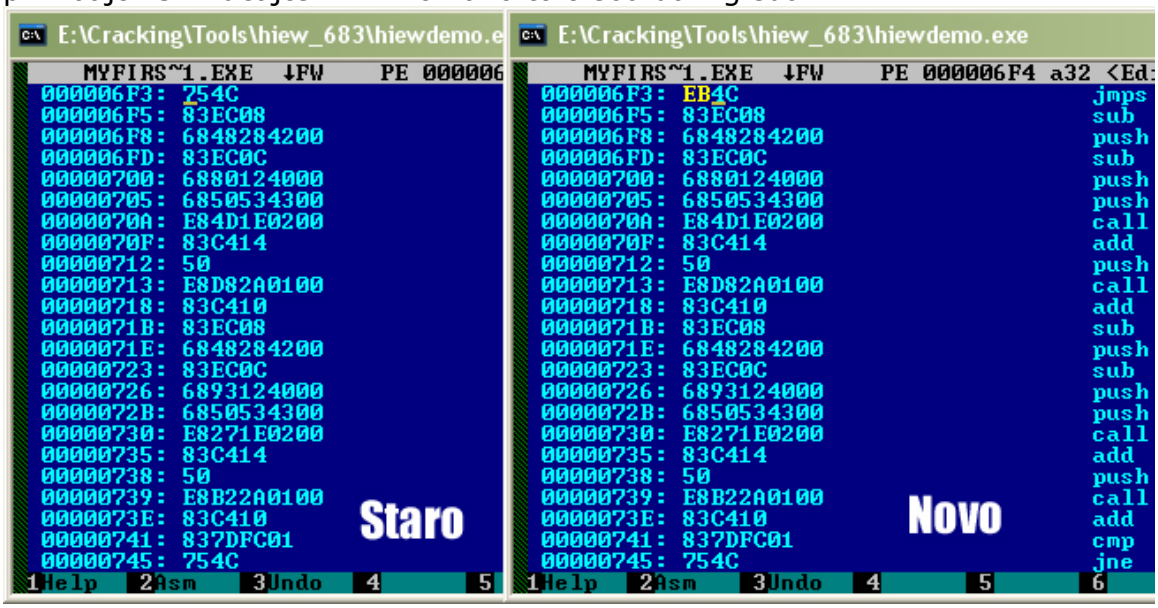
Ovo znaci ako nesto nije jednako program ce skociti na 00401341. Ako odemo malo dole da vidimo sta se nalazi na toj adresi videcemo sledece:

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004012F3(C)
|
:00401341 837DFC01              cmp dword ptr [ebp-04], 00000001
:00401345 754C                  jne 00401393 <-Vazan skok koji kad se izvrsi
:00401347 83EC08                sub esp, 00000008 ; poruka se preskace.

:0040134A 6848284200            push 00422848
:0040134F 83EC0C                sub esp, 0000000C

* Possible StringData Ref from Code Obj ->"Uspesno si me crackovao :)"
|
:00401352 68AE124000            push 004012AE
```

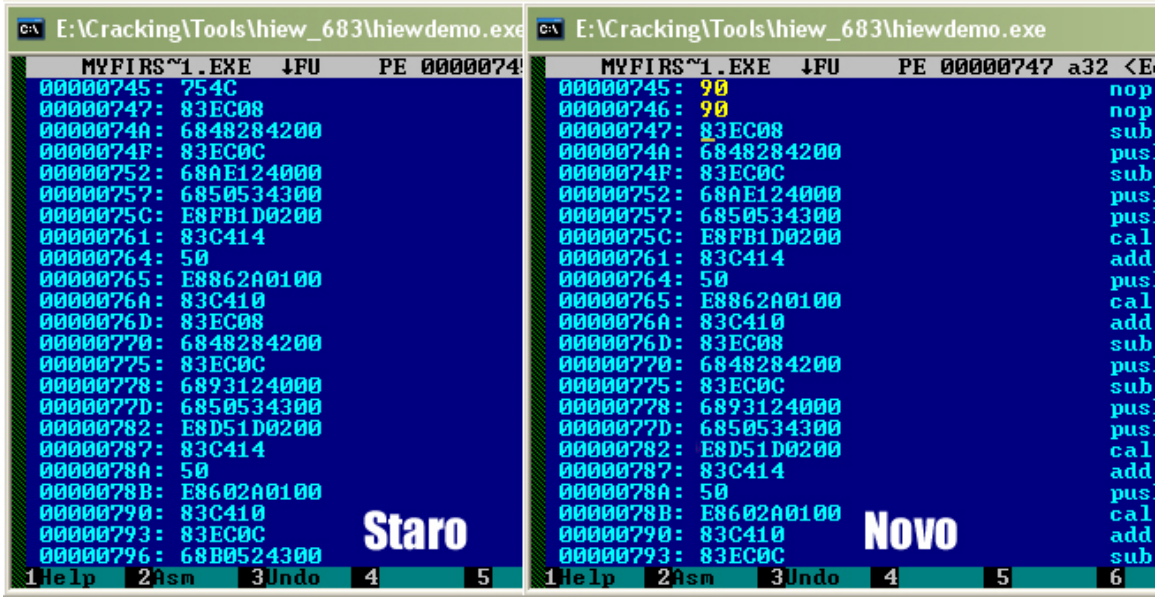
Primeticemo da se tu nalazi poruka “Uspesno si me crackovao :)” koja ce se prikazati na ekranu samo ako se skok sa adrese 004012F3 uvek izvrsava. Ono sto moramo da primetimo je da se izmedju adrese 00401341 i adrese 00401352 nalazi jos jedan skok koji treba da izmenimo. Da bismo uspesno crackovali ovaj program treba da izmenimo onaj skok na adresi 004012F3 sa JNE (*75 hex*) u JMP (*EB hex*) da bi se uvek izvrsavao to jest da bi progam uvek stizao do adrese 00401341, a skok na adresi 00401345 da promenimo tako da se nikad ne izvrsava, to jest da ga promenimo u dva NOPa (*No Operation*). Kada ovo zavrsimo uvek ce se na ekranu prikazivati poruka o uspesnom crackovanju. Ako vam ovaj deo nije jasan obratite paznju na hex adrese na koje vode skokovi. Videcete da nas oni vode ispod i preko poruka koje zelimo da se prikazu na ekranu. Mi moramo da izmenimo ove skokove tako da se uvek na ekranu prikazuje poruka koju mi zelimo. Ovo je mozda najbitniji deo knjige, sama osnova. Da biste uspesno napredovali dalje morate da shvatite kako menjanje odredjenih komandi utice na ponasanje programa. Stoga predlazem da ako se prvi put srecete sa crackovanjem uradite ovu prvu oblast temeljno i da ne prelazite na druge oblasti bez prethodnog razumevanja ove i bez samostalnog resavanja prve vezbe koja se nalazi na kraju ovog poglavlja. Radjenje vezbi je preporucljivo jer ce vam pomoci da se osamostalite i da sami resavate probleme, koje niste pre toga videli i obradili.

Da bismo izvrsili ove promene u programu treba da ga iskopiramo u direktorijum gde se nalazi Hiew i otvorimo ga pomocu njega.

Posto nam ovaj prvobitan prikaz nije bas razumljiv, pritiskom na F4 2x prelazimo u Decode oblik koji je isti kao onaj koji smo videli u W32Dasmu. Mozemo da skrolujemo dole do adrese 004012F3 koju hocemo da izmenimo a mozemo i da idemo do te adrese sa komandom GoTo. Pritisnite F5 i prvo unesite *tacku*, pa adresu 004012F3 i onda enter. Sada kada smo na toj adresi, mozemo je menjati prelaskom u edit mode sa F3. Stavite kursor na prvi bajt 75 i kucajte EB. Evo kako to treba da izgleda:




Sa F9 snimamo promene. Sada treba da promenimo i drugi skok. Pritisnite F5 i unesite tacku, pa adresu 00401345 i onda pritisnite enter. Sada kada smo na toj adresi, mozemo je menjati prelaskom u edit mode sa F3. Stavite kursor na prvi bajt 75 i kucajte 9090. Evo kako to treba da izgleda:




Sa F9 snimamo promene a sa F10 izlazimo iz Hiew-a. Sada mozete startovati ovaj exe fajl i videcete da ce on uvek prikazivati istu poruku, "Uspesno si me crackovao :)"

Mozete pogledati fajl …\Casovi\Cas1\main.cpp da vidite kako taj program izgleda u programskom jeziku C++. Kao sto vidite:

```
#include <iostream>

using namespace std;

int main (int argc, char *argv[])
{
  int i;
  i = 0;
 if (i == 0){
  cout << "Nisam crackovan :P" << endl;
  cout << "Press ENTER to continue..." << endl;
 }
 if (i == 1){
  cout << "Uspesno si me crackovao :)" << endl;
  cout << "Press ENTER to continue..." << endl;
 }
  cin.get();
  return 0;
}
```

postoje dva uslova. Ako je I jednako nula onda se pokazuje poruka da program nije crackovan, a ako je I jednako jedan onda se prikazuje poruka da je program crackovan. Posto je I uvek jednako nula ono sto smo mi uradili je kao da smo zamenili uslove, to jest kao da je za prvu poruku potrebno da je I jednako jedan a za drugu da je I jednako nula. Ovo je malo komplikovaniji primer jer je potrebno izmeniti dva skoka ali kada savladate ovo moci cete da crackujete veliki broj programa za pocetnike jer se skoro svi zasnivaju na ovom ili slicnom principu.

**Vezba:**

Ako zelite mozete proveriti do sada steceno znanje na istom primeru ili na slicnom vec pripremljenom fajlu. …\Casovi\Cas1\myFirstTest.exe je fajl koji treba da crackujete sami. Sam postupak se ne razlikuje mnogo od prethodnog primera. U ovom testu treba ispraviti samo jedan skok. Za one koji znaju ili pomalo razumeju C++ tu je i …\Casovi\Cas1\test.cpp da pogledaju razlike izmedju prvog primera i ovog testa, a fajl …\Casovi\Cas1\myFirstTest.cracked.exe je primer kako treba da izgleda konacan crackovan program. Radjenje vezbi kroz ovu knjigu je preporucljivo kako biste lakse zapamtili postupke crackovanja na slicnim primerima.

**Resenje:**

Treba promeniti skok na adresi 00401391 sa JNE (*755E*) u JMP (*EB5E*).

# OllyDbg from beginning

Pre nego sto pocnemo sa ozbiljnijim reverserskim problemima naucicete kako rade debuggeri i sto je najvaznije kako se koristi Olly.

Kao sto je vec receno na pocetku knjige debuggeri su alati koji su dizajnirani tako da nam omoguce nadgledanje izvrsenja svake ASM instrukcije bilo kojeg programa. Ovde se jasno vidi prednost koju debugeri imaju nad disassemblerima koji nam omogucuju jednostavan pregled "mrtvog koda". Naravno mrtav kod je samo zargonski naziv za staticni ASM kod ciji detaljan pregled mozemo videti ali ne mozemo da vidimo kako se kod ponasa prilikom njegovog izvrsavanja, to jest ne mozemo znati kad se i zasto izvrsavaju skokovi, koje parametre uzimaju CALLovi… Iz ovog razloga je za posmatranje rutina najbolje koriscenje debuggera kao sto je Olly.

# Debugging basics - BreakPoints

Vec je receno da debuggeri imaju moc da zaustave sve posmatrane programe i da njihove instrukcije izvrsavaju redom, jednu po jednu. Ali kako se u stvari pauzira program?

Da bi se ovo desilo, odnosno da bi debuggovani program zastao bas na jednoj specificnoj adresi, potrebno je da jedan od sledeca dva uslova bude ispunjen: 1) Da je postavljen obican software breakpoint na toj adresi ili 2) Da je postavljen hardware breakpoint na toj adresi. Ali sta je to u stvari break-point?

Logicno bi bilo da razmisljate o break-pointu kao komandi koju mi zadajemo nasem debuggeru kako bi on zastao na zeljenoj adresi. Ali software break point nije samo interna komanda u nasem debuggeru, on je u stvari fizicko menjanje sadrzaja memorije na takav nacin da nas debugger moze da detektuje ovakvu promenu i zastane bas kada se ona desi. Naravno posle ovakvog pauziranja programa originalni sadrzaj memorije biva povracen zbog cega smo mi nesvesni modifikacije memorije iako se ona u stvari desava. Posto je svakoj virutualnoj adresi, odnosno komandi, u jednom programu dodeljena tacno jedna adresa moguce je postaviti breakpoint na svaki pojedinacni bajt u memoriji. Ali ovde se namece jedan problem. Naime posto svaki bajt u fajlu moze imati samo vrednosti u opsegu 00h – FFh, odnosno od 0 do 255, kakav to bajt biva zapisan u memoriju tako da nas debugger prepozna break-point. Odgovor na ovo pitanje lezi u arhitekturi procesora koji je dizajniran tako da svaki bajt, sam po sebi ili u skupini bajtova, cini jednu celinu odnosno komandu. Tako ce procesor bajtu 90 dodeliti komandu NOP, bajtu C3 komandu RET itd. Upravo jedan od ovih bajtova ima takvu namenu da hardware-ski pauzira izvrsenje programa i da kontrolu nad procesom preda inicijatoru-vlasniku procesa. Ovaj bajt je oznacen sa CCh i ima asemblersku interpretaciju kao INT3 komanda. Kada ona bude izvrsena *debug exception handler* biva pozvan i sva kontrola nad daljim izvrsenjem programa biva prepustena debuggeru. Naravno sve ovo se odnosi samo na software-break pointove koje postavljamo pomocu naseg debuggera.

Nasuprot njima postoje i takozvani hardware break-pointi koji su posebna mogucnost svakog procesora. Oni se za razliku od softwareskih breakpoint izvrsavaju na direktnom hardwareskom nivou procesora omogucavajuci tako pauziranje bilo kojeg programa direktno, bez ikakve modifikacije memorije aktivnog fajla. Ovo je moguce iz razloga sto se svaki proces izvrsava na nacin na koji procesor tacno zna adresu komande koja se trenutno izvrsava i zato moze, po potrebi, da zaustavi na svakoj komandi. Posto nema modifikacije memorije ovakvi break-pointi ne mogu biti detektovani kao modifikacija memorije nekog programa.

## Debugging basics - User VS kernel mode

Tacniji naziv ovog poglavlja bi bio ring3 vs ring0. Sta je to u stvari user odnosno kernel mode debugging?

Sam Windows®tm je dizajniran na takav nacin da nemaju svi programi pristup svim delovima raspolozive memorije. Postoje dva nivoa pristupa koja programi mogu da imaju, ring3 i ring0. Glavni, odnosno kernelski, nivo pristupa se naziva ring0 i njemu mogu da pristupe samo sistemski programi koji direktno cine operativni sistem. Svi ostali programi imaju samo ogranicen pristup komponentama sistema cineci tako user mode, ili drugacije ring3. Da biste lakse zamislili kakav pristup imaju razliciti fajlovi razmisljajte na sledeci nacin: Program Internet Explorer direktno ima samo ring3 pristup dok se pomocu sistemskih .dll fajlova povezuje sa ring0. Ovi sistemski fajlovi su recimo kernel32.dll i user32.dll koji imaju pristup nativnim windows API pozivima koji su dalje direktno povezani sa procesorskim funkcijama.

Ovakva podela pristupa sistemskoj memoriji je uslovila podelu debuggera na user i kernel mode debuggere. Iako su kernel mode debuggeri mocniji iz razloga sto oni imaju pristup svim delovima sistema u svakom trenutku mi cemo se kroz ovu knjigu zadovoljiti radom sa user mode debuggerima iz razloga sto cemo reversovati aplikacije koje nemaju direktan pristup sistemu.

Da bismo uocili razliku, odnosti prednosti jednog moda u odnosu na drugi razmotrimo slucaj kada moramo da postavimo breakpoint na API koji se koristi za citanje teksta iz nekog prozora. Problem kod postavljanja ovakvog break-pointa je u tome sto vise API funkcija moze da cita podatke iz prozora tako da je potrebno ili izolovati tacan API koji se koristi za citanje podataka iz prozora ili postaviti breakpoint na sve APIje. U ovakvim slucajevima kernel mode debuggeri imaju prednost nad user mode debugerima jer oni mogu da postave breakpoint na low-level kod koji se koristi za citanje podatka iz prozora omogucavajuci nam tako da lociramo tacan API koji pristupa nekom prozoru i cita tekst.

## Introduction to OllyDBG

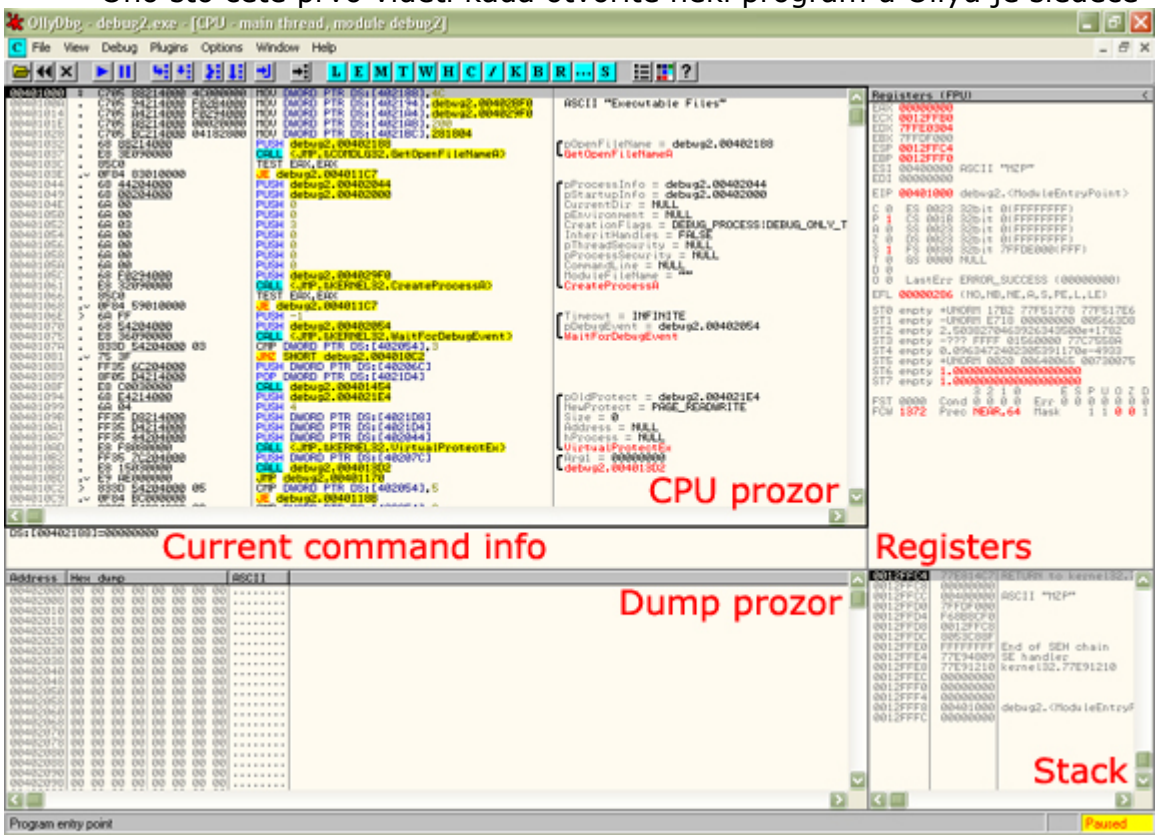
Za reversere OllyDBG, koji je napisao Oleh Yuschuk, predstavlja osnovni i nezaobilazni alat. Iako se moze desiti da je za neki specificni reverserski problem potrebno koriscenje kernel mode debuggera, Olly skoro

uvek predstavlja pravi izbor kada pristupamo nekom problemu. Ovo znaci da iako je Olly ogranicen na ring3 pristup i dalje predstavlja vise nego dovoljno reversersko sredstvo. Ali sta to cini Olly tako dobrim?

Prevashodno Ollyjeva snaga lezi u njegovom izuzetno mocnom disassembleru pomocu kojeg je analiza ASM koda podignuta na najvisi moguci nivo. Ovo znaci da Olly moze da prepozna loopove, switcheve, moze da nam pokaze lokacije na koje vode skokovi, zna sve parametre koje uzima bilo koja standardna Windows®tm API funkcija. Sta vise moze se reci da je Olly jedan od najboljih, ako ne i najbolji debugger ikada napravljen. Iz ovog razloga ce se autor ove knjige prilikom debugginga aplikacija prvenstveno koncentrisati na njegovu upotrebu.

## OllyDBG`s Key features

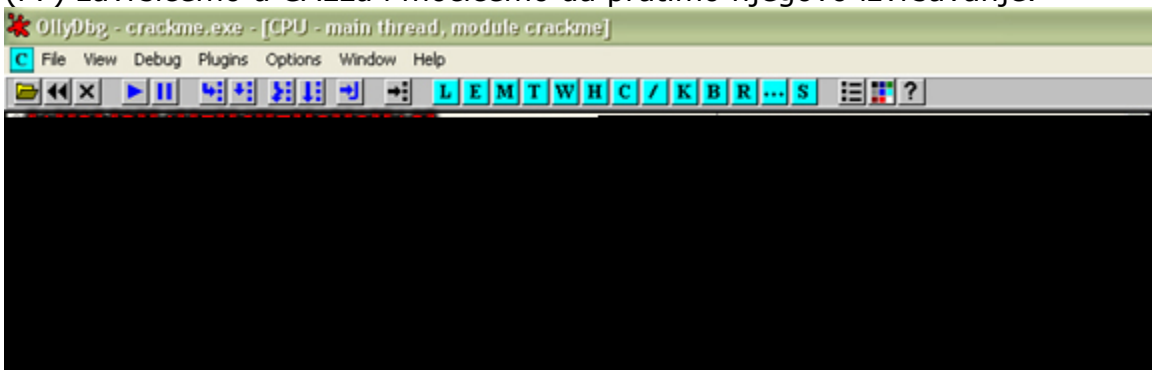
Ono sto cete prvo videti kada otvorite neki program u Ollyu je sledece




Ovaj prozor je glavni Olly prozor koji je nazivan CPU prozor iz razloga sto prikazuje ASM komande koje procesor izvrsava i sto je pomocu njega moguce nadgledanje i izvrsavanje komandu po komandu.
Kao sto se vidi na slici iznad ovaj prozor je podeljen na pet celina koje zajedno cine jednu izuzetno funkcionalnu celinu. Na slici su ove celine vec obelezene i sada cemo preci jednu po jednu celinu.

*CPU prozor* je glavni prozor u Ollyu i njegova namena je pracenje, odnosno postepeno, izvrsenje koda debugovane mete. Ovo se postize pomocu komandi *Step over* i *Step into*. Ove dve komande omogucavaju

izvrsavanje koda po ASM komandama, tako da se priliko izvrsenja step over ili step into komande izvrsi tacno jedna ASM komanda iz CPU prozora. Naravno komande se izvrsavaju na linearan nacin, jedna za drugom, cineci tek zajedno jednu funkcionalnu celinu. Komande step over i step into se funkcionalno razlikuju jedino kod izvrsenja ASM komande CALL. Kada se ova ASM komanda izvrsava step over postavlja nevidljivi breakpoint odmah iza CALLa, odnosno na sledecoj komandi, dok step into postavlja breakpoint na prvoj komandi unutar CALLa. Ovo znaci da ako debugujemo neki program i dodjemo do komande CALL, sledeca ASM komanda na kojoj cemo zavrsiti zavisi od izbora traceovanja kroz kod. Ako izaberemo step over (F8) zavrsicemo na komandi koja se nalazi odmah posle CALLa, sa tim sto ce sadrzaj CALLa biti izvrsen bez naseg nadgledanja, a ako izaberemo step into (F7) zavrsicemo u CALLu i mocicemo da pratimo njegovo izvrsavanje.



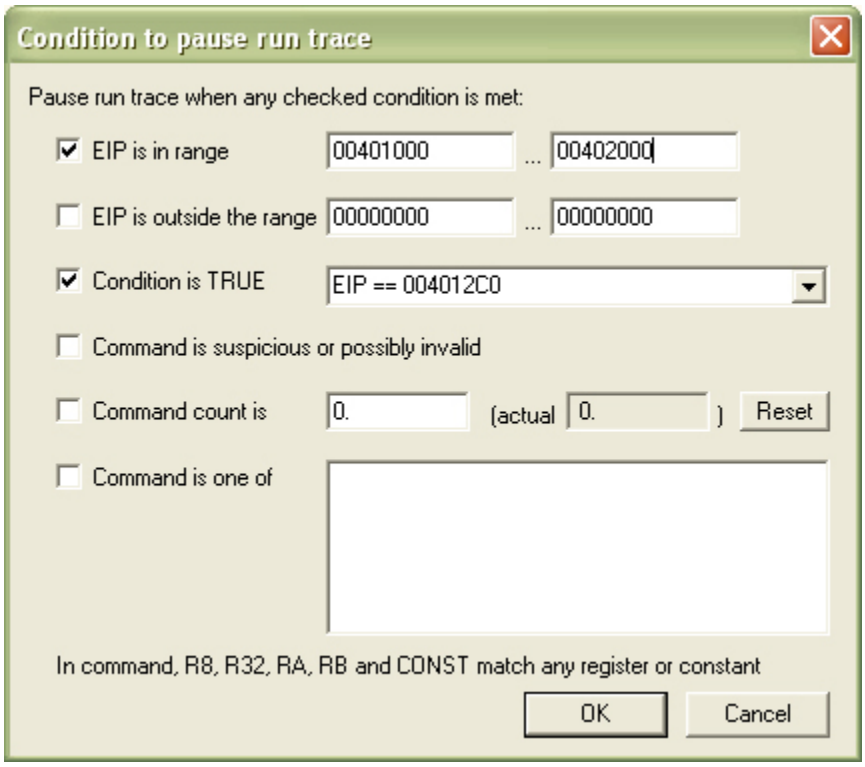

[01] **Reset trenutno otvorene mete**
[02] **Zatvaranje trenutno otvorene mete**
[03] **Run (F9), startovanje mete i njeno izvrsavanje do prvog break-pointa**
[04] **Pause, komanda koja pauzira izvrsenje program**
[05] **Step into (F7)**
[06] **Step over (F8)**
[07] **Trace into**
[08] **Trace over**
[09] **Execute till return (CTRL+F9)**
[10] **Go to adress (CTRL+G)**

Komande step into i step over su na slici iznad obelezene brojevima 5 i 6 i treba ih razlikovati od komandi trace into i trace over posto one imaju slicnu ali ne i istu upotrebu. Pored ovoga ove komande mozemo da koristimo i pomocu precica F7 i F8.

Pored vec opisanih funkcija step over i step into Olly poseduje jos par osnovnih traceing funkcija koje su dizajnirane tako da nam olaksaju analizu izvrsnih fajlova. Jedna ovakva opcija je opcija execute till retun koja nam omogucava da izvrsimo sav kod CALLa u kojem se nalazimo i da se nakon njegovog izvrsavanja nadjemo na ASM komandi koja se nalazi odmah ispod CALLa.

Vec je receno da se trace into i trace over komande razlikuju od njima slicnim komandi step over i step into. Ove komande se koriste za automatizovani traceing kroz kod. Ova vrsta traceinga nam omogucuje brz prelazak preko koda, odnosno negovo izvrsavanje sve dok neki uslov ne bude izvrsen. Da bi smo postavili uslov za traceovanje moramo otici u meni Debug -> Set condition (CTRL+T) gde mozemo podesiti razlicite tipove logickih uslova koji ce predstavljati pauzu u traceingu ako bilo koji od tih uslova bude ispunjen. Glavne tri opcije ovog menija se odnose na EIP i na

custom uslove koje mozemo da postavimo. EIP je registar kojem nemamo direktan pristup, odnosno njemu pristupamo jedino u read modu. Ovo znaci da je nemoguce direktno pomocu ASM komandi pristupiti EIP registru jer se on direktno podesava pri izvrsavanju ASM koda tako da dobija vrednost adrese sledece ASM komande koja ce se izvrsiti. Jedina moguca manipulacija EIP registra je putem STACKa. Bas zbog ovoga mozemo postavljati uslove traceovanje prema EIP registru. Tako mozemo da "kazemo" Ollyju da pauzira izvrsavanje programa na prvoj komandi koja se nalazi u ili van nekog opsega. Naravno opseg mozemo bazirati na EIP registru jer nam on uvek govori koja ce se sledeca komanda izvrsiti, odnosno pokazuje na bajt koji je poslednji izvrsen. Stoga ako postavimo uslove kao na sledecoj slici:




mozemo da biramo mod izvrsenja traceovanja i tako se za duze ili krace vreme nadjemo na adresi 004012C0 ili na nekoj adresi izmedju 00401000 i 00402000. U zavisnosti da li cemo izabrati trace over ili trace into Olly ce pratiti kod koji se izvrsava ali u trace over slucaju nece ulaziti u CALLove. Ovo znaci da ce trace over komanda uvek biti brze izvrsena ali nece biti pouzdana kao trace into komanda. Ovo se narocito primecuje ako je opseg za kojim tragamo jako mali.

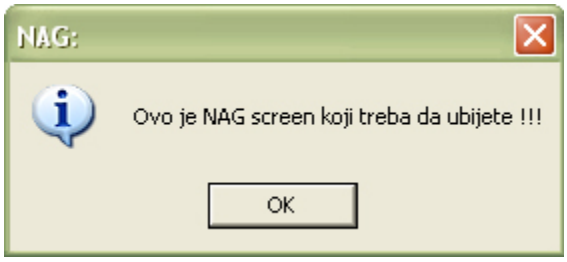
Poslednja komanda koja nam pruza mogucnost manipulacije adresama je opcija Go to adress koju mozemo da koristimo uvek kada znamo na kojoj se adresi nalazi komanda koja nas zanima. Precica koju mozete koristiti da bi ste aktivirali ovu opciju je CTRL+G.

# 02 NAG Screens

U prvom poglavlju smo savladali osnovne tehnike pronalazenja date informacije u exe fajlu i menjanja iste, a u drugom cemo nauciti kako da se izborimo sa osnovnim problemima u reversovanju.

# Killing NAGs – *MsgBoxes*

NAG ekrani su one dosadne poruke koje se pojavljuju pre ulaska u neki program ili na samom izlasku iz njega, a njihova glavna funkcija je da vas podsete da niste platili program koji koristite. Postoji vise vrsta NAGova ali su najcesce zastupljena dva standardna tipa, *message boxovi i dialozi.* U ovom primeru cu vam pokazati kako da se resite obicnog *message box* NAGa. On izgleda upravo ovako:




a nalazi se u fajlu …\Casovi\Cas2\NAG.exe. I za ovaj primer cemo upotrebiti iste alate (*W32Dasm i Hiew*) kao i za prvi primer.

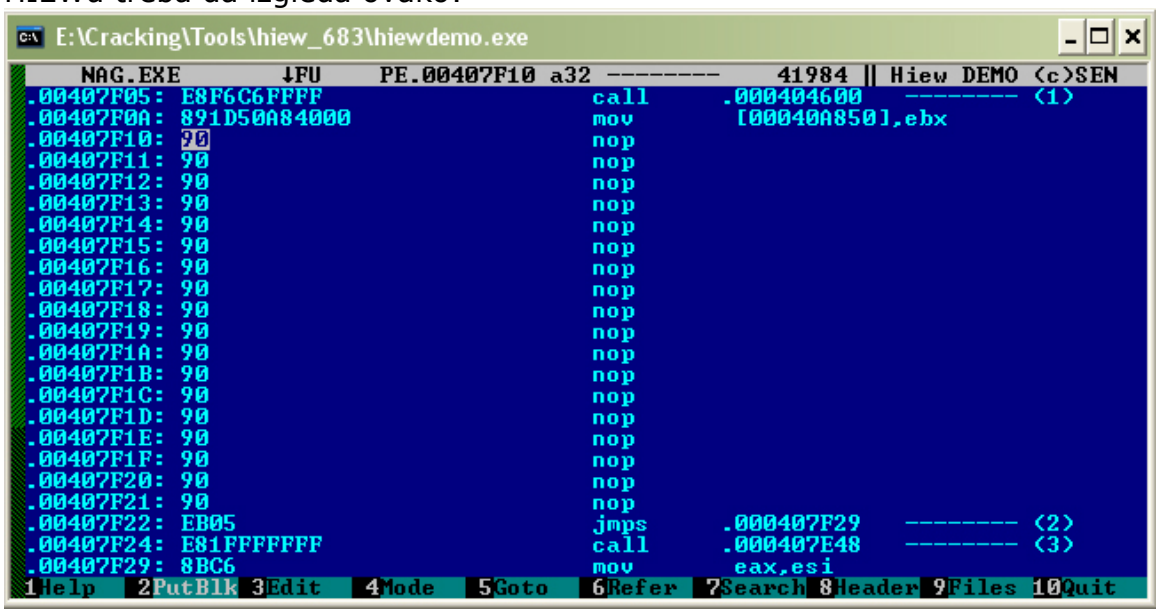
U W32Dasmu otvorite ovaj exe fajl i sacekajte da se disassembluje. Najlaksi nacin za ubijanje ovog NAGa je trazenje teksta koji se u njemu pojavljuje. Otvorite opet String Reference i nadjite tekst iz ovog message boxa. Duplim klikom na taj tekst zavrsicemo ovde:

```
* Reference To: user32.SetWindowTextA, Ord:0000h
|
:00407F05 E8F6C6FFFF            Call 00404600
:00407F0A 891D50A84000          mov dword ptr [0040A850], ebx
:00407F10 6A40                  push 00000040

* Possible StringData Ref from Code Obj ->"NAG:"
|
:00407F12 68407F4000            push 00407F40

* Possible StringData Ref from Code Obj ->"Ovo je NAG screen koji treba da "
                                       ->"ubijete !!!" <- Ovde smo
|
:00407F17 68487F4000            push 00407F48
:00407F1C 53                    push ebx

* Reference To: user32.MessageBoxA, Ord:0000h
|
:00407F1D E8C6C6FFFF            Call 004045E8
:00407F22 EB05                  jmp 00407F29
```

Ono sto je specificno za ove message box NAGove je da se oni iz bilo kog programskog jezika pozivaju na isti nacin:

```
MessageBoxA(handle[hwnd], Text, Naslov, MB_TIPMESSAGEBOXA);
```

sto znaci da se CALL funkciji koja generise ovaj message box prosledjuju cetiri parametra. Ovo prosledjivanje ide u obrnutom redosledu pa se tako pre CALL funkcije na adresi 00404F1D nalaze cetiri PUSH funkcije koje prosledjuju ove parametre u obrnutom redosledu. Ako vam ovo nije jasno predlazem da procitate deo o STACKu sa strane 8. Kao sto pretpostavljate ova funkcija ne treba da se izvrsava nikada stoga je treba NOPovati. Obratite paznju samo na jedno, a to je da ako NOPujete samo CALL onda ce doci do greske u programu. Pravi nacin ubijanja ovakvih NAG ekrana je da trebate NOPovati sve PUSH funkcije koje prethode CALLu a onda i sam CALL. To u HIEWu treba da izgleda ovako:




Sve od adrese 00407F10 pa do adrese 00407F21 treba da bude NOP. Sada mozete startovati NAG.exe i videcete da NAGa vise nema.

**Vezba:**

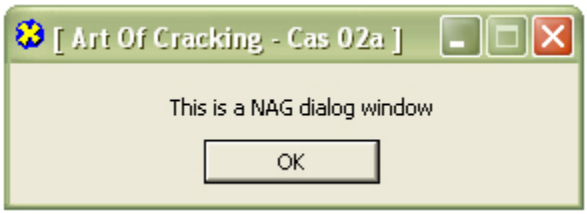
Ako zelite mozete proveriti do sada steceno znanje na istom primeru tako sto cete ubiti tekst koji se pojavljuje kada korisnik klikne na dugme **?.** kada se pojavljuje About dijalog.

**Resenje:**

Sve od adrese 00407EA0 pa do adrese 00407EB3 treba da bude NOPovano.

# Killing NAGs – *Dialogs*

U proslom primeru je objasnjeno kako se skidaju *messagebox* NAGovi a u ovom ce biti objasnjeno kako se skidaju *dialog* NAGovi. Taj dialog moze izgledati bas ovako:




Razlika je mozda neprimetna ali za razliku od *messagebox* NAGova, ovaj NAG je napravljen na isti nacin kao i ostali dijalozi u programu. Zasto nam je ovo bitno? Obicnim korisnicima i nije bitno ali ovo nama govori na koji nacin se generise ovaj NAG i kako mozemo da ga se resimo. Otvorimo program …\Casovi\Cas2\NagWindow.exe u W32Dasmu. Posto je ovo dijalog potrazicemo sve dijaloge u ovom exe fajlu. Pogledajmo malo dole od pocetka disassemblovanog fajla i videcemo sledece:

```
++++++++++++++++++ DIALOG INFORMATION ++++++++++++++++++++
Number of Dialogs =    1 (decimal) <- Zasto ???
Name: DialogID_0064, # of Controls=008, Caption:"½F", ClassName:""
    001 - ControlID:0000, Control Class:"EDIT" Control Text:""
    002 - ControlID:0000, Control Class:"EDIT" Control Text:""
    003 - ControlID:0000, Control Class:"BUTTON" Control Text:"&Check"
    004 - ControlID:0000, Control Class:"BUTTON" Control Text:"&Exit"
    005 - ControlID:0000, Control Class:"BUTTON" Control Text:"&?"
    006 - ControlID:0000, Control Class:"STATIC" Control Text:"Name:"
    007 - ControlID:0000, Control Class:"STATIC" Control Text:"Serial:"
    008 - ControlID:0001, Control Class:"" Control Text:""
```

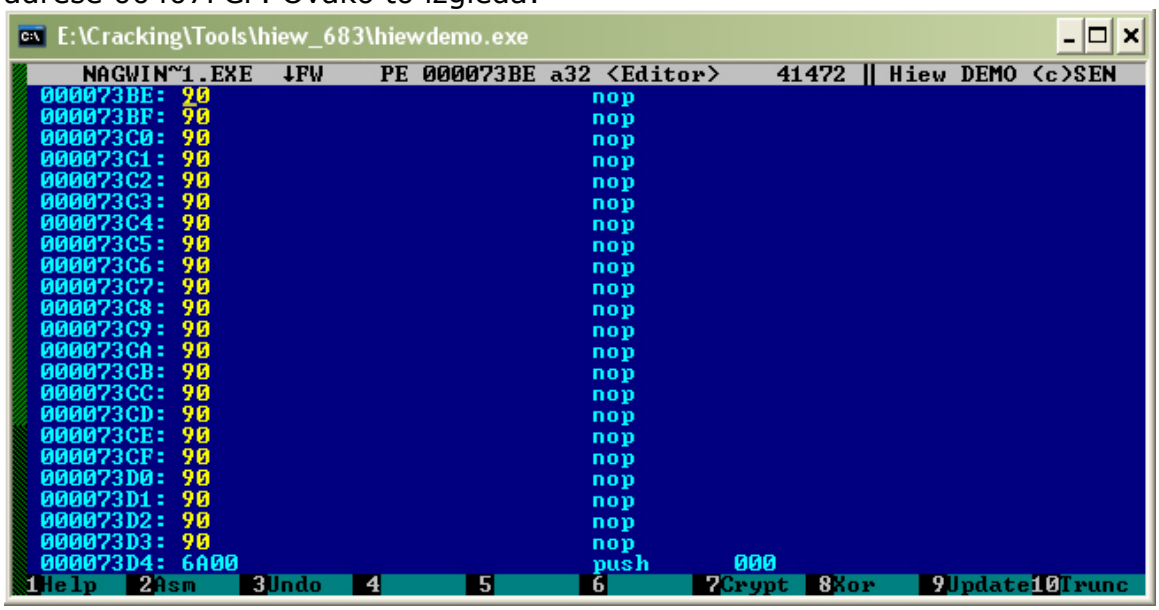
Ovde nesto ne valja! Dijalog koji se ovde nalazi je onaj drugi dijalog koji se pojavljuje nakon sto pritisnemo OK u NAG prozoru. Ali gde je onaj prvi, NAG window? Otkricemo to. Za sada samo zapamtite da je ime ovog dijaloga (*DialogID_0064*) Posto W32Dasm dodaje prefix DialogID_ stvarno ime ovog dijaloga je broj 64h u hex obliku a jednako je 100 u decimalnom. Ovo nam je bitno jer CALLu koji je zaduzen za prikazivanje ovog dialoga je potreban taj ID kako bi znao koji dialog da prikaze. Da bismo saznali odakle se poziva ovaj dijalog idemo gore u meni na dugme DLG Ref (*Dialog References*) i duplim klikom na ime dijaloga zavrsicemo ovde:

```
:00407FCF E8DCC5FFFF          Call 004045B0
:00407FD4 6A00                push 00000000
:00407FD6 68647E4000          push 00407E64
:00407FDB 6A00                push 00000000
:00407FDD 6A64                push 00000064  <- Ovde smo
:00407FDF FF354C984000        push dword ptr [0040984C]
:00407FE5 E8C6C5FFFF          Call 004045B0
:00407FEA E8D5B4FFFF          Call 004034C4
```

Ako prebrojimo parametre koji prethode prvom sledecem CALLu na adresi 00407FE5 videcemo koliko parametara ima funkcija za pozivanje dijaloga. Zakljucili smo da je potrebno 5 parametara za ovu funkciju a da je predzadnji odnosno drugi dialog ID. Posto se NAG pojavljuje pre pojavljivanja ovog dijaloga zakljucujemo da se ista funkcija mora nalaziti i iznad ovoga gde smo sada, samo sa razlikom u id-u dijaloga koji poziva. I bili smo u pravu.

```
:00407FBE 6A00                  push 00000000
:00407FC0 68647E4000            push 00407E64
:00407FC5 6A00                  push 00000000
:00407FC7 6A65                  push 00000065  <- Drugi dijalog ID
:00407FC9 FF354C984000          push dword ptr [0040984C]
* Reference To: user32.DialogBoxParamA, Ord:0000h
|
:00407FCF E8DCC5FFFF            Call 004045B0
```

Isti broj parametara prethodi CALLu na adresi 00407FCF. Znaci sigurno se CALL na adresi 00407FCF koristi za prikazivanje NAGa. Ovi NAGovi se uklanjaju na isti nacin kao i *message box* NAGovi. Znaci sve PUSH komande i CALL na kraju moraju biti NOPovane, a to znaci sve od adrese 00407FBE do adrese 00407FCF. Ovako to izgleda:




Ne dajte da vas brojevi sa strane zbune, 73BE je tacan polozaj virtualne adrese 00407FBE koja odgovara stvarnoj 73BE samo u memoriji. Ovi stvarni polozaji bajtova ce vam se prikazati tek kada pritisnete F3 i udjete u Edit mod.

**NAPOMENA:** U vecini slucajeva se svi dijalozi pojave u W32Dasmu ali sada iako se desila neka greska u W32Dasmu uspeli smo da pronadjemo sve dijaloge i da eliminisemo NAG ekran.

# Killing NAGs – MsgBoxes & Olly

Posto smo vec pokazali skoro sve sto je vazno da se pokaze vezano za W32Dasm, sada cemo nauciti kako se traze NAGovi pomocu Ollya i kako se to sve NAG moze ukloniti. Meta koju cemo reversovati se nalazi u folderu Cas1 a zove se vct_crackme1.exe a na njenom OEPu nalazi se sledeci kod.

```
00401000  . 6A 00          PUSH 0                    ; /hTemplateFile = NULL
00401002  . 6A 03          PUSH 3                    ; |Attributes = READONLY
00401004  . 6A 00          PUSH 0                    ; |Mode = 0
00401006  . 6A 00          PUSH 0                    ; |pSecurity = NULL
00401008  . 6A 00          PUSH 0                    ; |ShareMode = 0
0040100A  . 6A 00          PUSH 0                    ; |Access = 0
0040100C  . 68 9C314000    PUSH 0040319C             ; |FileName = "\\.\SICE"
00401011  . E8 24020000    CALL 0040123A             ; \CreateFileA
00401016  . 83F8 FF        CMP EAX,-1
00401019  . 74 06          JE SHORT dmp.00401021
0040101B  . 50             PUSH EAX                  ; /ExitCode
0040101C  . E8 1F020000    CALL <ExitProcess>        ; \ExitProcess
```

Ovaj kod ne predstavlja nista specijalno, ovo je samo standardan nacin za detekciju SoftICE debuggera pomocu CreateFileA komande. Pored vxd reference ka SotfICEu \\.\SICE takodje se pojavljuje i referenca \\.\NTICE ako se radi o NT sistemima. Da bismo ovu detekciju zaobisli, naravno ako koristite SoftICE, dovoljno je da patchujete JE skok sa adrese 00401019 u JMP. A ako pogledate malo ispod ovih par komandi primeticete da se tu nalazi prvi NAG. Taj deo koda izgleda ovako:

```
00401021  > \6A 00         PUSH 0                    ; /Style = MB_OK
00401023  . 68 20304000    PUSH 00403020             ; |Title = "... nag ..."
00401028  . 68 00304000    PUSH 00403000             ; |Text = "..."
0040102D  . 6A 00          PUSH 0                    ; |hOwner = NULL
0040102F  . E8 E2010000    CALL <MessageBoxA>        ; \MessageBoxA
```

Naravno ubijanje NAG ekrana se radi veoma lako, naime potrebno je samo NOPovati sve PUSH komande koje prethode CALLu koji se koristi za prikazivanje NAGa a na kraju i sam CALL. Operacija NOPovanja se radi duplim klikom na selektovanu komandu i unosenjem reci NOP u novo-otvoreni prozor. Posle klika na OK ili <ENTER> Olly ce selektovanu komandu promeniti u NOP, posle cega se ona nece vise izvrsavati. Dakle posle NOPovanja svih ovih komandi imacemo sledece stanje:

```
00401021    90              NOP
00401022    90              NOP
...
0040102F    90              NOP
00401030    90              NOP
00401031    90              NOP
00401032    90              NOP
00401033    90              NOP
```

Posto postoji jos jedan NAG u programu potrazicemo ga traceovanjem kroz kod, to jest pritiskanjem na F8 sve dok ne dodjemo do sledeceg CALLa:

```
0040104A  . 6A 0A           PUSH 0A
0040104C  . FF35 AC314000   PUSH DWORD PTR DS:[4031AC]
00401052  . 6A 00           PUSH 0
00401054  . FF35 A8314000   PUSH DWORD PTR DS:[4031A8]
0040105A  . E8 0B000000     CALL 0040106A
```

Posle izvrsavanja ovog CALLa glavni prozor mete ce se pojaviti na ekranu. Posto znamo da se NAG izvrsava nakon izlaska iz ove mete postavicemo jedan obican break-point (pritiskom na F2) na sledecu adresu koja se nalazi odmah ispod CALLa koji je zaduzen za prikazivanje ovog prozora, postavicemo break-point na adresu 0040105F.

```
0040105F  |. E8 61010000     CALL 004011C5
00401064  |. 50              PUSH EAX                    ; /ExitCode
00401065  \. E8 D6010000     CALL <ExitProcess>          ; \ExitProcess
```

Naravno posle zatvaranja glavnog prozora nase mete zavrsicemo na nasem break-pointu. Ovog puta cemo pritisnuti F7 da bismo usli u CALL na adresi 0040105F. Zasto? Zato sto se posle ovog CALLa nalazi jos samo kernel32.ExitProcess CALL koji sluzi za gasenje nase mete. Iz ovog razloga zakljucujemo da se drugi NAG nalazi u CALLu sa adrese 0040105F. Kada udjemo u taj CALL videcemo sledece:

```
004011C5  /$ 6A 00           PUSH 0                      ; /Style = MB_OK
004011C7  |. 68 66304000      PUSH 00403066               ; |Title = "..."
004011CC  |. 68 40304000      PUSH 00403040               ; |Text = "... nag ..."
004011D1  |. 6A 00            PUSH 0                      ; |hOwner = NULL
004011D3  |. E8 3E000000      CALL <MessageBoxA>          ; \MessageBoxA
004011D8  \. C3              RET
```

I kao sto vidimo bili smo u pravu! Trazeni NAG se zaista nalazi u ovom CALLu i njega cemo kao i prvi NAG ukloniti na isti nacin, a posle patchovanja nasa meta ce izgledati ovako:

```
004011C5    90               NOP
004011C6    90               NOP
004011C7    90               NOP
004011C8    90               NOP
004011C9    90               NOP
004011CA    90               NOP
...
004011D1    90               NOP
004011D2    90               NOP
004011D3    90               NOP
004011D4    90               NOP
004011D5    90               NOP
004011D6    90               NOP
004011D7    90               NOP
004011D8  \. C3              RET
```

Pored ovog nacina patchovanja NAGova postoji i drugi nacin patchovanja. Ovaj drugi nacin se koristi u slucaju da program broji NOPove koji se nalaze u njegovom kodu. Ovaj patcherski trik se ogleda u patchovanju zeljenih komandi u novi niz komandi koje cine dve ASM komande INC EAX, DEC EAX... Prva komanda povecava vrednost EAXa za jedan a druga smanjuje tu vrednost za jedan. Ovde samo treba paziti da

ako EAX ima uticaja na dalje izvrsavanje komandi bude isti broj DEC EAX i INC EAX komandi. Primenjeno na ovaj primer to izgleda upravo ovako:

```
004011C5    40              INC EAX
004011C6    48              DEC EAX
004011C7    40              INC EAX
004011C8    48              DEC EAX
004011C9    40              INC EAX
004011CA    48              DEC EAX
004011CB    40              INC EAX
004011CC    48              DEC EAX
004011CD    40              INC EAX
004011CE    48              DEC EAX
004011CF    40              INC EAX
004011D0    48              DEC EAX
004011D1    40              INC EAX
004011D2    48              DEC EAX
004011D3    40              INC EAX
004011D4    48              DEC EAX
004011D5    40              INC EAX
004011D6    48              DEC EAX
004011D7    40              INC EAX
004011D8  \. C3             RET
```

Pored ovoga postoji i jos jedan nacin patchovanja koji ce vam omoguciti da uklonite NAG tako da izmenite samo jedan bajt! Ovaj nacin se uvek moze primenjivati a ta promena bi izgledala ovako:

```
004011C5  /$  6A FF         PUSH FF                     ; /Style = MB_OK
004011C7  |.  68 66304000    PUSH 00403066               ; |Title = "..."
004011CC  |.  68 40304000    PUSH 00403040               ; |Text = "... nag ..."
004011D1  |.  6A 00          PUSH 0                      ; |hOwner = NULL
004011D3  |.  E8 3E000000    CALL <MessageBoxA>          ; \MessageBoxA
004011D8  \.  C3             RET
```

ili

```
004011C5  /$  6A 00         PUSH 00                     ; /Style = MB_OK
004011C7  |.  68 66304000    PUSH 00403066               ; |Title = "..."
004011CC  |.  68 40304000    PUSH 00403040               ; |Text = "... nag ..."
004011D1  |.  6A FF          PUSH FF                     ; |hOwner = NULL
004011D3  |.  E8 3E000000    CALL <MessageBoxA>          ; \MessageBoxA
004011D8  \.  C3             RET
```

Kao sto vidite potrebno je samo izmeniti MessageBox tip u neki broj za koji ne postoji pravi tip MessageBoxa ili umesto ovoga mozete programu proslediti HWND koji ne postoji. Ovaj drugi nacin, patchovanje samo jednog bajta, je mnogo isplativiji ako radite inline patching nekog pakera!

Konacno kada smo uradili sve promene jednostavnim klikom u CPU prozoru Ollya na desno dugme -> Copy to executable -> All modifications -> Copy All -> desno dugme -> Save file... snimicemo sve promene direktno pomocu Ollya. Ovo tehnika patchovanja ukida potrebu za nekim Hex Editorom pomocu koga biste direktno menjali fajl!

# Killing NAGs – Dialogs & Olly

Naravno obican MessageBox NAG je veoma lako "ubiti", ali sta ako se umesto poziva ka MessageBoxu kao NAG koristi dijalog? U ovom slucaju ne mozemo traziti karakteristicne stringove koji se pojavljuju u prozoru jer se ovaj tekst na dijalogu nalazi u obliku resursa. Ali saznanje da je dialog resurs koji program poziva nam govori dve stvari: 1) da se moze pronaci u fajlu pomocu resource editora i 2) da je statican, odnosno da jedan resurs moze predstavljati samo jedan prozor (*ili neki drugi tip podataka*).

Nasa meta koja se ponasa bas na gore opisani nacin se nalazi u folderu Cas02 a zove se editor.exe. Ovu metu cemo otvoriti pomocu Ollya i zahvaljujuci njemu cemo ukloniti ovaj NAG.

Vec smo rekli da ce se NAG u ovoj meti prikazati kao poseban prozor, sto znaci da najverovatnije koristi zaseban deo resursa (*.res*) koji se nalaze u ovom .exe fajlu. Zbog ovoga cemo iskorsititi Olly da pogledamo sve resurse koji se nalaze u ovom fajlu klikom na ALT + M da bismo videli koje sve .dll fajlove poziva nas .exe fajl, dalje selekcijom glavnog .exe fajla i konacnim klikom na desno dugme pa na View all resources,posle cega cemo videti ovo:

| Address | Type | Name | Language | Size | Information |
|---|---|---|---|---|---|
| 004120B4 | MENU | MAINMENU | 0000 Language | 0000005A | &File |
| 00412110 | DIALOG | 384 | 0000 Language | 00000278 | NAG-SCREEN |

Videcemo da se u fajlu nalazi tacno jedan dijalog, ciji je ID 384h a ime NAG-SCREEN. Stvari su ovde veoma ocigledne, ali sada se postavlja pitanje kako naci mesto sa koga se poziva ovaj dijalog? Ako se secate prethodnog primera sa W32Dasmom i dijalozima znacete da smo koristili ID dijaloga kako bismo pronasli NAG dijalog. Ovo cemo iskoristiti i ovde, samo sto ce ovde za razliku od proslog primera pretraga biti mnogo laksa.

Pritisnucemo ALT + C da bismo se vratili u glavni CPU prozor, posle cega cemo pritisnuti CTRL + F kako bismo potrazili komandu koja prikazuje ovaj NAG screen. Sada ostaje samo da dobro razmislimo koju komandu treba da trazimo u fajlu. Ovo je veoma lako (*naravno ako se setite prethodnog dialog primera*) posto se API funkciji mora proslediti ID objekta nad kojim se izvrsava neka komanda potrebno je samo potraziti PUSH 384 komandu pomocu Ollya. Nasa pretraga fajla ce nas dovesti ovde:

```
00401416  |.  6A 00          PUSH 0                                  /lParam = NULL
00401418  |.  68 2B124000    PUSH editor.0040122B                    |DlgProc = 0040122B
0040141D  |.  53             PUSH EBX                                |hOwner
0040141E  |.  68 84030000    PUSH 384                                |pTemplate = 384
00401423  |.  6A 00          PUSH 0                                  |/pModule = NULL
00401425  |.  E8 F28B0000    CALL <JMP.&KERNEL32.GetModuleHandleA>
0040142A  |.  50             PUSH EAX                                |hInst
0040142B  |.  E8 C68C0000    CALL <JMP.&USER32.DialogBoxParamA>      \DialogBoxParamA
```

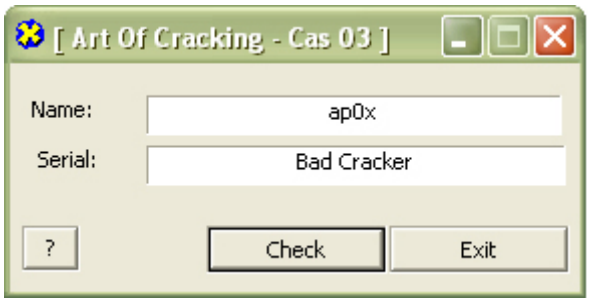
Kao sto vidimo na adresi 0040141E se ID dijaloga prosledjuje DialogBoxParam APIju zbog cega zakljucujemo da se u ovom delu koda prikazuje NAG. Posto se i GetModuleHandleA API odnosi na prikazivanje NAGa (*on odredjuje vrednost registra EAX*) mozemo i njega da uklonimo zajedno DialogBoxParamA API pozivom. Dakle da bismo uklonili ovaj NAG potrebno je da NOPujemo sve od adrese 00401416 pa do adrese 0040142B, zakljucno sa poslednjom komandom na adresi 0040142B, odnosno sa CALLom ka DialogBoxParam-u.

# 03 Cracking Serials

Sledece poglavlje govori o najcesce sretanom problemu pri reversingu. Veoma cesto se desava da je cela aplikacija ili da su neki njeni delovi zakljucani za koriscenje i da se mogu otkljucati samo pravim serijskim brojem. Ovde ce biti govora o vise tipova provere serijskih brojeva i o nacinima resavanja ovih problema. Vazno je napomenuti da ce se od ovog poglavlja iskljucivo koristi najvazniji reverserski alat, OllyDBG. Nazalost ovo poglavlje je specificno jer se prilikom "pecanja" serijskih brojeva mora nadgledati radna memorija a ne sadrzaj disassemblovanog fajla na disku. Zbog lakseg privikavanja na sve ovo prvo cu vam objasniti jedan primer na W32Dasmu.

# The Serials – *Jumps*

Pored NAG ekrana jedna od prepreka vezanih za crackovanje je i registracija ili otkljucavanje odredjenih funkcija programa pomocu rutina za proveru serijskih brojeva. Ovo je veoma cesto sretan problem pri reversingu stoga se ovo poglavlje moze smatrati jednim od kljucnih. Prvi deo ovog poglavlja ce vas nauciti kako se ovakvi programi crackuju, drugi kako da nadjete pravi serijski broj za vase ime, a sledece poglavlje kako da napisete keygenerator za ovaj primer. Ovaj primer se nalazi u folderu …\Casovi\Cas3\Serial.exe Za pocetak cemo startovati program i videcemo sta se desava. Ovo je obavezan korak koji nam omogucuje da skupimo sto je vise moguce informacija o "meti", da bismo je lakse reversovali. Startujte "metu" i unesite kao ime ap0x i kao serial 111111, pritisnite Check. Pojavice se ovo:




Ono sto smo saznali iz ovog testa je da kada unesemo pogresan serijski broj, program izbaci poruku "Bad Cracker". Ovo ce pomoci prilikom potrage za mestom gde se proverava tacnost serijskog broja. Otvorite ovu "metu" u W32Dasmu i pronadjite string "Bad Cracker". Primeticete da se pored stringa "Bad Cracker" nalazi i ovo:

```
"About..."
"AMPM "
"Bad Cracker"
"Cracked ok"
"eeee"
"Enter name !!!"
"Error"
```

Ovo je jako zanimljivo jer kako izgleda, mozda je poruka koja ce biti prikazana ako je serijski broj tacan "Cracked ok". Bez obzira na to mi cemo 2x kliknuti na "Bad Cracker" poruku i zavrsicemo ovde:

```
:00407DE9 E806BBFFFF            call 004038F4
:00407DEE 7517                  jne 00407E07
* Possible StringData Ref from Code Obj ->"Cracked ok"
|
:00407DF0 684C7E4000            push 00407E4C
:00407DF5 68B90B0000            push 00000BB9
```

```
:00407DFA A150984000            mov eax, dword ptr [00409850]
:00407DFF 50                    push eax
* Reference To: user32.SetDlgItemTextA, Ord:0000h
|
:00407E00 E8F3C7FFFF            Call 004045F8
:00407E05 EB15                  jmp 00407E1C
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00407DEE(C)
|
* Possible StringData Ref from Code Obj ->"Bad Cracker"
|
:00407E07 68587E4000            push 00407E58 <- Ovde smo
:00407E0C 68B90B0000            push 00000BB9
:00407E11 A150984000            mov eax, dword ptr [00409850]
:00407E16 50                    push eax
```

Obratimo paznju na ovaj red odmah iznad poruke o pogresnom serijskom broju:

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00407DEE(C)
```

Ovo znaci da postoji jedan uslovni (*zbog onog C, da stoji U to bi bio bezuslovni skok*) skok na adresi 00407DEE koji vodi na adresu 00407E07. Ako pogledamo sta se nalazi na toj adresi videcemo sledece:

```
:00407DEE 7517                  jne 00407E07
* Possible StringData Ref from Code Obj ->"Cracked ok"
|
:00407DF0 684C7E4000            push 00407E4C
```

Ovo znaci da ako nesto, a u ovom slucaju serijski broj, nije tacan skoci na poruku o pogresnom serijskom broju. Ako ovaj red izbrisemo (*citaj: NOPujemo*) onda ce program uvek prikazivati poruku o tacnom serijskom broju bez obzira na uneto ime ili serijski broj. To je jedan i ujedno i najlaksi nacin za resavanje ovog problema.

**Vezba:**

Posto smo ovaj primer uradili pomocu W32Dasma bilo bi dobro da ovo isto uradite i pomocu Ollya kako biste vezbali koriscenje Olly debuggera. Bolje je to da uradite sada posto ce kasnije svi primeri i vezbe biti radjene pomocu Ollya.

# The Serials – Fishing #1

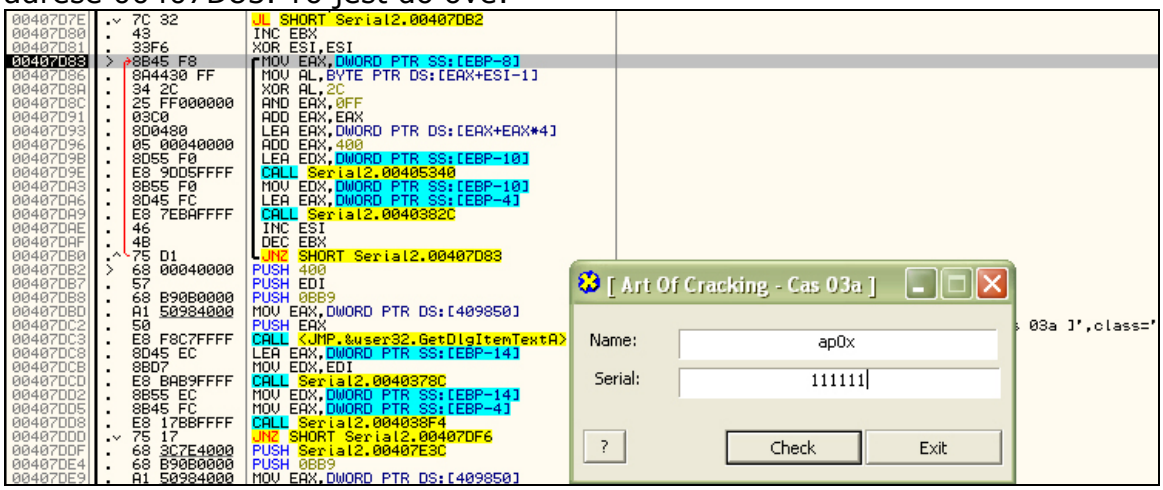
Prvi deo ovog poglavlja vas je naucio kako da program za bilo koji uneti serijski broj kaze da je ispravan, a ovaj drugi ce vas nauciti kako da nadjete pravi serijski broj za vase (*moje*) ime. Za ovo ce nam trebati malo drugaciji set alata koji cemo koristiti kako bismo pronasli serijski broj. Jedini alat koji ce nam trebati je OllyDBG. Posto je ovo prvi put da se ovaj program direktno koristi u ovoj knjizi trudicu se da veci broj stvari objasnim slikovno. Ucitajte program …\Casovi\Cas3\Serial2.exe Olly. Videcete OEP (*prvu liniju*) programa:

```
00407FA8 > $  55                PUSH EBP
00407FA9   .  8BEC              MOV EBP,ESP
```

Sada samo trebamo naci gde se to proverava serijski broj. Posto smo to mesto vec nasli u prvom delu ustedecemo malo vremena jer vec znamo kako da ga nadjemo. Ekvivalent String Reference opcije iz W32Dasma je u Olly dugme R u toolbaru. Da biste ucitali sve stringove iz fajla u R prozor kliknite desnim dugmetom na OEP,pa na **Search For -> All referenced text strings...** Tu u novootvorenom R prozoru nadjite string "Bad Cracker", i duplim klikom na njega zavrsicete ovde:

```
00407DF6  |> \68 487E4000         PUSH Serial2.00407E48   ; /Text = "Bad Cracker"
```

Ako odskrolujemo malo gore do 00407CF4 i dole do 00407E2C videcemo da je ovo Olly oznacio kao jednu celinu i da je 100% sigurno da se ovde negde proverava serijski broj. Pocnimo analizu od pocetka, od adrese 00407CF4. Selektujmo taj red i pritisnimo F2 da postavimo break-point (*pauzu*) na tu adresu. To znaci da kad program sa izvrsavanjem dodje do te adrese onda cemo mi imati punu kontrolu nad njegovim izvrsavanjem. Sada pritisnimo F9 da bismo pokrenuli program. Kao ime u nasu "metu" unesite ap0x (*sa 0 – brojem a ne sa O slovom*) i kao serijski broj 111111. Pritisnite Check i program ce zastati na adresi 00407CF4. Polako pritiskamo F8 da bismo se kretali red po red kroz program. Sve nam izgleda obicno i nevazno dok ne stignemo do adrese 00407D83. To jest do ove:



Ovo nam je zanimljivo jer serijski brojevi racunaju bas na ovaj nacin. Tu je uvek u pitanju neka petlja koja se ponavlja za sva slova imena na osnovu koga se racuna seriski broj, naravno ako on direktno zavisi od imena. Procicemo 5 puta kroz ovu petlju sa F8 i primeticemo da se sa strane u registrima (*gornjem desnom uglu*) pojavljuju neki brojevi:

```
EAX 0012FCAC
ECX 000005B8
EDX 00850CA8 ASCII "1464"
EBX 00000005
ESP 0012FC90
EBP 0012FCBC
ESI 00000000
EDI 00850888 ASCII "ap0x"
```

**1 Prolaz**

```
EAX 0012FCAC
ECX 00000702
EDX 00850CBC ASCII "1794"
EBX 00000004
ESP 0012FC90
EBP 0012FCBC
ESI 00000001
EDI 00850888 ASCII "ap0x"
```

**2 Prolaz**

```
EAX 0012FCAC
ECX 00000798
EDX 00850CA8 ASCII "1944"
EBX 00000003
ESP 0012FC90
EBP 0012FCBC
ESI 00000002
EDI 00850888 ASCII "ap0x"
```

**3 Prolaz**

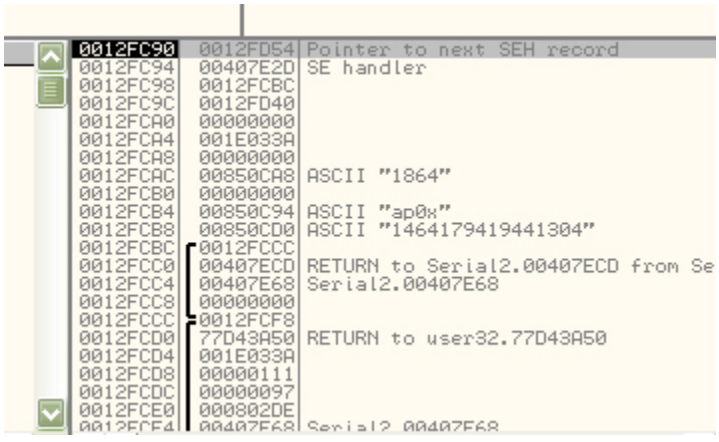
```
EAX 0012FCAC
ECX 00000518
EDX 00850CBC ASCII "1304"
EBX 00000002
ESP 0012FC90
EBP 0012FCBC
ESI 00000003
EDI 00850888 ASCII "ap0x"
```

**4 Prolaz**

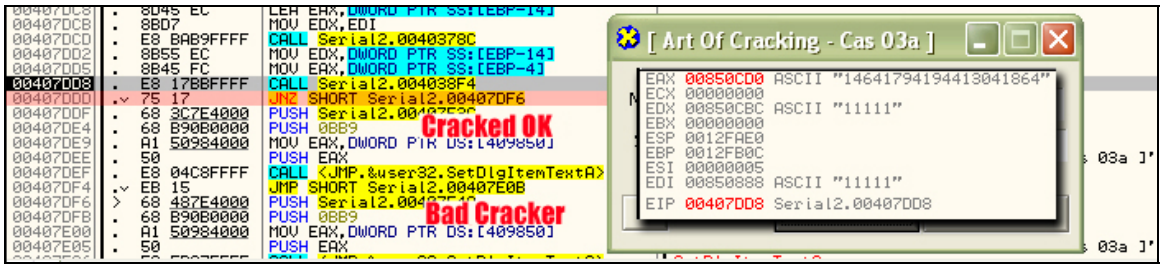
```
EAX 0012FCAC
ECX 00000748
EDX 00850CA8 ASCII "1864"
EBX 00000001
ESP 0012FC90
EBP 0012FCBC
ESI 00000004
EDI 00850888 ASCII "ap0x"
```

**5 Prolaz**

Ako pogledate u donjem desnom cosku (*STACK prozoru*) videcete sta se desava sa ovim brojevima pod navodnicima.




Ovde na slici vidimo da se brojevi polako dodaju jedan na drugi i da se ispod stringa "ap0x" polako stvara neki broj. Ovaj broj moze biti bas serijski broj koji mi trazimo. Polako sa F8 prelazimo preko redova i posmatramo registre, gore desno, cekajuci da se pojavi serijski broj koji smo uneli (111111). Ovde se desava jako zanimljiva stvar. I nas uneti serijski broj i ovaj novi koji smo primetili da se polako stvara u onoj petlji pojavljuju se ovde:




Stigli smo do adrese 00407DD8 i u registryma se nalaze stvari prikazane izdvojene na ovoj slici. EAX sadrzi moguci pravi seriski broj a EDX nas uneti lazni serijski broj. Posto je posle izvrsavanja CALLa na adresi 00407DD8 odluceno da li je serijski broj ispravan ili ne, pa zakljucujemo da ovaj CALL sluzi za uporedjivanje unetog (111111) serijskog broja i ispravnog serijskog broja u EAXu. Sledeci (crveni) red 00407DDD proverava rezultat izvrsavanja CALLa. Ako su EAX i EDX jednaki program ce nastaviti dalje a ako su razliciti onda ce prikazati poruku o pogresnom serijskom broju. Ovo znaci da je pravi serijski broj za ime ap0x jednak 14641794194413041864. Ovde moram da napomenem da se serijski brojevi uvek proveravaju direktno ispred (*iznad*) poruke o pogresnom serijskom broj. Znaci ne mnogo iznad u kodu se moraju uporediti na neki nacin dve vrednosti, vrednost naseg unetog, laznog serijskog broja i ispravnog broja za uneto ime. Kao sto ste primetili "pecanje" serijskog broja se zasniva na namernom unosenju pogresnog serijskog broja u metu kako bismo pronasli mesto gde se pogresan serijski broj proverava sa tacnim serijskim brojem. Kada pronadjemo ovo mesto preostaje nam samo da prepisemo ovaj tacan serijski broj i da ga unesemo u metu umesto naseg laznog serijskog broja.

**Vezba:**

Ako zelite mozete proveriti znanje steceno o trazenju serijskog broja za uneto ime jednostavnim unosenjem nekog drugog prozvoljnog imena i trazenjem pravog serijskog broja za to ime u ovom crackmeu ili mozete resiti crackme n0!se.exe za ime cracker koji se takodje nalazi u folderu Cas03.

**Resenje:**

Postavite break-pointe ovde: 00407BBB i 00407BE6 i videcete da se na prvoj adresi proverava da li je uneti serijski broj dugacak 29 karaktera a na drugom da se porede uneti serijski broj i tacan serijski broj.

```
00407BBB   50          PUSH EAX                          <-Poredjenje duzine
00407BBC   E8 9BC1FFFF CALL <JMP.&KERNEL32.lstrcmp>
00407BC1   85C0        TEST EAX,EAX

00407BE6   50          PUSH EAX                          <-Poredjenje serijskih
00407BE7   E8 70C1FFFF CALL <JMP.&KERNEL32.lstrcmp>
00407BEC   85C0        TEST EAX,EAX
00407BEE   74 12       JE SHORT n0!se.00407C02
```

Tacan serijski broj za ime cracker je 34308120-240170483-1463891792.

# The Serials – Fishing #2

Posto je trazenje serijskih brojeva cesto preduslov za pravljenje keygeneratora pokazacu vam kako da pronadjete serijski broj cak i kada se on ne poredi direktno (*nema ispitivanja da li je uneti serijski jednak izracunatom*) kao sto je bio slucaj u prvom primeru. Verujte mi na rec ovo ce biti za nijansu lakse nego otpakivanje mete, naravno uz pomoc odlicnog alata HexDecChara. Pocecemo tako sto cemo otvoriti metu koja se nalazi ovde …\Casovi\Cas3\vct_crackme2.exe sa Olly-jem. Na OEPu cemo videti ovo:

```
00401000  .  6A 00           PUSH 0                              ; /pModule = NULL
00401002  .  E8 6D060000     CALL vct_crac.00401674              ; \GetModuleHandleA
00401007  .  A3 0A324000     MOV DWORD PTR DS:[40320A],EAX
```

Posto svi crackmei sadrze poruke o tacno/netacno unesenim serijskim brojevima pocecemo tako sto cemo pretraziti sve string reference u ovom fajlu. Ovo cemo uraditi klikom na desno dugme -> Search for -> All referenced strings... posle cega cemo pri samom vrhu novootvorenog prozora primetiti sledece reference:

```
Text strings referenced in dumped_:, item 16
 Address=004015B9
 Disassembly=PUSH dumped_.004031AC
 Text string=ASCII "Wow, PM to me for earn 2* :D"

Text strings referenced in dumped_:, item 17
 Address=004015D2
 Disassembly=PUSH dumped_.004031C9
 Text string=ASCII "Ha ha ha, sai be't ro^`i ;-) ! Try again."
```

Ove reference predstavljaju takozvane GOOD BOY/BAD BOY poruke. One se pojavljuju u zavisnosti da li je uneti serijski broj tacan ili ne i u njihovoj fizickoj blizini se najcesce nalazi deo koda koji se koristi za proveru ispravnosti serijskog broja, stoga cemo duplim klikom na good boy poruku zavrsiti na sledecem delu koda:

```
004015B8  |>  FC             CLD                                 <- Good boy
004015B9  |.  68 AC314000     PUSH 004031AC                       ; /Text
004015BE  |.  FF35 1A324000   PUSH DWORD PTR DS:[40321A]          ; |hWnd
004015C4  |.  E8 87000000     CALL <JMP.&user32.SetWindowTextA>   ; \SetWTextA
004015C9  |.  B8 00000000     MOV EAX,0
004015CE  |.  5A              POP EDX
004015CF  |.  59              POP ECX
004015D0  |.  C3              RET
004015C9  |.  B8 00000000     MOV EAX,0
004015CE  |.  5A              POP EDX
004015CF  |.  59              POP ECX
004015D0  |.  C3              RET
004015D1  |>  FC             CLD                                 <- Bad boy
004015D2  |.  68 C9314000     PUSH 004031C9                       ; /Text
004015D7  |.  FF35 1A324000   PUSH DWORD PTR DS:[40321A]          ; |hWnd
004015DD  |.  E8 6E000000     CALL <JMP.&user32.SetWindowTextA>   ; \SetWTextA
004015E2  |.  B8 00000000     MOV EAX,0
004015E7  |.  5A              POP EDX
004015E8  |.  59              POP ECX
004015E9  \.  C3              RET
```

Ono sto nam je bitno je da pronadjemo skokove sa kojih se dolazi do bad boy poruke. Ovo je jako bitno jer se ova vrsta uslovnih skokova izvrsava samo posle nekog poredjenja, a to poredjenje je najcesce poredjenje serijskih brojeva, t.j. poredjenje unetog serijskog broja i tacnog serijskog broja. Iz ovog razloga, da bi program imao sta da poredi, unose se lazni, netacni serijski brojevi u cilju pronalaska tacnih serijskih brojeva.

Ovo je jedna cinjenica koju morate da znate, a druga takodje jako bitna je da se veoma cesto poredjenje serijskih brojeva i poruke o tacnosti, t.j. netacnosti unetog serijskog broja nalaze u istom CALLu. Ova cinjenica ce nam pomoci da pronadjemo gde i kako se to racuna serijski broj. Za sada je potrebno da izolujemo celinu u kojoj se racuna serijski broj, to jest da sagledamo ceo ovaj CALL. Ovo cemo uraditi jednostavnim scrollom na pocetak ove celine, to jest CALLa, koji pocinje bas ovde:

```
0040141C /$ 51              PUSH ECX
0040141D |. 52              PUSH EDX
0040141E |. 6A 28           PUSH 28                                 ; /Count = 28
00401420 |. 68 70324000     PUSH dumped_.00403270                 ; |Buffer = 00403270
00401425 |. FF35 1A324000   PUSH DWORD PTR DS:[40321A]             ; |hWnd
0040142B |. E8 E4010000     CALL <JMP.&user32.GetWindowTextA>       ; \GetWTextA
00401430 |. A3 FE314000     MOV DWORD PTR DS:[4031FE],EAX
00401435 |. 833D FE314000> CMP DWORD PTR DS:[4031FE],20
0040143C |. 0F85 8F010000   JNZ dumped_.004015D1
```

Kao sto se vidi sa pocetka ovog CALLa program ce uzeti podatke iz polja za unos pomocu API funkcije GetWindowTextA. Ovo je jako bitno da zapazite, to jest bitnije je da vidite na kojoj ce se adresi snimiti procitani tekst iz polja za unos. Ovaj podatak koji se prosledjuje GetWindowTextA APIju je oznacen kao buffer i sadrzi adresu 00403270 na kojoj ce biti snimljen sadrzaj polja za unos.

Na ovoj adresi ce se nalaziti tekst, kome ce meta dalje pristupati pomocu reference ka njemu, to jest preko njegove adrese. Isto kao sto ceo tekst ima adresu na kojoj se nalazi, tako i svako slovo unetog teksta ima svoju adresu. U stvari adresa celog teksta predstavlja adresu ka prvom slovu unetog teksta. Stoga cemo potraziti operacije koje se odnose na adresu 00403270. Takve operacije cemo naci ovde:

```
00401442 |. BB 70324000     MOV EBX,dumped_.00403270
00401447 |. 0FBE03          MOVSX EAX,BYTE PTR DS:[EBX]
0040144A |. 83F8 34         CMP EAX,34
0040144D |. 0F85 7E010000   JNZ dumped_.004015D1
00401453 |. 43              INC EBX
00401454 |. 0FBE03          MOVSX EAX,BYTE PTR DS:[EBX]
00401457 |. 83F8 44         CMP EAX,44
0040145A |. 0F85 71010000   JNZ dumped_.004015D1
00401460 |. 43              INC EBX
```

Kao sto vidimo prvo se sadrzaj adrese 00403270 smesta u registar EBX. Ali pazite ne smesta se u EBX sadrzaj adrese 00403270, nego se smesta referenca ka adresi, a sam EBX se koristi kao pointer ka tekstu. U slucaju kada je EBX jednak 00403270 on vodi do prvog slova unetog teksta, sto ce biti iskorisceno vec na sledecoj adresi, na kojoj ce se u EAX

smestiti heksadecimalna vrednost prvog slova unetog teksta. Primetite da kako se EBX povecava tako se u EAX smesta drugo slovo iz imena. Ovako se proveravaju sva slova iz unetog serijskog broja. Posto je ovaj kod jako dugacak i prakticno predstavlja istu stvar ovde je "zalepljen" samo jedan deo koda. Ali kako se poredi uneti serijski broj sa tacnim?

Poredjenje se radi pomocu komande CMP! Secate se da smo rekli da ce EAX sadrzati vrednosti svih unesenih slova. Iz tog razloga se EAX poredi sa raznim brojevima pomocu komande CMP. Sta su ovi brojevi? Ti brojevi predstavljaju heksadecimalne vrednosti slova tacnog serijskog broja. Tako ce prvo slovo serijskog broja iz razloga sto se EAX poredi sa 34 biti broj 4, drugo slovo ce biti D iz razloga sto se EAX poredi sa 44, i tako dalje... Da biste pronasli ceo serijski broj potrebno je da pretrazite ceo kod do za komandama CMP EAX,*. Ovo * ce predstavljati jedno hexdecimalno slovo serijskog broja. Pretvaranje heksadecimalnih vrednosti uslova mozete uraditi pomocu programa HexDecChar ili preko ASCII tablice sa adrese www.asciitable.com. Kako god da uradite pretvaranje brojeva u slova dobicete da je tacan serijski broj: 4DEAD697C78633134D3A16C75CD6E2DB

**Vezba:**

Ako zelite mozete proveriti znanje steceno o trazenju serijskog broja na crackmeu AD_CM#1.exe za ime cracker koji se takodje nalazi u folderu Cas03.

**Resenje:**

Resenje je krajnje jednostavno i sve se vidi iz isecka koda koji se nalazi ispod:

```
004010BF  |.  B8 5C304000     MOV EAX,AD_CM#1.0040305C
004010C4  |.  BB 1E304000     MOV EBX,AD_CM#1.0040301E          ; ASCII "qWeRtZ"
004010C9  |.  B9 07000000     MOV ECX,7
004010CE  |>  8A13            /MOV DL,BYTE PTR DS:[EBX]
004010D0  |.  3810            |CMP BYTE PTR DS:[EAX],DL
004010D2  |.  75 18           |JNZ SHORT AD_CM#1.004010EC
004010D4  |.  40              |INC EAX
004010D5  |.  43              |INC EBX
004010D6  |.^ E2 F6           \LOOPD SHORT AD_CM#1.004010CE
```

Kao sto vidite u ovom loopu [004010CE-004010D6] se poredi uneti string sa stringom qWeRtZ. Dakle zakljucujemo da je trazeni serijski broj qWeRtZ.

# The Serials – Fishing #3

Ovo je drugi deo poglavlja o takozvanom "pecanju" serijskih brojeva. Ovaj deo knjige ce vas nauciti kako da iskoristite Windowsove API funkcije kako bi ste nasli mesto gde se racuna pravi serijski broj. Primer za ovaj deo knjige se nalazi u folderu Cas3 a zove se keygenme1.exe. Ovaj primer sam izabrao zato sto se u njemu ne nalaze klasicni stringovi koji bi vam pomogli da nadjete mesto gde se racuna i proverara serijski broj. Otvoricemo ovaj program uz pomoc Ollya i pocecemo sa potragom za pravim serijskim brojem. Pre nego sto pocnemo moracemo da znamo sve API funkcije pomocu kojih se citaju podaci iz polja za unos. Te API funkcije se nalaze u sledecoj tabeli:

| Win16 | Win32 |
|---|---|
| **GetWindowText** | **GetWindowTextA** |
| **GetDlgitemText** | **GetDlgitemTextA** |
| **GetDlgitemInt** | **GetDlgitemIntA** |

Sve API funkcije, koje poziva nasa meta se mogu videti u prozoru *Executable modules -> View names*. Do ovoga cemo stici jednostavnim klikom na dugme E iz toolbara, selekcijom naseg exe fajla u novom prozoru i izabirom opcije View Names do koje stizemo desnim klikom na nas exe fajl. U prozoru view names cemo pronaci neke od API funkcija iz gornje tabele. Jedini API poziv koji cemo naci je GetWindowTextA pa cemo desnim klikom na njega postaviti break-point na svakoj referenci kao ovom APIu. Sada cemo se vratiti u nasu metu i kao ime i serijski broj unecemo: ap0x, 111111. Posle ovoga cemo pritisnuti OK i naci cemo se ovde:

```
00401074  |.  8B1D 9C404000  MOV EBX,DWORD PTR DS:[<&USER32.GetWindow>;
0040107A  |.  8D45 94        LEA EAX,DWORD PTR SS:[EBP-6C]
0040107D  |.  6A 64          PUSH 64                    ; /Count = 64 (100.)
```

Sada sa 100% sigurnoscu znamo da se negde ispod proverava serijski broj. Sa F8 cemo ispitati kod ispod nase trenutne pozicije.

```
004010AC  |.  50             PUSH EAX                          <- Ucitaj ime
004010AD  |.  E8 CE020000    CALL keygenme.00401380            <- Vraca duzinu imena
...
004010CC  |> \8D85 30FFFFFF LEA EAX,DWORD PTR SS:[EBP-D0]     <- Ucitaj uneti serijski
...
004010D3  |.  E8 A8020000    CALL keygenme.00401380            <- Vraca duzinu serijskog
...
00401104  |> /8A4C15 94      /MOV CL,BYTE PTR SS:[EBP+EDX-6C] <- Pocetak generisanja
00401108  |.  0FBEC1         |MOVSX EAX,CL                     <- tacnog serijskog broja
...
004011E1  |.^\7C E0          \JL SHORT keygenme.004011C3       <- Kraj generisanja
```

Posle ove petlje sledi sredjivanje rezultata u ASCII oblik pomocu wsprinfA funkcije. Na zadnoj adresi, 004011FE, cete primetiti da se u EAXu nalazi neki

```
004011E3  |> \52            PUSH EDX                                  ; /<%lu>
004011E4  |.  52            PUSH EDX                                  ; |<%X>
...
004011F1  |.  FF15 B0404000  CALL DWORD PTR DS:[<&USER32.wsprintfA>] ;\wsprintfA
...
004011FE  |.  8D85 CCFEFFFF  LEA EAX,DWORD PTR SS:[EBP-134]     ;  <- EAX sadrzi serijski
```

broj koji moze biti pravi serijski broj. Ako pogledate malo ispod videcete CALL koji poredi ovaj broj sa nasim unetim 111111, stoga je sadrzaj EAXa pravi serijski broj. Uspeli smo, tacan serijski je ED0C68403977013312.

## The Serials – Fishing #4

Prosli primer je bio jednostavan posto je program poredio serijske brojeve kao stringove. Ali sta da radimo kada ovo nije slucaj, sta kada se porede brojevne vrednosti? U ovom slucaju resenje nije uvek lako uocljivo ali mi cemo kao i uvek uspeti da resimo sve nase probleme sa lakocom. Primer za ovaj deo knjige se nalazi u folderu Cas3 a zove se KeyGen-me#1.exe i njega cemo otvoriti sa Olly-jem.

Posto smo odlucili da vezbamo postavljanje break-pointa na APIje i ovde cemo otici u Executable modules window i tamo cemo pronaci sve APIje u nasem programu selekcijom glavnog exe fajla koji reversujemo i klikom na CTRL+N. Od APIja u fajlu ce se nalaziti samo sledece:

```
Address     Section     Type        (  Name
0040200C    .rdata      Import      (  USER32.DialogBoxParamA
00402010    .rdata      Import      (  USER32.EndDialog
00402004    .rdata      Import      (  KERNEL32.ExitProcess
00402018    .rdata      Import      (  USER32.GetDlgItem
0040201C    .rdata      Import      (  USER32.GetDlgItemTextA
00402000    .rdata      Import      (  KERNEL32.GetModuleHandleA
00402020    .rdata      Import      (  USER32.LoadIconA
00402024    .rdata      Import      (  USER32.MessageBoxA
00402028    .rdata      Import      (  USER32.SendMessageA
00402014    .rdata      Import      (  USER32.wsprintfA
```

Posto je ocigledno da se za uzimanje teksta iz prozora koristi API GetDlgItemTextA (*jer se GetDlgItem koristi za pristupanje objektima koji se nalaze u formi [prozoru] a ne za citanje teksta iz njih*) postavicemo break-point na njega klikom na desno dugme i selekcijom opcije Set break-point on every reference. Kada smo ovo uradili ostaje nam da startujemo program klikom na F9 u Olly-ju i da u nasu metu unesemo sledece podatke: kao ime cemo uneti ap0x a kao serijski broj 111111. Naravno ovi podaci su pogresni ali mi cemo posle klika na Okey! dugme biti na pravom putu da pronadjemo tacan serijski broj jer smo postavili break-point na API koji se koristi za citanje unetih podataka. Dakle posle klika na dugme Okey! mi cemo se naci ovde:

```
004010EE   .  E8 23010000      CALL <JMP.&USER32.GetDlgItemTextA>      ; GetDlgItemTextA
```

Posle klika na F8 posto zelimo da predjemo preko ovog poziva, to jest zelimo da se ovaj API izvrsi, zavrsicemo ovde:

```
00401216   JMP NEAR DWORD PTR DS:[<&USER32.GetDlgIt>;  USER32.GetDlgItemTextA
```

Posto je ovo direktan poziv ka user32.dll uklonicemo ovaj break-point klikom na F2. Iako smo ovo uradili klikom na F8 dolazimo u user32.dll fajl ali ovo ce se desiti samo ovaj put zbog cega je potrebno izvrsiti sve komande u ovom .dll fajlu sa F8 sve dok ne izvrsimo i poslednju RET komandu i vratimo se iz .dll fajla u kod .exe fajla, to jest ovde:

```
004010F3   .  0BC0             OR EAX,EAX
004010F5   .  75 19            JNZ SHORT KeyGen-m.00401110
```

A kao sto vidimo ovde se samo proverava da li smo uneli nase ime u prozor. Posto smo ovo uradili pritiskacemo F8 sve dok ne izvrsimo sve komande i dok se ne nadjemo ovde:

```
0040111C   .  FF75 08          PUSH DWORD PTR SS:[EBP+8]           ; |hWnd
0040111F   .  E8 F2000000      CALL <JMP.&USER32.GetDlgItemTextA> ; \GetDlgItemTextA
```

```
00401124   . 0BC0            OR EAX,EAX
```
Primeticete da se na ekranu pojavio string 111111 odmah posle izvrsavanja poslednjeg GetDlgItemTextA APIa. Ovo znaci da je meta procitala i sadrzaj drugog polja. Ovde se OR EAX,EAX komanda ponovo odnosi na proveru ispunjenosti serial polja u glavnom prozoru crackmea. Posto smo i ovo polje ispunili ponovo pritiskamo F8 kako bismo dosli dovde:
```
00401148   . FF75 08          PUSH DWORD PTR SS:[EBP+8]            ; |hWnd
0040114B   . E8 C6000000      CALL <JMP.&USER32.GetDlgItemTextA>   ; \GetDlgItemTextA
00401150   . 33D2             XOR EDX,EDX
```
Ako sada pogledate sadrzaj registra EAX videcete da je EAXu dodeljena vrednost 4 odmah posle izvrsavanja GetDlgItemTextA API poziva. Bez nekog velikog razmisljanja zakljucujemo da ovaj API stavlja duzinu unetog podatka iz polja koje se cita u EAX registar. Izvrsavajuci kod sa F8 dolazimo do sledeceg isecka koda koji se ocigledno koristi za izracunavanje serijskog broja na osnovu unetog imena:
```
00401158   . BE 80334000      MOV ESI,KeyGen-m.00403380          ;  ASCII "ap0x"
0040115D   >  8A1C31          MOV BL,BYTE PTR DS:[ECX+ESI]
00401160   . 03C3             ADD EAX,EBX
00401162   . 41               INC ECX
00401163   . 80FB 00          CMP BL,0
00401166   .^ 75 F5           JNZ SHORT KeyGen-m.0040115D
```
Ovaj deo koda se izvrsava cetiri puta (*od adrese 0040115D pa do 00401166*) jer je nase ime dugacko cetiri slova. Dakle zakljucujemo da program racuna serijski broj na osnovu svih unetih slova imena. Pratite vrednost registra EAX i nacin na koji se ona povecava za svaki od ova cetiri prolaza. Posto se konacno izvrsi i cetvrti prolaz izvrsice se i sledeci kod koji ce dodatno izmeniti vrednost EAX registra:
```
00401168   . BA 28000000      MOV EDX,28
0040116D   . F7E2             MUL EDX
0040116F   . 83C0 19          ADD EAX,19
```
ali ovo i nije bitno posto sada samo trazimo tacan serijski broj. Ako pogledate malo dole videcete da se ova vrednost iz EAXa pretvara pomocu wsprintfa APIa u decimalni oblik, ovde:
```
00401172   . 50               PUSH EAX                           ; /<%d>
00401173   . 68 64334000      PUSH KeyGen-m.00403364             ; |Format = "%d"
00401178   . 68 9A334000      PUSH KeyGen-m.0040339A             ; |s = KeyGen-m.0040339A
0040117D   . E8 7C000000      CALL <JMP.&USER32.wsprintfA>; \wsprintfA
00401185   . BA 9A334000     MOV EDX,KeyGen-m.0040339A          ;  ASCII "15105"
```
a rezultat ovog APIa je vise nego ocigledan. 15105 moze biti tacan serijski broj za uneto ime ap0x. Ovo mozete proveriti posle ali pogledajte prvo deo koda za poredjenje serijskih brojeva koji se ocigledno nalazi ovde:
```
004011A5   . 8B06             MOV EAX,DWORD PTR DS:[ESI]
004011A7   . 8B1A             MOV EBX,DWORD PTR DS:[EDX]
004011A9   . 3BC3             CMP EAX,EBX
004011AB   . 75 15            JNZ SHORT KeyGen-m.004011C2   <- Bad Boy
```
Kao sto vidite u EAX i EBX se smestaju dve vrednosti. Prva je nas uneti serijski broj (EAX) a druga je tacan serijski broj (EBX). Posto se registri porede pomocu CMP EAX,EBX komande zakljucujemo da je tacan serijski broj upravo sadrzaj registra EBX to jest tacan serijski broj bi u heksadecimalnom obliku bio 30313531. Ovaj heksadecimalni DWORD mozemo da pretvorimo u trazeni serijski broj pomocu HexDecChara klikom na Invert registers i selekcijom drugog prikaza za Char opciju. Tako konacno saznajemo da je tacan serijski broj za uneto ime ap0x 1510 (*mada je i 15105 zbog nacina provere unetog serijskog broja tacan serijski broj*).

# The Serials – Fishing #5

Vec smo naucili razne nacine za hvatanje serijskih brojeva koji se nalaze u formi stringa u memoriji fajla, ali sta ako se serijski brojevi ne nalaze u ovom obliku. Ovaj slucaj cemo razmotriti na meti ..\Cas03\Crackme1.exe koju cemo otvoriti pomocu Ollya. Kao sto vidite ova meta ima ukupno cetiri nivoa, a mi cemo ovde resiti samo tri. Cetvrti nivo ostavljam vama za vezbu.
Posle otvaranja mete pomocu Ollya potrazicemo karakteristicne stringove koji se odnose na level1. I u string referencama cemo videti sledece:

```
Text strings referenced in Crackme1:.text, item 4
 Address=004012E0
 Disassembly=PUSH Crackme1.00403000
 Text string=ASCII "Level 1 Complated"
```

Stoga cemo 2x kliknuti na ovaj string i naci cemo se ovde:

```
004012E0  |.  68 00304000      PUSH Crackme1.00403000       ; |Text = "Level 1 Complated"
```

Posto se negde iznad proverava tacnost serijskog broja odskrolovacemo na pocetak ovog CALLa i postavicemo break-point na njegov pocetak, odnosno postavicemo break-point na 004012A0. Posto smo ovo uradili mozemo da unesemo u metu serijski broj po zelji. Ja sam uneo 111111 i pritisnuo Register sto me je odvelo do mog break-pointa. Dalje slobodno traceujemo kroz kod sa F8 sve do adrese 004012B4 gde se ocigledno proverava da li je unet bilo koji serijski broj. Posto je EAX jednak 6, odnosno duzini unetog serijskog broja, a ESP pokazuje na uneti serijski broj, znamo da smo na pravom mestu. Dalje se krecemo kroz kod sve dok ne dodjemo do:

```
004012BB  |.  8136 504F5453 XOR DWORD PTR DS:[ESI],53544F50
004012C1  |.  8176 04 454C5>XOR DWORD PTR DS:[ESI+4],52554C45
```

Ocigledno je da je ovo kod za proveru tacnosti serijskog broja iz razloga sto se odmah ispod ovoga nalaze skokovi koji odlucuju o tome da li ce se prikazati poruka o tacnosti unetog serijskog broja. Dakle sada treba da saznamo kako se to proverava uneti serijski broj. Da bismo ovo uradili pogledacemo sta se to nalazi na ESI adresi kada smo sa izvrsavanjem programa dosli do adrese 004012BB. Ako uradimo Follow in dump na ESI registar videcemo ovo:

```
0012FB00  31 31 31 31 31 31 00 00  111111..
```

Dakle prve cetiri cifre naseg seriskog broja se XORuju sa 53544F50. A onda se na adresi 004012C1 i druge cetiri cifre naseg unetog serijskog broja XORuju sa 52554C45. Kada se izvrsi i ova druga komanda sledi provera tacnosti unetog serijskog broja pomocu dve sledece komande:

```
004012C8  |.  813E 123D3525  CMP DWORD PTR DS:[ESI],25353D12
004012CE  |.  75 21          JNZ SHORT Crackme1.004012F1
004012D0  |.  817E 04 2A6D7> CMP DWORD PTR DS:[ESI+4],73746D2A
004012D7  |.  75 18          JNZ SHORT Crackme1.004012F1
```

Kao sto vidimo XORovane vrednosti prvog i drugog dela serijskog broja se porede sa 25353D12 i 73746D2A. Ovo nije nikakav problem. Malo reversne matematike i resicemo nas problem, to jest pronacicemo tacan serijski broj. Dakle

```
Deo_serijskog_broja_1 XOR 53544F50 = 25353D12
Deo_serijskog_broja_2 XOR 52554C45 = 73746D2A
```

Posto je XORovanje revezibilna funkcija onda imamo:

**Deo_serijskog_broja_1 = 53544F50 XOR 25353D12**
**Deo_serijskog_broja_2 = 52554C45 XOR 73746D2A**

A ovo mozemo da izracunamo:

**Deo_serijskog_broja_1 = 76617242**
**Deo_serijskog_broja_2 = 2121216F**

Ostaje nam jos samo da pretvorimo ove heksadecimalne brojeve u string koji se unosi. Srecom po nas HexDecChar ovo moze da uradi mnogo brze od nas i stoga u njemu izabiramo mod 2 umesto moda 1 jer pretvaramo ceo sadrzaj registra u string. Kada paste-ujemo Deo_serijskog_broja_1 kao Char ce se pokazati string varB. Ali posto su registri uvek invertovani moramo da uradimo i invert register, ondnosno klikom na zelenu strelicu dobijamo umesto varB, string Brav. A kada isti postupak uradimo i za Deo_serijskog_broja_2 dobijamo string o!!!. Dakle ceo seriski je Deo_serijskog_broja_1 + Deo_serijskog_broja_2 = Bravo!!!. Konacno kad unesemo tacan serijski broj u metu dobijamo odgovor Level 1 ComplAted!!! Predjimo sada na Level 2.

## The Serials – Fishing #6

Posto smo sa izuzetnom lakocom resili Level 1, vreme je da predjemo na Level2, koji donosi par novina. Dakle ponovo cemo kao i za prvi Level pronaci gde se to porede uneti i tacan serijski broj. Ovo cemo uraditi pomocu string referenci, kao i prosli put:

```
Text strings referenced in Crackme1:.text, item 8
 Address=00401355
 Disassembly=PUSH Crackme1.00403012
 Text string=ASCII "Level 2 Complated"
```

Duplim klikom na ovaj string dolazimo ovde:

```
00401355  |.  68 12304000      PUSH Crackme1.00403012      ; |Text = "Level 2 Complated"
```

Kao i prosli put skrolujemo do samog pocetka CALLa i postavljamo break-point na 00401309 posle cega u nasu metu unosimo lazni serijski broj 111111 i pritiskamo dugme Register. Ovo nam zaustavlja izvrsavanje nase mete i stavlja nas na nas break-point. Posto se sada nalazimo gde treba mozemo da tracujemo kroz kod sa F8. Ovo radimo sve dok ne dodjemo do sledece adrese:

```
0040131D  |.  83F8 11          CMP EAX,11
00401320  |.  75 44            JNZ SHORT Crackme1.00401366
```

Posto se ova provera CMP EAX,11 nalazi odmah ispod GetWindowTextA API poziva zakljucujemo da je potrebna duzina serijskog broja 11h odnosno 17 karaktera. Stoga umesto 111111 kao serijski broj unosimo 1111111111111111. Ponovo zastajemo kod naseg break-pointa i tracujemo kroz kod sve do JNZ skoka koji se ovaj put ne izvrsava! Tracujemo dalje sve dok ne dodjemo do:

```
00401325  |.  807E 08 2D       CMP BYTE PTR DS:[ESI+8],2D
00401329  |.  75 3B            JNZ SHORT Crackme1.00401366
```

Sada vidimo da se jedan bajt poredi sa 2D, odnosno sa karakterom '-'. Ako pogledamo malo ispod CPU windowa videcemo sledeci tekst:

**Stack DS:[0012FB08]=31 ('1')**
Njegovom selekcijom i klikom na Follow address in Dump videcemo ovo:
**0012FB08  31 31 31 31 31 31 31 31  11111111**
**0012FB10  31 00 00 00 4C 10 40 00  1...L.@.**
Odnosno videcemo da 9 slovo u unetom serijskom broju mora biti '-'. Ponovo unosimo novi serijski broj u nasu metu samo sto sada umesto prethodno unesenog serijskog broja unosimo 11111111-11111111. Kada sada budemo stigli do poredjenja sa 2D videcemo da se ovaj JNZ skok vise ne izvrsava i da cemo nastaviti sa daljim izvrsavanjem koda, stizuci do:
**00401340  |.  33C3            XOR EAX,EBX**
**00401342  |.  05 444F4F47     ADD EAX,474F4F44**
**00401347  |.  3D 504F5453     CMP EAX,53544F50**
Primetite samo da se pre ovoga nas serijski broj podelio na dve celine koje se sada nalaze u registrima EAX i EBX. Posto su EAX i EBX nepromenjeni u odnosu na unetu formu zakljucujemo da program trazi unosenje serijskog broja u heksadecimalnoj formi. Da vidimo sta se dalje desava sa registrima EAX i EBX. Oni se dalje medjusobno XORuju i na vrednost EAXa se dodaje 474F4F44, posle cega se EAX poredi sa 53544F50. Dakle ponovo imamo jednacinu:

**(Deo_serijskog_1 XOR Deo_serijskog_2) + 474F4F44 = 53544F50**

odnosno

**53544F50 - 474F4F44 = Deo_serijskog_1 XOR Deo_serijskog_2**
**C05000C = Deo_serijskog_1 XOR Deo_serijskog_2**

Ali sta dalje? Sada imamo dva nepoznata dela serijskog broja! To stvarno nema veze u ovom slucaju posto se ni jedan deo ne poredi direktno, vec se samo poredi rezultat jednacine od gore. Posto se serijski broj proverava na ovaj nacin postoji prakticno neogranicen broj serijskih brojeva. Ne zavisno od svega ovoga mi mozemo da pretpostavimo da je Deo_serijskog_1 = 11111111 i onda imamo sledecu jednacinu:

**C05000C = 11111111 XOR Deo_serijskog_2,** odnosno
**Deo_serijskog_2 = C05000C XOR 11111111.**
**Deo_serijskog_2 = 1D14111D.**

Tako da je nas serijski broj jednak Deo_serijskog_1 + '-' + Deo_serijskog_2. I konacno jedan o mnogo tacnih serijskih brojeva je 11111111-1D14111D. Ostavljam vama da pronadjete jos par tacnih serijskih brojeva za ovaj level.

## The Serials – Fishing #7

Videli smo da ni drugi level nije bio velika prepreka za nas, a sta se to nalazi u Levelu 3 ostaje nam da vidimo. Bez neke velike mudrosti pronalazimo rutinu koja se koristi za proveru levela 3, i postavljamo break-point na adresu 0040137E posle cega mozemo da unesemo lazne podatke u nasu metu. Unecemo ap0x kao ime i 111111 kao serijski broj. Posle klika na

dugme Register dolazimo do naseg break-pointa. Sada mozemo da tracujemo dalje kroz kod sve dok ne dodjemo dovde:

```
00401393  |.  E8 28010000      CALL <JMP.&USER32.GetDlgItemInt>      ; \GetDlgItemInt
```

Kao sto znamo GetDlgItemTextA vraca sadrzaj nekog EditBoxa kao string, a sta onda radi GetDlgItemInt? Mozete konsultovati dokumentaciju ili mozete jednostavno pogledati kod ispod i bice vam jasno da ova funkcija takodje cita EditBox ali umesto stringa vraca vrednost u brojevnom obliku. Tracejuci dalje stizemo do koda:

```
004013A8  |.  E8 19010000      CALL <JMP.&USER32.GetWindowTextA>   ; \GetWindowTextA
```

koji cita ime koje smo uneli u metu. Posto je kod za proveru serijskog broja blizu zakljucujemo da sledeci loop racuna tacan serijski broj na osnovu unetog imena:

```
004013BB  |> /8D0C9B          /LEA ECX,DWORD PTR DS:[EBX+EBX*4]
004013BE  |. |8D0C4B          |LEA ECX,DWORD PTR DS:[EBX+ECX*2]
004013C1  |. |0FBE16          |MOVSX EDX,BYTE PTR DS:[ESI]
004013C4  |. |83C1 21         |ADD ECX,21
004013C7  |. |0FAFD1          |IMUL EDX,ECX
004013CA  |. |03FA            |ADD EDI,EDX
004013CC  |. |43              |INC EBX
004013CD  |. |46              |INC ESI
004013CE  |. |3BD8            |CMP EBX,EAX
004013D0  |.^\7C E9           \JL SHORT Crackme1.004013BB
```

Sam algoritam za generisanje serijskog broja i nije toliko bitan posto cemo mi samo pronaci tacan serijski broj za nase (moje) ime. Dakle mozemo slobodno da postavimo break-point odmah ispod ovog loopa. Postavicemo break-point na sledecu komandu

```
004013D2  |.  3B7D FC          CMP EDI,DWORD PTR SS:[EBP-4]
```

Ovo radimo iz razloga sto se odmah ispod ove komande nalazi JNZ skok od kojeg zavisi da li ce se prikazati poruka o tacnom, odnosno netacnom serijskom broju. Iz ovog razloga cemo pogledati sta se to nalazi u registru EDI, ali i na adresi EBP-4. Ono sto vidimo u registrima je:

```
EAX 00000004
ECX 00000042
EDX 00000C60
EBX 00000004
ESP 0012FAEC
EBP 0012FB1C
ESI 0012FAFB
EDI 0000351C <- Vrednost koja se poredi
```

Dok na stacku (EBP-4) imamo sledece podatke:

```
Stack SS:[0012FB18]=0001B207 <- Vrednost sa kojom se EDI poredi
EDI=0000351C
```

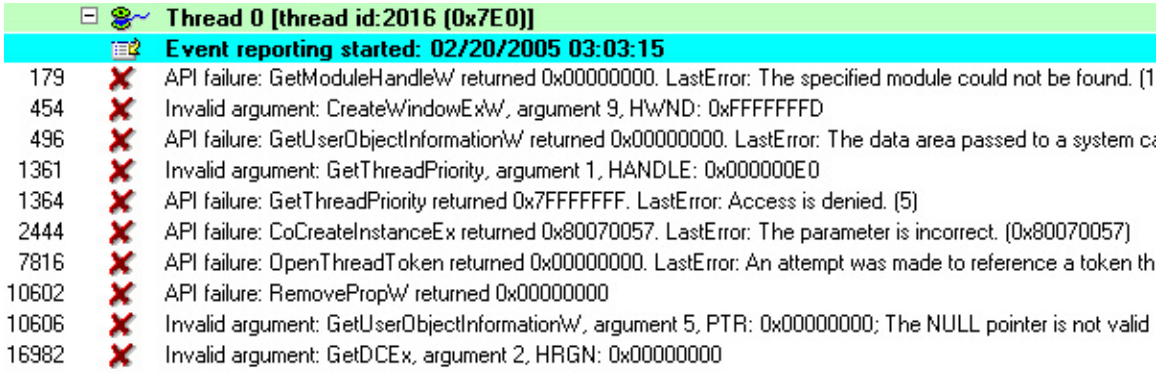
Posto znamo da je decimalno 11111 jednako 1B207h zakljucujemo da se na adresi EBP-4 nalazi nas uneti serijski broj, sto ujedno znaci da se u registru EDI nalazi tacan serijski broj! Ali da je to bas tako? Nije, 351C nije tacan serijski broj za ime ap0x iz razloga sto se porede heksadecimalne vrednosti unetog i izracunatog serijskog broja! Dakle posto je 111111 pretvoreno u hexadecimalno i 351C mora biti pretvoreno nazad u decimalno kako bi uneti serijski broj bio ispravan! Ovo znaci da je tacan serijski broj za ime ap0x u stvari 13596, sto mozemo i proveriti njegovim unosenjem u metu. I to bi bilo to sto se tice ove mete, ostao nam je jos samo Level 4, ali iz razloga sto je njegovo resavanje isto kao i resavanje Levela 3, to ostavljam vama da sami resite.

# The Serials – Smart Check #1

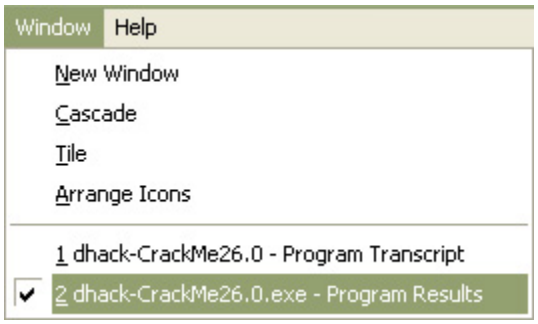
Da bismo "ulovili" serijske brojeve u programima koji su napisani u Visual Basicu najbolje je da koristimo program koji je specijalno napisan za ovu vrstu kompajlera, koristicemo Smart Check. Pre nego sto pocnete sa koriscenjem ovog programa izuzetno je vazno da prvo pravilno konfigurisete program. Sama konfiguracija programa je detaljno objasnjena na pocetku knjige pa ako niste podesili Smart Check na ovaj nacin predlazem da se vratite na pocetak ili ako i posle ovoga imate problema sa konfiguracijom vaseg SCa iskoristite dva registry fajla koja se nalaze u sledecoj arhivi ...\Settings\SmartCheck.zip.

Kada uspesno konfigurisete Smart Check mozete da predjete na nasu metu za ovaj deo poglavlja, ...\Cas03\dhack-CrackMe26.0.exe. Kada otvorite metu pomocu SCa pratite sledece korake:

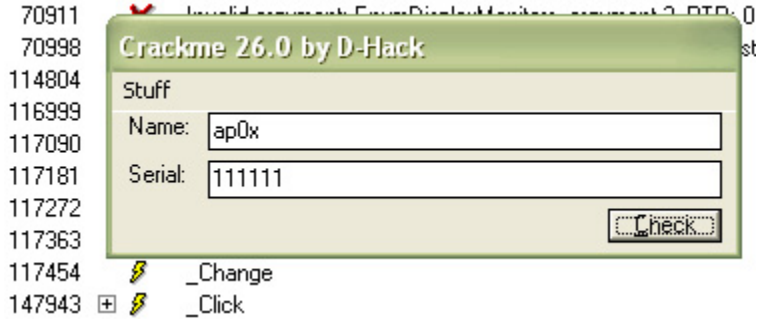
- Pritisnite dugme start (F5)
- Posle ovoga biste trebali da vidite ovo:




- A ako ne vidite ovaj prozor kliknite na sledece posle se prozor od gore mora pojaviti a ako se i tada ovi podaci ne pojave na ekranu pogledajte da li je ukljucen Event reporting, klikom na Program -> Event reporting...
Ova opcija mora biti ukljucena kako bi program uopste prihvatao podatke koje mu salje aplikacija koju debuggujemo.

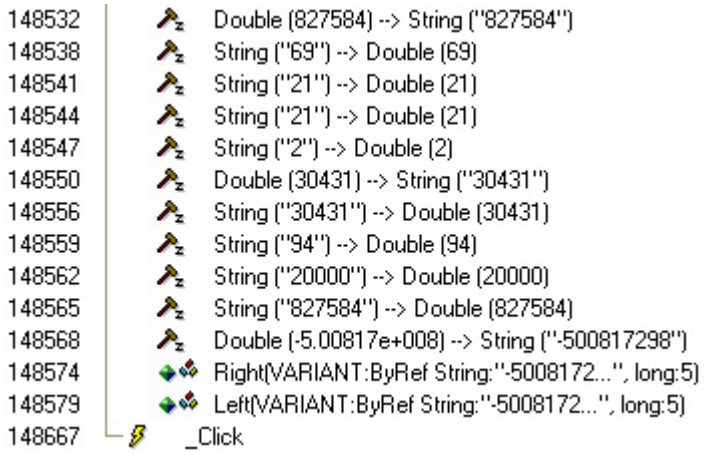



- Sada cemo u debuggovanu aplikaciju uneti lazne podatke i pritisnuti Check.



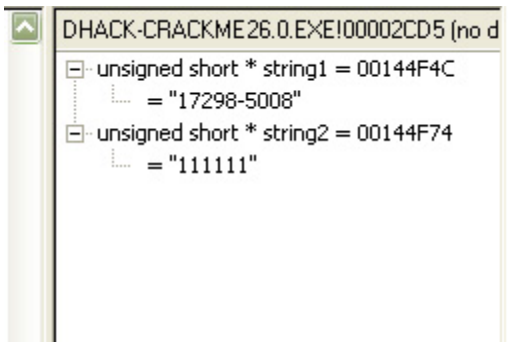

Kao sto se vidi na slici pored, SC je prepoznao neke akcije koje smo uradili. Prepoznao je akciju unosenja serijskog broja u odredjeno polje i prepoznao je akciju klika na dugme Check.

- Posto smo pritisnuli dugme Check i program je prepoznao ovo kao akciju ostaje nam rasirimo granu dogadjaja koji su se desili prilikom klika. Stoga cemo kliknuti na [+].
  Kada kliknete na dugme [+] Click pojavice se sledeci podaci na ekranu. Naravno ovo je samo deo svih dogadjaja koji su se dogodili prilikom klika. Nama je interesantan samo sam kraj ovog Clicka jer se na samom kraju poredi serijski broj. Iz ovog razloga cemo selektovati red 148574 i pritisnuti View -> Show All events...




- Posle toga ce se pojaviti ovo:

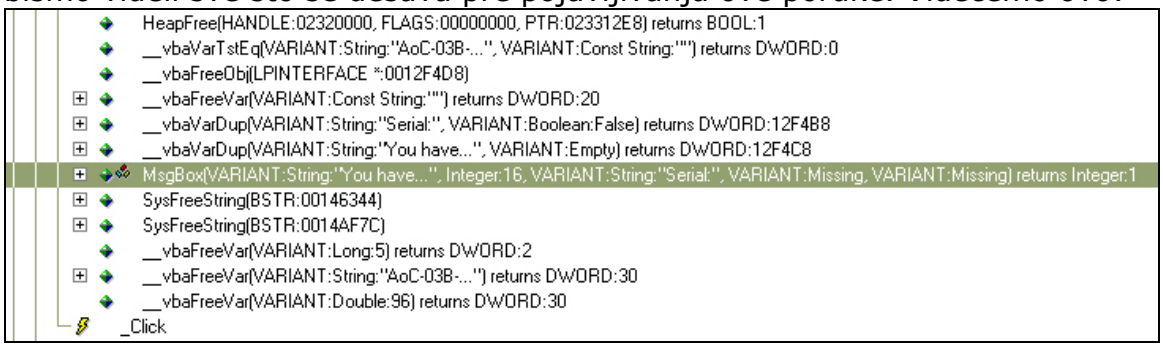

Sada bismo trebali da pronadjemo neko poredjenje naseg laznog serijskog broja 11111 i tacnog serijskog broja. Posle malo skrolovanja pronasli smo gde se porede ove dve vrednosti. Posto se ne vide svi brojevi iz tacnog seriskog broja povucicemo scrollbar sa strane u levo, posle cega cemo videti ovo:



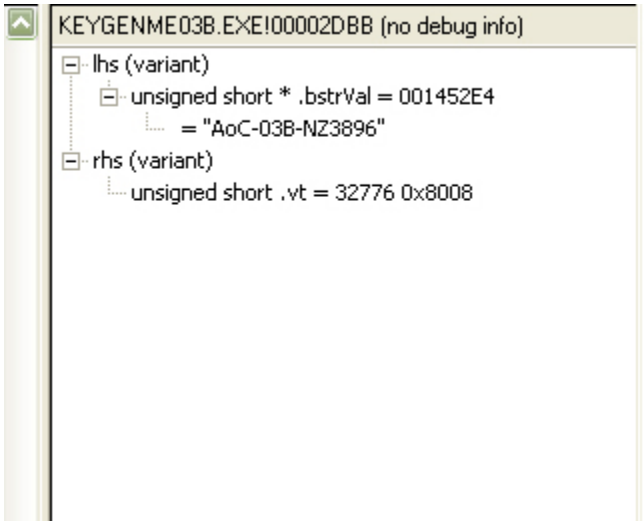

I sledeca slika sve sama govori. Posto se porede vrednosti 111111 i 17298-5008, zakljucujemo da je pravi serijski broj upravo ova druga vrednost (*serijski ce se razlikovati na vasem racunaru*).
SC je u ovom slucaju bio od velike pomoci, stoga predlazem da prvo probate da pronadjete podatke o VB meti pomocu njega a tek onda da predjete na Olly.

# The Serials – Smart Check #2

Cetvrti deo ovog poglavlja ce vas nauciti kako da nadjete tacan serijski broj za bilo koje uneto ime uz pomoc Smart Checka. Ovaj program je specifican po svojoj upotrebi i koristi se za programe koji su pisani iskljucivo u Visual Basicu. Za potrebe ovog dela napravljen je specijalan primer koji se nalazi ovde …\Casovi\Cas3\keygenme03b.exe i njega cemo otvoriti pomocu SCa. Pritisnite F5 da biste startovali ovu "metu". Kada se ona pojavi u nju unesite kao ime ap0x a kao serijski broj 111111 i pritisnite Check. Pojavila se poruka o pogresnom serijskom broju. Kliknucete na OK i u SCu potrazicemo ovu poruku rucno (*moze i preko opcije find*). Pored zadnjeg reda [+]Click vidimo da imamo [+] i da ta grana moze da se rasiri. Uradimo to i na samom kraju ove grane videcemo jedno Click koje oznacava kraj te grane a odmah iznad njega MsgBox komandu sa tekstom o pogresnom serijskom broju. Selektovacemo nju i u meniju cemo kliknuti na ***View -> Show All events…*** da bismo videli sve sto se desava pre pojavljivanja ove poruke. Videcemo ovo:




Ono sto mora da se desi pre pojavljivanja ovog MsgBoxa je neko poredjenje vrednosti. Obicno je u pitanju prvo poredjenje iznad samog MsgBoxa. Prvo poredjenje koje vidimo iznad MsgBoxa je _vbaVarTstEq koje poredi neku vrednost koja lici na tacan serijski broj. Posto se ne vidi ceo, selektovacemo




ga i desno ako se ne vidi ovaj deo ekrana, povucite ScrollBar levo i videcete ceo taj string u trecem redu. Dakle mozete isprobati ovaj serijski broj sa imenom ap0x. Videcete da je serijski ispravan. Ovo je najjednostavniji primer na kojem se zasnivaju skoro svi programi pisani u Visual Basicu, tako da je serijske brojeve veoma lako naci. Ako zelite da vezbate mozete pronaci serijski u meti ...\Cas3\CrackMe1_bf.exe

# The Serials – Computer ID

Do sada smo obradili programe koji generisu serijske brojeve na osnovu unetog imena. Ono sto preostaje u ovom opusu je generisanje serijskih brojeva na osnovu nekog hardwareskog parametra, to jest na osnovu computer IDa. Ono sto vecini korisnika nije jasno je da svaki program za sebe generise poseban computer ID, koji se razlikuje od programa do programa. Iako se serijski broj zasniva na nekom computer IDu sam princip generisanja je isti posto se serijski broj opet generise na osnovu neke konstante. Za potrebe ovog dela knjige sa interneta sam skinuo sledeci crackme …\Casovi\Cas3\abexcm5.exe Posto je Abexov program pisan u TASMu jedini alat koji ce nam trebati je OllyDBG. Otvorite Olly i ucitajte ovaj fajl u njega. Kao sto primecujemo ovo je jako kratak program, pocinje na adresi 00401000 a zavrsava se na adresi 00401171. Takodje su ocigledne dve poruke koje se nalaze u programu, *"The serial you entered is not correct!"* i *"Yep, you entered a correct serial!".* Primeticemo komandu RET 10 na adresi 00401069 koja oznacava da stvaran deo za proveru serijskog broja ne pocinje na adresi 00401056 nego na adresi 0040106C, stoga cemo postaviti break-point na tu adresu.

```
00401069  |.  C2 1000           RET 10
0040106C  |>  6A 25             PUSH 25
0040106E  |.  68 24234000       PUSH abexcm5.00402324
00401073  |.  6A 68             PUSH 68
00401075  |.  FF75 08           PUSH DWORD PTR SS:[EBP+8]
00401078  |.  E8 F4000000       CALL <JMP.&USER32.GetDlgItemTextA>
```

Sa F9 cemo startovati program i bez ikakvog unosenja serijskog broja cemo pritisnuti Check. Normalno, program je zastao na break-pointu. Evo sta se desava... Prvo program ucitava nas uneti serijski broj u memoriju na adresi 00401078. Dalje se prosledjuju parametri funkciji koja vraca ime particije na kojoj se nalazi, to jest takozvani volume label na adresi 00401099.

```
0040107D  |.  6A 00             PUSH 0
0040107F  |.  6A 00             PUSH 0
00401081  |.  68 C8204000       PUSH abexcm5.004020C8
00401086  |.  68 90214000       PUSH abexcm5.00402190
0040108B  |.  68 94214000       PUSH abexcm5.00402194
00401090  |.  6A 32             PUSH 32
00401092  |.  68 5C224000       PUSH abexcm5.0040225C
00401097  |.  6A 00             PUSH 0
00401099  |.  E8 B5000000       CALL <JMP.&KERNEL32.GetVolumeInformation>
```

Na adresi 004010A8 se poziva funkcija lstrcatA koja spaja dva stringa. Prvi string je volume label a drugi je uvek isti i jednak je 4562-ABEX.

```
0040109E  |.  68 F3234000       PUSH abexcm5.004023F3        ; /StringToAdd = "4562-ABEX"
004010A3  |.  68 5C224000       PUSH abexcm5.0040225C        ; |ConcatString = ""
004010A8  |.  E8 94000000       CALL <JMP.&KERNEL32.lstrcatA>; \lstrcatA
```

Dalje na adresi 004010AF pocinje loop koji se izvrsava dva puta i sluzi za menjanje prva cetiri slova ili broja iz volume labela.

```
004010AD  |.  B2 02             MOV DL,2
004010AF  |>  8305 5C224000>/ADD DWORD PTR DS:[40225C],1
004010B6  |.  8305 5D224000>|ADD DWORD PTR DS:[40225D],1
004010BD  |.  8305 5E224000>|ADD DWORD PTR DS:[40225E],1
004010C4  |.  8305 5F224000> |ADD DWORD PTR DS:[40225F],1
004010CB  |.  FECA             |DEC DL
004010CD  |.^ 75 E0            \JNZ SHORT abexcm5.004010AF
```

Idemo dalje i na adresi 004010D9 se ponovo spajaju dva stringa L2C-5781 i nista, stoga je rezultat uvek L2C-5781. Sa F8 se krecemo dalje i primecujemo da se ponovo spajaju dva stringa. Prvi je L2C-5781 a drugi je vec spojeni i promesani volume label na koji je "zalepljen" string 4562-ABEX, stoga je rezultat sledeci: L2C-5781volumelabel4562-ABEX, gde je volumelabel promesano ime particije na kojoj se nalazi. Ostaje samo da se na adresi 004010F7 uporede uneti serijski broj i ovaj dobijeni serijski broj pomocu funkcije lstrcmpiA. Ova funkcija vraca 0 ako su uneti serijski i tacan serijski broj isti a 1 ako su razliciti.

```
004010CF  |.  68 FD234000     PUSH abexcm5.004023FD          ; /StringToAdd = "L2C-5781"
004010D4  |.  68 00204000     PUSH abexcm5.00402000          ; |ConcatString = ""
004010D9  |.  E8 63000000     CALL <JMP.&KERNEL32.lstrcatA> ; \lstrcatA
004010DE  |.  68 5C224000     PUSH abexcm5.0040225C          ; /StringToAdd = ""
004010E3  |.  68 00204000     PUSH abexcm5.00402000          ; |ConcatString = ""
004010E8  |.  E8 54000000     CALL <JMP.&KERNEL32.lstrcatA> ; \lstrcatA
004010ED  |.  68 24234000     PUSH abexcm5.00402324           ; /String2 = ""
004010F2  |.  68 00204000     PUSH abexcm5.00402000           ; |String1 = ""
004010F7  |.  E8 51000000     CALL <JMP.&KERNEL32.lstrcmpiA>; \lstrcmpiA
004010FC  |.  83F8 00         CMP EAX,0
004010FF  |. /74 16           JE SHORT abexcm5.00401117
```

Na osnovu toga se na adresama 004010FC i 004010FF odlucuje da li je serijski broj ispravan ili ne.

Ovo je samo jedan od mogucih primera kako se koriste hardware-ske komponente u generisanju serijskiog broja i kako se tako dobijeni serijski broj uporedjuje sa nasim unetim laznim serijskim brojem. Cesti su primeri da programi koriste karakteristicne kljuceve koji su validni samo za jedan kompjuter. Ovakvi kljucevi se nazivaju computer IDovi i zavise od hardwareskih i softwareskih komponenti samog sistema. Na osnovu ovog broja ili niza slova se generise unikatni serijski broj validan samo za masinu na kojoj je computer ID jednak spisku komponenti u odnosu na koje se racuna. U ovom slucaju da bi serijski broj radio na svim masinama moraju se napraviti keygeneratori koji ce sa uneti computer ID generisati tacan serijski broj. Naravno moguce je napraviti i keygenerator koji ce vam otkriti tacan computer ID za vas kompjuter u slucaju da sam program ne daje takve informacije o samom sebi. Oba postupka su ista i bice obradjena u odeljku koji se bavi keygeneratorima.

**Vezba:**

Ako zelite mozete proveriti znanje steceno o trazenju serijskog broja na crackmeu vcrkme01.exe za ime cracker koji se takodje nalazi u folderu Cas04.

**Resenje:**

Ovde vam uopste necu pomoci posto je potrebno da malo modifikujete kod kako biste nasli pravi serijski broj. Moja pomoc ce se odnositi samo na jedan validan serijski broj koji moze da se unese, a to je:

```
name:          1racker
serial         1-R25357-50033
```

i na lokaciju na kojoj se racuna i proverava serijski broj, od 00401000 - 0040119B. Ovaj primer nema veze sa Computer IDom ali je odlican za vezbanje "fishinga". Obavezno ga resite sami!

# The Serials – VB & Olly

Kao sto smo videli moguce je naci serijski broj sa specijalizovanim programom za VB programe koji se zove Smart Check. Sada cu vam pokazati kako se sa istom takvom lakocom mogu pronaci serijski brojevi uz pomoc Ollya. Otvorite primer keygenme03b.exe pomocu Ollya. Kada ovo uradite pogledajte imena koja se nalaze kao importi (*videti The Serials - API Fishing*) u ovom exe fajlu. Najcesce koriscene funkcije za racunanje serijskih brojeva su

```
????????  .text   Import   MSVBVM60.__vbaCmpStr        <- Nema je u ovom fajlu
00401010  .text   Import   MSVBVM60.__vbaLenBstr
004010C8  .text   Import   MSVBVM60.__vbaStrMove
00401030  .text   Import   MSVBVM60.__vbaVarCmpGe
00401090  .text   Import   MSVBVM60.__vbaVarCmpLe
0040105C  .text   Import   MSVBVM60.__vbaVarTstEq
```

Posto se skoro sve one nalaze u ovom exe fajlu problem predstavlja stvar izbora. Naravno da mozemo da postavimo break-point na sve ove importe ali cemo uraditi sledece. Prvo cemo u crackme uneti neko ime i neki serijski broj, a onda cemo postaviti break-point samo na importe koji sluze za poredjenje vrednosti. Posto u ovom exe fajlu nemamo vbaStrCmp postavicemo break-point on every reference na vbaVatTstEq, vbaVarCmpGe i vbaVarCmpLe. Kada uradimo ovo pritisnucemo Check u crackmeu i naci cemo se ovde:

```
00402CDC   .  FF15 30104000          CALL DWORD PTR DS:[MSVBVM60.__vbaVarCmpGe]
00402CE2   .  8D4D BC                LEA ECX,DWORD PTR SS:[EBP-44]
00402CE5   .  50                     PUSH EAX
```

Ovo nam ne govori mnogo pa cemo pritiskati F9 sve dok ne dodjemo do poslednjeg break-pointa koji se izvrsi bas pred prikazivanje poruke o pogresnom serijskom broju.

```
00402DBA   .  FF15 5C104000          CALL DWORD PTR DS[MSVBVM60.__vbaVarTstEq]
00402DC0   .  8D4D B0                LEA ECX,DWORD PTR SS:[EBP-50]
00402DC3   .  8BF0                   MOV ESI,EAX
00402DC5   .  FF15 E0104000          CALL DWORD PTR DS:[ MSVBVM60.__vbaFreeObj]
```

Ako pogledamo malo ispod videcemo sledece:

```
00402DFA   .  C785 48FFFFFF>         MOV   DWORD   PTR   SS:[EBP-B8],keygenme.00402>;
UNICODE "Serial:"
...
00402E20   .  C785 58FFFFFF>         MOV   DWORD   PTR   SS:[EBP-A8],keygenme.00402>;
UNICODE "You have entered correct serial!"
...
00402E6D   .  C785 58FFFFFF>MOV DWORD PTR SS:[EBP-A8],keygenme.00402>;  UNICODE
"You have entered WRONG serial!"
```

Mozemo da udjemo u CALL vbaVarTstEq sa F7 kako bismo videli sta se to poredi. Kada dodjemo do adrese 66109845 (*razlikovace se na vasem kompjuteru*) videcemo u registrima sledece:

```
EDX 0014B85C UNICODE "AoC-03B-NZ3896"
EIP 66109845 MSVBVM60.66109845
```

Ali postoji i drugi nacin, a to je da predjemo preko CALLa ka vbaVarTstEq sa F8 i da na sledecoj adresi u EDX registru primetimo tacan serijski broj. Kao sto vidimo vrednosti sa kojom se poredi nas uneti serijski je AoC-03B-NZ3896 Na ovaj nacin se pronalaze sva moguca poredjenja laznog serijskog broja sa tacnom vrednoscu serijskog broja.

# The Serials – Patching

Ponekad je potrebno naterati program da misli da je svaki uneti serijski broj ispravan. Ova tema je vec dotaknuta na samom pocetku poglavlja ali cemo ovde tu pricu prosiriti sa novim cinjenicama i olaksati sebi u mnogo cemu posao. Primer za ovaj deo poglavlja se nalazi u folderu Cas2 a zove se patchme.exe. Otvoricemo ovaj program pomocu Ollya i potrazicemo string koji se pojavljuje na ekranu kada unesemo pogresan serijski broj. Ta poruka je "Bad Cracker" - bez navodnika, a naci cemo je na standardan nacin pomocu string referenca. Poruka se nalazi ovde:

```
004087C8  |> \68 18884000    PUSH patchme.00408818          ; /Text = "Bad Cracker"
```

a ako pogledamo malo gore videcemo i poruku o ispravnom seriskom broju.

```
004087B1  |.  68 0C884000    PUSH patchme.0040880C          ; /Text = "Cracked ok"
```

Primeticemo jedan kondicionalni skok iznad poruke o ispravnom serijskom broju.

```
004087A9  |.  E8 7EFDFFFF    CALL patchme.0040852C
004087AE  |.  48             DEC EAX
004087AF  |.  75 17          JNZ SHORT patchme.004087C8
```

Tom skoku prethodi jedan CALL od kojeg direktno zavisi da li ce ovaj skok biti izvrsen, a ovo znaci da je ovo mesto na kome se proverava uneti seriski broj, stoga cemo postaviti jedan break-point na taj CALL i pokrenucemo program. U polja za unos cemo uneti ap0x kao ime a kao serijski broj 111111, pa cemo pritisnuti Check. Naravno program je zastao na nasem break-pointu. Sa F7 uci cemo u njega i naci cemo se ovde:

```
0040852C  /$  55             PUSH EBP
0040852D  |.  8BEC           MOV EBP,ESP
0040852F  |.  81C4 E4FEFFFF  ADD ESP,-11C
00408535  |.  53             PUSH EBX
```

Ono sto znamo o ASM komandama pomoce nam da resimo ovaj problem. Ako pogledamo adrese 004087AE i 004087AF videcemo da se EAX smanjuje za jedan, a ako je nesto manje ili jednako necemu onda se skace na poruku o pogresnom serijskom broju. Ovo nesto je zero flag. Dakle ako je EAX jednak minus 1 odnosno FFFFFFFF onda ce se skociti na poruku o pogresnom serijskom broju. Da bismo ovo izbegli moramo da EAXu dodelimo takvu vrednost da kada se od nje oduzme jedan vrednost u EAXu bude veca od minus jedan. Naravno u EAX cemo smestiti najveci sledeci broj to jest jedan, tako da kad od njega oduzmemo jedan JNZ ne bude izvrsen jer je EAX posle oduzimanja jednak nula a ne minus jedan. Ovu izmenu koda cemo raditi unutar samog CALLa a ne pre CALLa jer ne znamo da li se ova funkcija poziva vise puta i odakle, tako da je najsigurnije da to uradimo u samom CALLu. Da bismo uradili ovo sto smo zamislili selektovacemo adresu 0040852C i duplim klikom izmenicemo njen sadrzaj u ovo:

```
0040852C    B8 01000000     MOV EAX,1
00408531    C3              RET
```

Izmenili smo ASM kod u ovo jer ovaj kod radi bas ono sto mi zelimo. On EAXu dodeljuje vrednost jedan i pomocu komande RET se vraca iz ovog CALLa, a samim tim ce svaka provera koja je prosledjena ovom CALLu uvek biti uspesna to jest za svaki uneti serijski broj i ime program ce pokazivati da je registrovan. Na isti nacin se resava ovaj isti problem ako se umesto DEC EAX komande nalazi TEST EAX,EAX komanda. Ovaj primer se cesto srece u

praksi a posledica je loseg nacina programiranja jer ovakav primer predstavlja jednostavan poziv funkciji koja vraca vrednosti True ili False u zavisnosti da li je uneti serijski broj ispravan ili ne. Imajte na umu da se ovakva vrsta zastite, bez obzira koliko ona bila slozena unutar koda CALLa, resava za svega par minuta. Dakle resenje ovog problema shvatite tako sto nikada nemojte da pisete funkcije koje ce vracati vrednosti True ili False pri proveri serijskog broja, rutine za proveru pisite drugacije.
Iako izgleda da smo sa ovim primerom zavrsili to nije tacno. Ovaj program kao i mnogi drugi belezi podatke o registraciji negde u sistemu. Mi cemo pronaci mesto gde se belezi ova informacija i zaobici cemo proveru tacnosti ove informacije.
Kao sto sto smo vec videli, jednostavnim patchovanjem resavamo problem funkcija koje proveravaju tacnost nekog serijskog broja. Ono sto se sada pitamo je da li smo zaobisli vazne provere koje se mogu nalaziti unutar samog CALLa. Pogledacemo malo detaljnije CALL i videcemo sledece stvari:

1) On generise fajl TheArtOfCracking.key u koji se najverovatnije zapisuje registracija.
2) Ovaj fajl ce biti smesten u Windows direktorijum tako da kada se sledeci put crackme startuje, ovi podaci ce se ucitati i proveriti
3) Postoji jedan zanimljiv skok:

```
004085DE    /0F85 B8000000  JNZ patchme.0040869C
```

Ovaj skok nas vodi dovde:

```
0040869C  |> \33DB          XOR EBX,EBX
0040869E  |>  33C0          XOR EAX,EAX
```

a sluzi za dodeljivanje vrednosti 0 EAXu i EBXu. Posto znamo da EAX na izlazu iz ovog CALLa mora biti jednak jedan, ovo znaci da se gornji skok ne sme izvrsiti. Ono sto nas trenutno buni je to sto ako se ne izvrsi gornji skok primeticemo da ce se sledeca dva reda uvek izvrsiti:

```
00408695  |.  BB 01000000     MOV EBX,1
0040869A  |. /EB 02           JMP SHORT patchme.0040869E
```

Hmmm, ovde imamo mali problem u EBX se smesta 1 a ne u EAX. Ako pogledamo odmah ispod prvog donjeg RETa videcemo odgovor na ovo pitanje:

```
004086C0  .  8BC3           MOV EAX,EBX
004086C2  .  5E             POP ESI
004086C3  .  5B             POP EBX
004086C4  .  8BE5           MOV ESP,EBP
004086C6  .  5D             POP EBP
004086C7  .  C3             RET
```

Aha na samom kraju se EAXu dodeljuje vrednost iz EBXa.

Ostaje nam samo da se postaramo da se skok sa adrese 004085DE nikada ne izvrsi jer ce tada oba uslova biti ispunjena. I fajl ce biti snimljen i EAX ce uvek biti jednako jedan. Stoga cemo samo NOPovati JUMP na adresi 004085DE i fajl ce biti snimljen sa bilo kojim unetim imenom i serijskim brojem. Ako uradimo trajni patch na ovoj adresi videcemo da ako ponovo startujemo ovaj crackeme on ce uvek biti registrovan sa bilo kojim unetim imenom i sa bilo kojim unetim serijskim brojem.

# The Serials – KeyFile

Cesto sretani problem prilikom reversinga su i takozvani keyfajlovi. Ovi fajlovi predstavljaju samo mesto gde su sacuvani podaci vezani za registraciju. Mi cemo uz pomoc Ollya i vaseg omiljenog Hex editora razbiti jednu ovakvu zastitu. Meta se zove CrackMe.exe a nalazi se u Cas3 folderu.

Najcesce se za citanje fajlova koristi API funkcija CreateFileA pa cemo postaviti break-point na nju. Ovo cemo uraditi na standardan nacin: U Ollyu cemo izabrati Executable modules -> View names -> CreateFileA -> Set breakpoint on every reference...

Posle ovoga cemo startovati program pomocu opcije Run, posle cega ce program zastati na sledecem mestu:

```
0040416A   .  E8 9DD0FFFF          CALL <JMP.&KERNEL32.CreateFileA>
```

Sa F8 cemo izvrsiti ovaj CALL i zavrsicemo ovde:

```
0040120C   $- FF25 04914200        JMP DWORD PTR DS:[<&KERNEL32.CreateFileA>
```

Uklonite ovaj break-point. Sada cemo pritiskati F8 onoliko puta koliko nam treba da se vratimo iz kernel32.dll. Pritiskajte F8 dok ne izvrsite prvu RET komandu, posle cega cemo se vratiti ovde:

```
0040416F   > /83F8 FF              CMP EAX,-1
00404172   . /74 29                JE SHORT CrackMe.0040419D
```

Ovaj CMP proverava da li postoji fajl na disku koji sadrzi podatke o registraciji programa. Kako se zove taj fajl??? Ova informacija je sigurno morala da bude prosledjena gornjoj CALL CREATEFILEA funkciji, jer ovaj API mora da zna koji fajl treba da otvori. Zbog ovoga cemo postaviti break-point na prvu PUSH komandu koja se prosledjuje tom CALLu. Postavicemo break-point ovde:

```
0040415A   > \6A 00                PUSH 0                   ; /hTemplateFile = NULL
```

Pritisnite F9 kako biste nastavili sa izvrsavanjem programa. Sada nazad u crackmeu pritisnite dugme Try Again i program ce zastati na novopostavljenom break-pointu. Sada cemo izvrsiti ovaj CALL sa F8 sve dok ne dodjemo do PUSH EAX komande. Kada dodjemo do nje sadrzaj EAXa ce biti ime fajla koji nam treba. Taj fajl je ctm_cm02.key.

Sada cemo napraviti jedan fajl sa sadrzajem jednog stringa, sa stringom ap0x (nula a ne O). Ovo mozete uraditi uz pomoc Notepada ili sa nekim Hex editorom. Pitanje je samo gde ce se to ovaj fajl nalaziti? Odgovor je jednostavan: On ce se nalaziti u istom direktorijumu ili bi PUSH EAX funkcija sadrzala celu putanju do ctm_cm02.key fajla.

Sada cemo ponovo pritisnuti dugme Try again u crackmeu i program ce zastati na PUSH EAX break-pointu. Polako cemo izvrsavati sve redove koda sa F8 i posmatracemo sta se desava. Ovde:

```
0040416F   > /83F8 FF              CMP EAX,-1
00404172   . /74 29                JE SHORT CrackMe.0040419D
```

se sada ne izvrsava JE skok pa cemo se posle izvrsavanja donje RET komande naci ovde:

```
00426592   .  E8 4DC1FDFF          CALL CrackMe.004026E4
...
004265AA   .  E8 F9C0FDFF          CALL CrackMe.004026A8
```

kada dodjemo do ovog drugog CALLa videcemo da EAX sada sadrzi broj 4 sto je duzina naseg stringa u key fajlu. Vidimo i koji red ispod da se EAX koristi za proveru da li je key fajl prazan ili ne.

```
004265B2   .  837D FC 00          CMP DWORD PTR SS:[EBP-4],0
...
004265B8   .  BA 64674200         MOV EDX,CrackMe.00426764
...
004265CD   > \817D FC 00000>      CMP DWORD PTR SS:[EBP-4],10000
```

Kao sto vidimo u fajlu mora biti nesto zabelezeno a to nesto mora biti krace od 0x10000h karaktera. Prvi put se u registrima pojavljuje adresa koja sadrzi nas uneti string na adresi 00426600.

```
00426600   .  8DB5 FCFFFEFF       LEA ESI,DWORD PTR SS:[EBP+FFFEFFFC]
```

Izvrsavajuci kod red po red dolazimo do sledece petlje:

```
00426616   > /8A1C16              MOV BL,BYTE PTR DS:[ESI+EDX]
00426619   . |84DB                TEST BL,BL
0042661B   . |74 29               JE SHORT CrackMe.00426646
0042661D   . |E8 16000000         CALL CrackMe.00426638
00426622   . |52                  PUSH EDX
00426623   . |F7E3                MUL EBX
00426625   . |5A                  POP EDX
00426626   . |35 326D5463         XOR EAX,63546D32
0042662B   . |FEC2                INC DL
0042662D   . |39CA                CMP EDX,ECX
0042662F   . |74 42               JE SHORT CrackMe.00426673
00426631   . |80FA FF             CMP DL,0FF
00426634   . |74 3D               JE SHORT CrackMe.00426673
00426636   .^\EB DE               JMP SHORT CrackMe.00426616
```

Analizom dolazimo do zakljucka da se na prvoj adresi ove petlje 00426616 u registar BL smesta hex vrednost svakog slova iz unetog stringa. Ovaj registar se koristi da bi se u EAX smestio neki broj. Mozda cak i pravi serijski broj! Primeticete da se u ovom loopu nalazi jedan CALL. Uci cemo u njega da vidimo sta se tu desava:

```
00426638  /$  57                  PUSH EDI
00426639  |.  8DBD F4FFFEFF       LEA EDI,DWORD PTR SS:[EBP+FFFEFFF4]
0042663F  |.  8B3F                MOV EDI,DWORD PTR DS:[EDI]
00426641  |.  881C17              MOV BYTE PTR DS:[EDI+EDX],BL
00426644  |.  5F                  POP EDI
00426645  \.  C3                  RET
```

Kao sto vidimo nista specijalno samo se slova iz stringa smestaju na neki duzi string. Posto skok:

```
0042661B   . /74 29               JE SHORT CrackMe.00426646
```

vodi van gornjeg loopa, postavicemo jedan break-point na adresu na koju on vodi. Postoji jos par skokova koji vode van tog loopa ali kako vidite ovde:

```
00426619   .  84DB                TEST BL,BL
```

taj skok ce se izvrsiti kada se iskoriste sva slova iz unetog stringa. Takodje cemo postaviti break-point na adresu na koju vode ostala dva skoka:

```
00426634   . /74 3D               JE SHORT CrackMe.00426673
```

Prodjimo vise puta kroz ovaj loop i videcemo da se skok ispod TEST BL,BL nikada nece izvrsiti. Hmmm... ovde nesto ne valja, jer ako se ne izvrsi ovaj skok mi cemo zavrsiti na delu koji samo prikazuje poruku o pogresnom serijskom broju na ekran. Sledi pitanje: Kako da BL bude nula?

Odgovor je jednostavan: U BL se smestaju hex vrednosti slova iz unetog stringa, jedan po jedan. JE skok ce se izvrsiti samo ako je BL jednak 0x00 stoga cemo pomocu Hex editora na kraj naseg stringa dodati 0x00 bajt. Posle ovoga cemo pritisnuti F9 kako bi smo nastavili sa izvrsavanje programa i ponovo cemo pritisnuti Try again u crackmeu. Kada zastanemo na PUSH 0 break-pointu pritisnucemo F9 2x kako bismo dosli do ovde:

```
00426646   > \E8 EDFFFFFF         CALL CrackMe.00426638
```

Pritisnucemo F8 4x dok ne dodjemo dovde:

```
0042664F   . 39D1                CMP ECX,EDX
```
Vrednosti koje se nalaze u ECXu i EDXu su:
**ECX = 5**
**EDX = 9**
Sta se ovde poredi? Duzina stringa ap0x + 0x00 sa 9. Ovo znaci da duzina stringa mora da bude 9. Sa Hex Editorom cemo dodati string 1234 na kraj key fajla. Ponovo cemo morati da pritisnemo F9 i da u crackmeu kliknemo na Try again dugme. Ponovo cemo doci na adresu 00426646 i sa F8 cemo izvrsavati red po red sve dok ne dodjemo dovde:
```
0042665C   > \3B0416            CMP EAX,DWORD PTR DS:[ESI+EDX]
0042665F   . 75 09               JNZ SHORT CrackMe.0042666A
00426661   . B8 00000000         MOV EAX,0
00426666   . 8907                MOV DWORD PTR DS:[EDI],EAX
00426668   . EB 10               JMP SHORT CrackMe.0042667A
0042666A   >  B8 01000000        MOV EAX,1
0042666F   . 8907                MOV DWORD PTR DS:[EDI],EAX
00426671   . EB 07               JMP SHORT CrackMe.0042667A
```
U prvom gornjem redu se poredi EAX = 69BB21A1 sa 34333231. Sta ovo znaci? Pogledajmo to ovako: 34 33 32 31 = 4 3 2 1, posto smo mi uneli kao string 1234 a ne 4321 vidimo da se brojevi okrecu. Ovo znaci da EAX sadrzi tacan serijski broj. EAX = 69BB21A1, samo morate da pazite kada ga unosite u fajl pomocu Hex Editora, ovaj broj se okrece pa cete uneti A1 21 BB 69.
Analizirajmo dalje... Kada su su ova dva broja jednaka EAX ce postati jednak 0 a ako nisu EAX ce postati jednak 1. Ovo se dole proverava i u zavisnosti od toga prikazuje se poruka ili o tacnom serijskom broju ili o pogresnom serijskom broju. Sada mozemo zatvoriti Olly i pomocu Hex editora sadrzaj fajla ctm_cm02.key promeniti u ovo:
**61 70 30 78 00 A1 21 BB 69  ap0x..!.i**
Sada mozete startovati crackme i videcete da je on registrovan. Dakle uspeli smo :)

**Dodatna analiza:**
Kao sto smo videli moramo da vodimo racuna o obliku key fajla. Ta forma izgleda ovako:
vase_ime 0x00 serijski duzine 4 bajta

**Napomena:**
Ako resavate problem sa kljucem nepoznatog oblika potrebno je da izvrsite detaljnu analizu dobijenih podataka. Bitno je sta se odakle cita i sa cime se poredi. Posto sam i ja prvi put reversovao ovaj primer ja sam to uradio ali je zbog duzine objasnjenja analiza problema svedena na minimalnu meru. Najbitnije je da posmatrate registre i da gledate gde se sta poredi.

# The Serials – KeyFile & Registry

Rec smo resili jedan crackme koji koristi keyfajlove za cuvanje i verifikaciju registracije. Ovaj put nas problem je dodatno otezan cinjenicom da nam je pored nepoznatog formata keyfajla nepoznat i potreban sadrzaj Registry baze. Pa prva stvar prvo, otvorite crackme Crackme3.exe koji se nalazi u folderu Cas03 kako bismo zapoceli sa reversingom.

Da bismo uopste poceli sa resavanjem ovog problema potrebno je da pronadjemo mesto gde pocinje algoritam za proveru tacnosti serijskog broja, a ovo cemo saznati tako sto cemo analizirati standardan poziv ka pravljenju dijaloga koji se nalazi ovde:

```
00401273  |.  6A 00           PUSH 0                                 ; /lParam = NULL
00401275  |.  68 93124000     PUSH Crackme3.00401293  ; |DlgProc = Crackme3.00401293
0040127A  |.  6A 00           PUSH 0                                 ; |hOwner = NULL
0040127C  |.  56              PUSH ESI                               ; |pTemplate
0040127D  |.  FF35 B0314000   PUSH DWORD PTR DS:[4031B0]             ; |hInst = NULL
00401283  |.  E8 6E050000     CALL DialogBoxIndirectParam; \DialogBoxIndirectParamA
```

Odavde vidimo da se MessageLoop koji obradjuje poruke koje mi saljemo nalazi na adresi 00401293 na kojoj se sigurno mora nalaziti i "link" ka kodu koji vrsi proveru tacnosti unetog serijskog broja. Iz ovog razloga odlazimo tamo i skrolujemo kroz kod sve dok ne ugledamo ovo:

```
00401361   .  E8 FC030000     CALL 00401762
00401366   .  68 39314000     PUSH 00403139              ; /Arg4 = 00403139
0040136B   .  68 D1124000     PUSH 004012D1              ; ASCII "Regkey"
00401370   .  68 AA124000     PUSH 004012AA              ; ASCII "Software\DFCG Crackme"
00401375   .  68 3D314000     PUSH 0040313D              ; |Arg1 = 0040313D
0040137A   .  E8 D1020000     CALL 00401650              ; \Crackme3.00401650
0040137F   .  84C0            TEST AL,AL
00401381   .  74 44           JE SHORT 004013C7
```

Ovaj kod sigurno predstavlja prvi deo provere tacnosti unetog serijskog broja, odnosno predstavlja deo koji se odnosi na verifikaciju stanja Registry baze. Pre nego sto udjemo u CALL na adresi 0040137A, videcemo sta se nalazi u CALLu koji prethodi Registry check CALLu. Dakle ulazimo u CALL na adresi 00401361 i vidimo sledeci kod:

```
00401782  |.  50              PUSH EAX
00401783  |.  6A 00           PUSH 0
00401785  |.  E8 3C000000     CALL <JMP.&kernel32.GetVolumeInformation>;
0040178A  |.  8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
0040178D  |.  0FC8            BSWAP EAX
0040178F  |.  05 47434644     ADD EAX,44464347
00401794  |.  68 C4314000     PUSH Crackme3.004031C4
00401799  |.  50              PUSH EAX
0040179A  |.  E8 CD000000     CALL Crackme3.0040186C
```

koji se koristi za dobijanje unikatne informacije o nasem kompjuteru na osnovu podataka koje vraca funkcija GetVolumeInformation. Ti podaci su uvek vezani za hard disk na kojem se fajl nalazi. Posto se racunanje Computer IDa zasniva na podatku koji je specifican za svaki kompjuter, odnosno u ovom slucaju svaki hard disk, rezultati koje ce vam program prikazivati, a koji se zasnivaju na ovom broju, drasticno ce se razlikovati od onih koje cu ja dobijati.

Posle izvrsenja poziva ka GetVolumeInformation APIju rezultat se smesta na vec alocirana mesta na koja pokazuje EBP. Posle ovoga EAX dobija vrednost parametra koji se nalazi na adresi EBP-4, a ova vrednost se prvo menja pomocu BSWAP komande, a onda se na nju dodaje 44464347h. Ova konacna

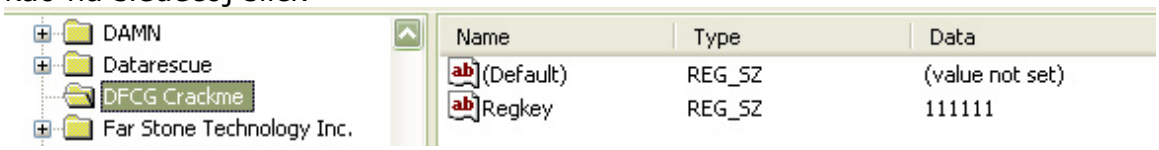
vrednost koju sadrzi EAX posle dodavanja heksadecimalne vrednosti na nju je nas Computer ID koji je u mom slucaju jednak 88A7B7A8. Ova vrednost se pre nego sto se vratimo iz ovog CALLa pretvara u string pomocu CALLa na adresi 0040179A i smesta se na adresu 004031C4. Konacno kada se posle izvrsenja RET komande vratimo iz ovog CALa mozemo da predjemo na analizu sledeceg CALLa na adresi 0040137A koji predstavlja citanje/proveru sadrzaja Registryja. Analiziracemo taj CALL da bismo videli kako to Registry utice na registraciju programa. Najzanimljiviji deo ovog CALLa je sledeci deo:

```
00401660  |.  50              PUSH EAX                 ; /pHandle
00401661  |.  6A 01           PUSH 1                   ; |Access = KEY_QUERY_VALUE
00401663  |.  6A 00           PUSH 0                   ; |Reserved = 0
00401665  |.  FF75 0C         PUSH DWORD [EBP+C]         Subkey = "Software\DFCG Crackme"
00401668  |.  68 01000080     PUSH 80000001            ; |hKey = HKEY_CURRENT_USER
0040166D  |.  E8 B8020000     CALL <JMP.&advapi32.RegOpenKeyExA>
```

a on nam otkriva sve sto nam treba da bismo savladali ovu Registry zastitu. Odavde znamo da je potrebno da u Registryju postoji kljuc pod glavnom granom HKEY_CURRENT_USER koji se zove DFCG Crackme a koji se nalazi pod kljucem Software. Dakle cela putanja do kljuca koji je potrebno da postoji je HKEY_CURRENT_USER\Software\DFCG Crackme. Ali ovo je samo registry kljuc, on sam po sebi ne sadrzi nikakvu vrednost. Organizacija Windows Registryja je takva da se kljucevi ponasaju kao folderi, a vrednosti u njima kao fajlovi. Kao i u Windowsu tako i u Registryju "fajlovi" mogu biti razlicitih tipova, gde je osnovni tip "fajla" string tip. Jedni problem je sto ne znamo ime "fajla", ali ovo cemo otkriti cim napravimo kljuc DFCG Crackme, jer ce nasa meta onda doci dovde:

```
0040167F  |.  FF75 14         PUSH DWORD PTR SS:[EBP+14]     ; /pBufSize
00401682  |.  FF75 08         PUSH DWORD PTR SS:[EBP+8]      ; |Buffer
00401685  |.  8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]   ; |
00401688  |.  50              PUSH EAX                       ; |pValueType
00401689  |.  6A 00           PUSH 0                         ; |Reserved = NULL
0040168B  |.  FF75 10         PUSH DWORD PTR SS:[EBP+10]     ; |ValueName = "Regkey"
0040168E  |.  FF75 F8         PUSH DWORD PTR SS:[EBP-8]      ; |hKey
00401691  |.  E8 9A020000     CALL <JMP.&advapi32.RegQueryValueExA>
```

Iz ovoga je jasno da je ime "fajla" Regkey stoga cemo dodati value nasem kljucu sa imenom Regkey i vrednoscu 111111. To bi trebalo da izgleda isto kao na sledecoj slici:




A ovo je i poslednja stvar koju moramo da uradimo kako bi se ovaj CALL uspesno izvrsio, dodeljujuci ALu vrednost 1, odnosno dozvoljavajuci sledecem CALLu da se izvrsi:

```
0040137F   .  84C0          TEST AL,AL
00401381   .  74 44         JE SHORT Crackme3.004013C7
00401383   .  E8 91000000   CALL Crackme3.00401419
```

Uci cemo u njega klikom na F7 i analiziracemo kod koji vidimo. Odmah primecujemo vrednost koju smo uneli u registry, kao i njeno prosledjivanje sledecem CALLu:

```
0040141C  |.  68 3D314000     PUSH Crackme3.0040313D           ;  ASCII "111111"
00401421  |.  E8 7A040000     CALL Crackme3.004018A0
00401426  |.  3C 00           CMP AL,0
00401428  |.  74 26           JE SHORT Crackme3.00401450
```

Ovo je ocigledno poredjenje duzine unetog stringa u registryju sa nulom, odnosno ovo je provera da li smo uneli neki string u registry. Dalje tracujemo kroz kod i primecujemo ovo:

```
0040142A  |.  8D35 C4314000  LEA ESI,DWORD PTR DS:[4031C4]
00401430  |.  8D3D 3D314000  LEA EDI,DWORD PTR DS:[40313D]
```

Odnosno u ESI se smesta adresa 004031C4 na kojoj se nalazi nas Computer ID u string obliku, dok se u EDI smesta pointer ka vrednosti koja se nalazi u Registryju. Posle ove dve komande sledi jedan loop koji se koristi za poredjenje ove dve vrednosti:

```
00401436  |> /8A06          /MOV AL,BYTE PTR DS:[ESI]
00401438  |. |84C0          |TEST AL,AL
0040143A  |. |74 0F         |JE SHORT Crackme3.0040144B
0040143C  |. |8A17          |MOV DL,BYTE PTR DS:[EDI]
0040143E  |. |80F2 74       |XOR DL,74
00401441  |. |38C2          |CMP DL,AL
00401443  |. |75 0B         |JNZ SHORT Crackme3.00401450
00401445  |. |46            |INC ESI
00401446  |. |47            |INC EDI
00401447  |. |84C0          |TEST AL,AL
00401449  |.^\75 EB         \JNZ SHORT Crackme3.00401436
```

Ocigledno je da se ovde porede vrednost iz registryja i Computer ID tako sto se svako slovo/broj iz registryja XORuje sa 74h i onda poredi sa slovom Computer IDa sa istim indexom (*pozicijom u stringu*). Posto imamo Computer ID mozemo da izracunamo tacan serisjki broj, za moj kompjuter, tako sto cemo XORovati svako slovo Computer IDa sa 74h. Kada ovo uradimo dobijamo rezultat LL5C6C5L. Dakle vrednost koja treba da se nalazi u registryju nije 11111 nego LL5C6C5L (*ovo je vrednost za moj kompjuter*).

Ako sada restartujemo metu u Ollyju, ispavimo sadrzaj registryja videcemo da ce AL registar biti jednak jedan na sledecoj adresi i da ce se sledeci CALL izvrsiti:

```
00401388   .  84C0          TEST AL,AL
0040138A   .  74 3B          JE SHORT Crackme3.004013C7
0040138C   .  E8 1C030000    CALL Crackme3.004016AD
```

Iz ovog razloga cemo uci u taj CALL. U njemu cemo, tracejuci, primetiti sledeci API poziv:

```
004016C3  |.  FF35 B0314000  PUSH DWORD PTR DS:[4031B0]          ; |hModule = 00400000
004016C9  |.  E8 DA000000    CALL <JMP.&kernel32.GetModuleFileNameA
```

koji je vratio ime i putanju do .exe fajla nase mete koja se ovde menja

```
004016CE  |.  8D35 CD314000  LEA ESI,DWORD PTR DS:[4031CD]
004016D4  |.  B3 69          MOV BL,69
004016D6  |.  885C30 FF      MOV BYTE PTR DS:[EAX+ESI-1],BL
004016DA  |.  885C30 FD      MOV BYTE PTR DS:[EAX+ESI-3],BL
004016DE  |.  80C3 05        ADD BL,5
004016E1  |.  885C30 FE      MOV BYTE PTR DS:[EAX+ESI-2],BL
```

i transformise u ime nase mete .ini. Iz ovoga saznajemo da je drugi deo ove zastite vezan za citanje podataka iz .ini fajla. Posle ove transformacije sledi:

```
004016E5  |.  68 CD314000    PUSH Crackme3.004031CD          ASCII "D:\...\Crackme3.ini"
004016EA  |.  E8 01020000    CALL Crackme3.004018F0            ; \Crackme3.004018F0
004016EF  |.  3C 00          CMP AL,0
004016F1  |.  74 68          JE SHORT Crackme3.0040175B
```

jednostavna provera da li fajl postoji na disku, odnosno na zadatoj putanji. Da bi smo izbegi restart programa mozemo napraviti ovaj fajl pre nego sto program sa izvrsavanjem stigne do te komande.

Tracujuci dalje primecujemo jos jedan zanimljiv API poziv:

```
004016FE  |.  E8 B7000000   CALL <JMP.&kernel32.GetPrivateProfileSec>;
```

Ovaj poziv vraca imena sectiona koji se nalazi u nasem .ini fajlu. E sada se postavlja pitanje sta su to sectioni u .ini fajlovima? Mozete konsultovati API dokumentaciju ali cete se sigurno iz iskustva setiti da postoje .ini fajlovi sa ovakvom strukturom:

```
[Section Name 1]
Value1 = "11111"
Value2 = "11111"

[Section Name 2]
Value1 = "11111"
Value2 = "11111"
....
```

Kao sto se iz ovog kratkog objasnjenja i vidi, struktura .ini fajlova je krajnje jednostavna. Posto nasa meta trazi ime sekcije u .ini fajlu mi cemo nas fajl modifikovati tako da sadrzi sledeci tekst:

```
[SecName1]
Value = "ap0x"
```

Posle ovoga cemo restartovati nasu metu i ponovo cemo uci u ovaj CALL. Posto nas .ini fajl postoji i postoji section u njemu imamo neku vrednost program ce uci u loop koji pocinje na adresi 0040170A. Ovde primecujemo jos jedan poziv ka APIju GetPrivateProfileSectionA. Sta se ovde desava?
Posto je meta dobila SVA imena sekcija sada prolazi kroz sekcije jednu po jednu, brojeci ih i brojeci vrednosti u njima. U sledecem loopu se broji koliko to vrednosti sadrzi svaka sekcija:

```
0040172B  |> /57              |/PUSH EDI                          ; /String
0040172C  |. |E8 BF000000     ||CALL <JMP.&kernel32.lstrlenA>     ; \lstrlenA
00401731  |. |03F8            ||ADD EDI,EAX
00401733  |. |47              ||INC EDI
00401734  |. |FEC2            |INC DL
00401736  |> |803F 00         | CMP BYTE PTR DS:[EDI],0
00401739  |.^\75 F0           |\JNZ SHORT Crackme3.0040172B
0040173B  |.  80FA 02         |CMP DL,2
```

Ovde se primecije jedan problem, naime EDX registar koji meta koristi za racunanje broja vrednosti u svakom sectionu se resetuje na NT sistemima od strane lstrlenA APIja pa je poredjenje CMP DL,2 besmisleno i nece se nikada izvrsiti. Bez obzira na ovaj mali bug, meta nam jasno saopstava da u svakoj sekiji mora postojati tacno dve vrednosti. Ali ostaje pitanje koliko to sekcija ima? Pogledajte sada ovaj deo koda:

```
00401749  |.  FEC3            |INC BL
0040174B  |>  803E 00         |CMP BYTE PTR DS:[ESI],0
0040174E  |.^ 75 BA           \JNZ SHORT Crackme3.0040170A
00401750  |.  80FB 02         CMP BL,2
00401753  |.  75 06           JNZ SHORT Crackme3.0040175B
```

Odavde se jasno vidi da se BL povecava za jedan prilikom vracanja imena svake sekcije. Posto BL mora da bude jednak 2, iz ovoga sledi da mora postojati tacno dve sekcije sa dve vrednosti u nasem .ini fajlu. Ispravicemo ovo tako da nas .ini fajl sada izgleda ovako:

```
[SecName1]
Value1 = "11111"
Value2 = "22222"

[SecName2]
Value1 = "33333"
Value2 = "44444"
```

Sada posto je ovo ispunjeno AL ce na sledecem mestu biti jednak jedan, zbog cega ce sledeci CALL biti izvrsen:

```
00401391  . 84C0            TEST AL,AL
00401393  . 74 32           JE SHORT Crackme3.004013C7
00401395  . E8 BC000000     CALL Crackme3.00401456
```
U ovom CALLu ce se dalje proveravati sadrzaj naseg .ini fajla i to prvobitno u proveri sekcija i vrednosti koje smo uneli u njega. Ta provera se nalazi ovde:
```
0040147B  |. 68 98124000     PUSH Crackme3.00401298          ; |Key = "Name"
00401480  |. 68 DA124000     PUSH Crackme3.004012DA          ; |Section = "User Data"
00401485  |. E8 36030000     CALL <JMP.&kernel32.GetPrivateProfileStr>;
0040148A  |. 84C0            TEST AL,AL
0040148C  |. 0F84 B7010000   JE Crackme3.00401649
```
Ova procedura jednostavno pristupa sekciji User Data i iz nje iscitava vrednost kljuca Name. Ako pogledamo malo kod koji se nalazi malo dole:
```
0040149F  |. 68 E6124000     PUSH Crackme3.004012E6          ; |Key = "Serial"
004014A4  |. 68 DA124000     PUSH Crackme3.004012DA          ; |Section = "User Data"
004014A9  |. E8 12030000     CALL <JMP.&kernel32.GetPrivateProfileStr>;
```
Iz ovoga se vidi i naziv druge vrednosti koja je Serial. Dakle ponovo moramo da izmenimo sadrzaj .ini fajla:
```
[User Data]
Name = "ap0x"
Serial = "111111"

[SecName2]
Value1 = "33333"
Value2 = "44444"
```
Kada ovo uradimo ponovo cemo restartovati metu u Ollyju i traceovanjem kroz kod cemo stici dovde:
```
004014E3  |. 50              PUSH EAX                        String2 = "111111"
004014E4  |. 8D45 A0         LEA EAX,DWORD PTR SS:[EBP-60]
004014E7  |. 50              PUSH EAX                        String1 = "ap0x-88A7B7A8"
004014E8  |. E8 FD020000     CALL <JMP.&kernel32.lstrcmpA>
```
Dakle sada znamo i koje vrednosti sekcija User Data u nasem .ini fajlu mora da ima da bi nas program bio registrovan. Imajte na umu da je drugi deo serijskog broja specifican za moj kompjuter posto je ocigledno jednak Computer IDu. Ali da ne bismo restartovali Olly mnogo vise puta nego sto je stvarno potrebno pogledacemo sta nas dalje ceka u ovoj proveri:
```
00401502  |. 68 9F124000     PUSH Crackme3.0040129F          ; |Key = "SecData1"
00401507  |. 68 C2124000     PUSH Crackme3.004012C2          ; |Section = "Secrete Data"
0040150C  |. E8 AF020000     CALL <JMP.&kernel32.GetPrivateProfileStr>;
```
i sledeci deo provere drugog sectiona .ini fajla:
```
0040156E  |. 68 EF124000     PUSH Crackme3.004012EF          ; |Key = "SecData2"
00401573  |. 68 C2124000     PUSH Crackme3.004012C2          ; |Section = "Secrete Data"
00401578  |. E8 43020000     CALL <JMP.&kernel32.GetPrivateProfileStr>;
```
Dakle ponovo moramo da izmenimo nas .ini fajl, ovaj put on izgleda ovako:
```
[User Data]
Name = "ap0x"
Serial = "ap0x-88A7B7A8"

[Secrete Data]
SecData1 = "12345678"
SecData2 = "12345678"
```
Kada ovo uradimo mozemo ponovo da restartujemo Olly i da tracujemo sve dok ne dodjemo do ucitavanja prve vrednosti iz druge "secret" sekcije. Kada dodjemo dovde:
```
00401511  |. 3C 08           CMP AL,8
00401513  |. 0F85 30010000   JNZ Crackme3.00401649
```
shvatate zasto su vrednosti SecData1 i SecData2 promenjene u osmocifrene brojeve. Posle ovoga sledi provera tacnosti unetog podatka za SecData1 u .ini fajlu, pomocu sledeceg dela koda:

```
0040151C  |.  50              PUSH EAX                 ;  Pretvaranje SecData1 u Hex [EAX]
0040151D  |.  E8 FA020000     CALL Crackme3.0040181C
00401522  |.  33C9            XOR ECX,ECX
00401524  |>  F6C1 01         /TEST CL,1
00401527  |.  74 04           |JE SHORT Crackme3.0040152D
00401529  |.  34 74           |XOR AL,74
0040152B  |.  EB 02           |JMP SHORT Crackme3.0040152F
0040152D  |>  34 79           |XOR AL,79
0040152F  |>  FEC1            |INC CL
00401531  |.  80F9 04         |CMP CL,4
00401534  |.  74 04           |JE SHORT Crackme3.0040153A
00401536  |.  D1C0            |ROL EAX,1
00401538  |.^ EB EA           \JMP SHORT Crackme3.00401524
0040153A  |>  8985 5CFFFFFF   MOV DWORD PTR SS:[EBP-A4],EAX
00401540  |.  8B1D 4D324000   MOV EBX,DWORD PTR DS:[40324D]
00401546  |.  339D 5CFFFFFF   XOR EBX,DWORD PTR SS:[EBP-A4]
0040154C  |.  68 C4314000     PUSH Crackme3.004031C4           ;  ASCII "88A7B7A8"
00401551  |.  E8 C6020000     CALL Crackme3.0040181C
00401556  |.  3BC3            CMP EAX,EBX
00401558  |.  0F85 EB000000   JNZ Crackme3.00401649
```

Loop koji sam izdvojio sluzi za proracunavanje vrednosti na osnovu unetog podatka u .ini fajl. Na osnovu ovoga se racuna broj koji se dodatno menja XORovanjem sa konstantom koja se nalazi na adresi 004324D. Dakle ova jednacina izgleda ovako SecData1 ^ algoritam XOR const = Computer IDu.
Sada jos samo treba da reversujemo ovaj algoritam. Ako tracujemo kroz njega primeticemo da ce se on bez obzira na SecData1 sadrzaj uvek ponasati isto, dakle imacemo:

```
XOR AL,79
ROL EAX,1
XOR AL,74
ROL EAX,1
XOR AL,79
ROL EAX,1
XOR AL,74
ROL EAX,1
XOR EAX, 78307061
```

Dakle da bismo izracunali vrednost SecData1 moramo da reversujemo ovaj algoritam krecuci od kraja, od Computer IDa. Dakle kada izracunamo 88A7B7A8 xor 78307061 (*const*) krecemo sa reversingom algoritma, odnosno propustamo vrednost F097C7C9 kroz reversovan algoritam:

```
ROR EAX,1
XOR AL,74
ROR EAX,1
XOR AL,79
ROR EAX,1
XOR AL,74
ROR EAX,1
XOR AL,79
```

i kao rezultat toga dobijamo vrednost koju bi SecData1 trebalo da ima. Da ovo ne bismo racunali rucno napisao sam program koji radi bas ovo. Dakle samo treba da unesemo F097C7C9 u ReCrackme3.exe i da dobijeni rezultat unesemo u .ini fajl. Kada ovo uradimo i unesemo vrednost DE12F88F u nas .ini fajl videcemo da je rezultat algoritma SecData1 ^ algoritam XOR const jednak 88A7B681 a ne 88A7B7A8. Odakle ova greska? Iz razloga sto kod ROL/ROR komande moze da dodje do gubitka podataka mora se uraditi ponovno racunanje pocetne SecData1 vrednosti na osnovu prethodnog rezultata. Dakle XORovacemo 88A7B681 sa 78307061 i dobijenu vrednost cemo uneti u moj program. Rezultat ovoga ce ovaj put biti FE12F8AA. I sada

imamo pravu vrednost sa SecData1 tako da nam ostaje jos samo da resimo poslednji deo ove mete, odnosno da pronadjemo poslednju vrednost SecData2. Ona direktno zavisi od dva uneta parametara: imena i SecData1. Ovo se vidi na sledecem isecku koda:

```
004015B5  |> \8DB5 60FFFFFF LEA ESI,DWORD PTR SS:[EBP-A0]       ; Ucitaj SecData2
004015BB  |. 8D3D 4D324000  LEA EDI,DWORD PTR DS:[40324D]       ; Ucitaj ime
004015C1  |. 33C0           XOR EAX,EAX
004015C3  |. 33D2           XOR EDX,EDX
004015C5  |> 8A17           /MOV DL,BYTE PTR DS:[EDI]          /* Loop1
004015C7  |. 84D2           |TEST DL,DL
004015C9  |. 74 0A          |JE SHORT Crackme3.004015D5
004015CB  |. 8A1E           |MOV BL,BYTE PTR DS:[ESI]
004015CD  |. 03D3           |ADD EDX,EBX
004015CF  |. 03C2           |ADD EAX,EDX
004015D1  |. 46             |INC ESI
004015D2  |. 47             |INC EDI
004015D3  |.^ EB F0         \JMP SHORT Crackme3.004015C5       */
004015D5  |> 8985 58FFFFFF  MOV DWORD PTR SS:[EBP-A8],EAX
004015DB  |. 33C0           XOR EAX,EAX
004015DD  |. 33C9           XOR ECX,ECX
004015DF  |. 33D2           XOR EDX,EDX
004015E1  |. B1 08          MOV CL,8
004015E3  |. 8D75 80        LEA ESI,DWORD PTR SS:[EBP-80]       ; Ucitaj SecData1
004015E6  |> 8A16           /MOV DL,BYTE PTR DS:[ESI]          /* Loop 2
004015E8  |. 84C9           |TEST CL,CL
004015EA  |. 74 06          |JE SHORT Crackme3.004015F2
004015EC  |. 03C2           |ADD EAX,EDX
004015EE  |. 46             |INC ESI
004015EF  |. 49             |DEC ECX
004015F0  |.^ EB F4         \JMP SHORT Crackme3.004015E6       */
004015F2  |> 0385 58FFFFFF  ADD EAX,DWORD PTR SS:[EBP-A8]
```

Prvi loop radi sabiranje vrednosti koje imaju karakteri iz imena i karakteri od SecData2 podatka. Ovaj loop se ponavlja onoliko puta koliko nase uneto ime ima slova. Rezultat ovog loopa se privremeno snima na adresi EBP-A8, posle cega usledjuje drugi loop koji samo sabira vrednosti koje imaju slova iz podatka SecData1. Posle ovoga se na vrednost EAXa koja je postavljena u drugom loopu dodaje vrednost koja je privremeno snimljena na EBP-A8.

Kada se ove vrednosti izracunaju sledi deo koda koji proverava tacnost unetih podataka i on izgleda bas ovako:

```
004015F8  |. 33C9           XOR ECX,ECX
004015FA  |. 66:B9 5A00     MOV CX,5A
004015FE  |. 66:F7F1        DIV CX
00401601  |. 8915 6D324000  MOV DWORD PTR DS:[40326D],EDX
00401607  |. D905 94204000  FLD DWORD PTR DS:[402094]          ; 24.010 - Konstanta
0040160D  |. D835 88204000  FDIV DWORD PTR DS:[402088]         ; 100.000 - Konstanta
00401613  |. DA05 6D324000  FIADD DWORD PTR DS:[40326D]        ; Ostatak pri deljenju
00401619  |. D80D 8C204000  FMUL DWORD PTR DS:[40208C]         ; 3.141593 - Pi
0040161F  |. D835 90204000  FDIV DWORD PTR DS:[402090]         ; 180.000
00401625  |. D9FF           FCOS
00401627  |. D9E1           FABS
00401629  |. D80D 84204000  FMUL DWORD PTR DS:[402084]         ; 7479.00 - Konstanta
0040162F  |. D80D 88204000  FMUL DWORD PTR DS:[402088]         ; 100.000 - Konstanta
00401635  |. D9FC           FRNDINT                            ; Zaokruzivanje
00401637  |. D815 80204000  FCOM DWORD PTR DS:[402080]         ; Poredjenje sa 313233
0040163D  |. 9B             WAIT
0040163E  |. DFE0           FSTSW AX
00401640  |. 9E             SAHF
00401641  |. 75 06          JNZ SHORT Crackme3.00401649
```

Ocigledno je da nam ovde treba bar osnovno poznavanje trigonometrije kako bismo zavrsili sa resavanjem ovog crackmea. Ali ne zurimo toliko, da vidimo sta se prvo desava sa izracunatim sadrzajem EAXa.

Dakle prvo se ECX resetuje na nulu, pa mu se dodeljuje konstantna vrednost 5Ah, posle cega se EAX deli sa ECX tako da se ostatak pri deljenju smesta u EDX. Ovaj ostatak pri deljenju se smesta na adresi 0040326D odakle ce kasnije biti koristen za proveru tacnosti serijskog broja. Sada trebamo videti kako se FPU komande koriste za proveru tacnosti unetih podataka.
Analizirajmo taj kod kako bismo dosli do algoritma za proveru:

```
Round(abs(cos((((24.010 / 100.000) + EDX) * 3.141593) / 180.000)) * 7479 * 100) == 313233
```

Kao sto se vidi iz gornje jednacine samo je EDX nepoznata a ona se racuna na osnovu ostatka pri deljenju. Da bismo izracunali vrednosti koje EDX moze da ima napisacemo jednostavan Delphi program:

```
var
 edx,rez:integer;
begin
  writeln('EDX moze biti jednak:');
 for edx := 1 to 255 do begin
  rez := Round(abs(cos((((24.010 / 100.000) + EDX) * 3.141593) / 180.000)) * 7479 * 100);
  if rez = 313233 then writeln(edx);
 end;
  readln(rez);
end.
```

Ovaj program ce nam pokazati da EDX moze biti jednak samo 245 ili 65. Na osnovu ovoga moramo da dobijemo koliki je zbir SecData1 + SecData2 + Ime. Ovo cemo dobiti na osnovu sledece jednacine:

**5Ah * x + 41h = y**

Ovde je x proizvoljan mnozilac i za njega mozemo izabrati bilo koji broj tako da on bude priblizno jednak zbiru SecData1 + SecData2 + Ime. Posto su nam poznati podaci Ime i SecData1 i njih mozemo uvrstiti u jednacinu:

**(5Ah * x + 41h) – Ime – SecData1 = SecData2**

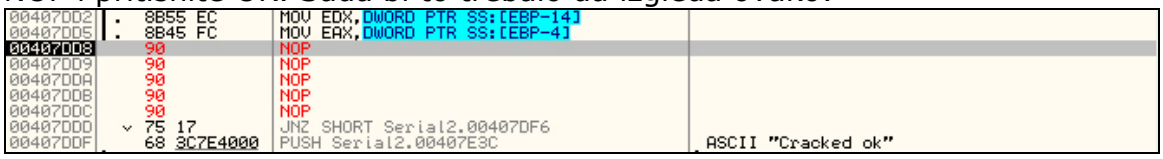
Dakle sada znamo tacno kako da dobijemo zbir koji imaju sva slova iz serijskog broja SecData2, ali sta posle toga? Pa posto SecData2 moze imati 8 ili vise slova jednostavno cemo podeliti zbir slova SecData2 sa 8, zaokruzicemo rezultat i odredicemo slovo koje smo dobili za vrednost koju smo izracunali. Ako rezultat zbira ovog novog slova bude manji ili veci od zbira koji SecData2 mora da ima jednostavno cemo izabrati neko dodatno slovo, ili cemo neko slovo oduzeti a dodati novo, tako da zbir slova bude jednak broju koji smo izracunali za SecData2.
To je to ceo algoritam ove mete je detaljno analiziran i sada sami mozete bez ikakvih problema napraviti keygenerator za nju ili mozete da pronadjete tacan serijski broj za vas kompjuter. Ovo objasnjenje je dugacko iz razloga sto se provera tacnosti serijskog broja zasniva na vecem broju parametara, ali bez obzira na to smo uspesno analizirali ovu metu i matematicki izracunali tacan serijski broj za nas kompjuter. Sada shvatate zasto je vazno znati bar osnove trigonometrije :)

# 04 Making KeyGens

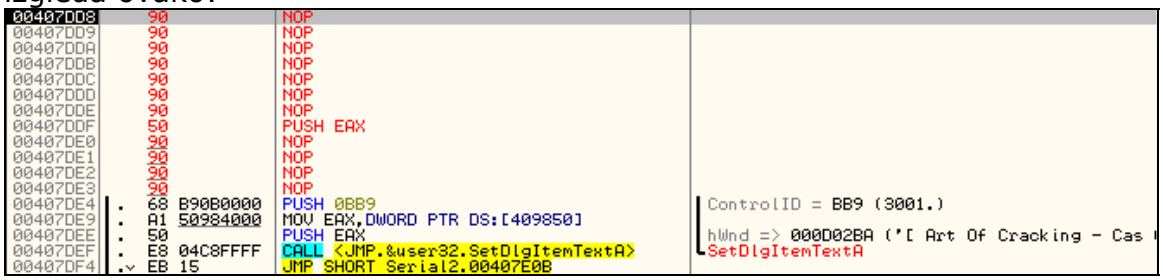
U proslom poglavlju smo naucili kako da pronadjemo mesto na kome se proverava serijski broj i kako da saznamo gde se to racuna tacan serijski broj. U ovom poglavlju cemo nauciti kako da iskoristimo ovo znanje i da napravimo keygeneratore za razlicite programe. Za ovo poglavlje je preporucljivo znanje nekog od sledeca tri programska jezika, morate znati ili C++ ili Visual Basic ili Delphi. Posto su ova tri programska jezika relativno laka za ucenje i razumevanje samo oni ce biti obradjivani u knjizi. Naravno postoji i deo za one koji ne znaju ni jedan od ova tri programska jezika, a nalazi se na samom pocetku ovog poglavlja i on ce vas nauciti kako da izmenite kod programa i da pretvorite njega samog u svoj keygenerator.

# KeyGen – Ripping #1

Poglavlje o pravljenju keygeneratora je podeljeno u vise delova. Prvi deo ce vas nauciti kako da izmenite kod u nekoj "meti" tako da ona sama postane svoj keygenerator, a ostali kako da u nekom drugom programskom jeziku napisete keygen."Meta" ce biti slicna kao na pocetku proslog poglavlja. Ucitajte program …\Casovi\Cas4\Serial2.exe u Olly. Kao sto smo u proslom poglavlju utvrdili, serijski broj se generise u petlji koja pocinje na adresi 00407D83. Postavite break-point odmah ispod ove petlje, postavite break-point na adresu 00407DB2. Sa F9 pokrenite program i unesite bilo koje ime. Zatim pritisnite Check i program ce zastati na adresi 00407DB2. Idite polako kroz program sa F8 sve dok ne dodjete do adrese 00407DD8 kada ce se u EAXu prikazati tacan serijski broj za uneto ime. Ako izvrsite CALL na toj adresi iz EAXa ce nestati tacan serijski broj, a to ne zelimo. Stoga selektujte taj CALL i pritisnite space da biste ga izmenili. U novootvoreni dijalog unesite NOP i pritisnite OK. Sada bi to trebalo da izgleda ovako:




Posto ni skok na adresi 00407DDD ne treba nikada da se izvrsi NOPovacemo i njega. Ako bismo stvari ostavili kao sto sada jesu program bi stalno na ekranu prikazivao poruku o tacnom serijskom broju ("Cracked OK"), ali ne i tacan serijski broj. Zapamtite da u EAXu sada imate tacan serijski broj i da samo treba da ga prikazete na ekran. Iskoristicemo cinjenicu da se string "Cracked OK" prikazuje na ekranu i samo cemo PUSH komandu malo modifikovati tako da umesto tog stringa prikazuje sadrzaj iz EAXa. Ta izmena izgleda ovako:




Umesto PUSH 00407E3C (*gde je ovaj hex broj adresa na kojoj se nalazi string "Cracked OK"*) stavicemo jednostavno PUSH EAX i program ce umesto stringa prikazivati sadrzaj EAXa na ekranu kada sa izvrsavanjem programa dodje do adrese 00407DEF to jest do windows api funkcije SetDlgItemTextA. Ako zelite da snimite ove promene kliknite desnim dugmetom na modifikovani deo koda i pritisnite ***Copy to executable -> All modifications -> Copy all*** i samo ga snimite pod nekim drugim imenom. Na ovaj jednostavan nacin mozemo dodavati ili modifikovati postojece funkcije u exe fajlu. Imajte na umu da ovo nije pravi keygenerator posto nismo saznali kako radi algoritam za generisanje pravog serijskog broja. Bez obzira na sve napravili smo program koji moze da prikaze tacan serijski broj za bilo koje uneto ime.

# KeyGen – Ripping #2

Vec smo videli da je tehnika ubacivanja koda u bilo koju metu veoma pogodna jer ne moramo da razumemo algoritam da bismo napravili keygenerator. Ovaj postupak cemo primeniti na metu in-keygen.exe.

Otvoricemo ovu metu pomocu Ollyja i pronaci cemo karakteristicne stringove koji se odnose na poruke o tacno/netacno unetom serijskom broju. Posto su nam u ovom fajlu samo ovi stringovi vazni:

```
Text strings referenced in in-keyge:.text
Address     Disassembly                      Text string
00401103    PUSH in-keyge.00403078           ASCII "Yup"
00401108    PUSH in-keyge.0040307C           ASCII "Good Job!"
00401118    PUSH in-keyge.00403072           ASCII "Error"
0040111D    PUSH in-keyge.0040309D           ASCII "Nope. Try Again."
```

Izabracemo string Good Job! i duplim klikom na njega dolazimo do mesta sa koga se poziva ovaj string, to jest naci cemo se ovde:

```
00401108  |.  68 7C304000     PUSH in-keyge.0040307C           ; |Text = "Good Job!"
```

Posto se ocigledno odmah iznad ovog prikazivanja poruke o tacnom serijskom broju proverava da li je serijski broj ispravan nase izmene cemo uraditi na sledecem delu koda:

```
004010EE  |.  68 E2304000     PUSH in-keyge.004030E2           ; /String2 = "111111"
004010F3  |.  68 C2304000     PUSH in-keyge.004030C2           ; |String1 = "bq1y"
004010F8  |.  E8 9B000000     CALL <JMP.&kernel32.lstrcmpA>    ; \lstrcmpA
004010FD  |.  0BC0            OR EAX,EAX
004010FF  |.  75 15           JNZ SHORT in-keyge.00401116
```

Naravno ovakav prikaz (*vidimo i stringove koji se porede*) cemo dobiti tek kada postavimo break-point na adresu 004010EE i u nasu metu kao ime unesemo ap0x a kao serijski broj 111111.

Kao sto vidimo APIju lstrcmpA se prosledjuju dva parametra, pravi serijski broj i nas uneti serijski broj. Posle izvrsavanja ovog APIja EAX ce sadrzati rezultat poredjenja. Ovaj rezultat ce biti 1 ako su uneti i tacan serijski broj jednaki ili 0 ako se ove dve vrednosti razlikuju. Ovde se jasno vidi razlika izmedju ovog i prethodnog primera. Za razliku od proslog crackmea ovde registri ne pokazuju direktno ka serijskom broju, to jest ne sadrze adresu na kojoj se nalazi tacan serijski broj zbog cega sada ne mozemo da koristimo PUSH EAX (*ili bilo koji drugi registar*) komandu. Sada moramo da izmenimo kod tako da ovaj kod:

```
00401101  |.  6A 00           PUSH 0                           ; /Style = MB_OK
00401103  |.  68 78304000     PUSH in-keyge.00403078           ; |Title = "Yup"
00401108  |.  68 7C304000     PUSH in-keyge.0040307C           ; |Text = "Good Job!"
0040110D  |.  6A 00           PUSH 0                           ; |hOwner = NULL
0040110F  |.  E8 66000000     CALL <JMP.&user32.MessageBoxA>   ; \MessageBoxA
```

sada umesto Good Job! prikazuje tacan serijski broj. Naravno posto je i Good Job! string i on ima adresu na kojoj se nalazi, a ta adresa je 0040307C (*PUSH 0040307C*). Posto se i tacan serijski broj nalazi na odredjenoj, staticnoj, adresi mozemo da izmenimo parametar PUSH 0040307C tako da umesto ka stringu Good Job! pokazuje na tacan serijski broj. Dakle izmenicemo PUSH 0040307C u PUSH 004030C2 (*videti adrese od 004010EE do 004010F8*) i MessageBox ce uvek prikazivati tacan serijski broj. Naravno ovo nije kraj naseg posla, ostaje nam jos samo da izmenimo JNZ skok pre prikazivanja poruke o tacnom serijskom broju, na adresi 004010FF, u NOP kako bi se nasa nova poruka koja sadrzi tacan serijski broj uvek prikazivala.

# KeyGen – Beginning #1

Videli smo da mete mogu da se modifikuju da bi prikazale tacan serijski broj. Ali ovo nije smisao pravih keygeneratora, pravi smisao ovih programa se ogleda u rekonstrukciji algoritma koji se koristi za racunanje tacnog serijskog broja. "Meta" ce za pocetak biti veoma jednostavna. Ucitajte program …\Casovi\Cas4\AD_CM#2.exe u Olly. Ovaj prvi primer je izuzetno lak i tu je da vam pokaze osnovu keygeneratora.

Otvorimo metu u Ollyju, startujte je i u nju unesite sledece podatke: ime: *cracker* i serial: *111111*. Potrazicemo stringove koji se pojavljuju u samoj meti klikom na Search for -> All referenced text strings... i videcemo da je ovaj string jako zanimljiv:

```
Text strings referenced in AD_CM#2:.text
Address          Disassembly                          Text string
00401167         PUSH AD_CM#2.00403027                ASCII "Yeah, you did it!"
```

Kada kliknemo dva puta na ovaj string naci cemo se ovde:

```
00401160  |.  6A 40            PUSH 40                      ;Style = MB_OK
00401162  |.  68 12304000      PUSH AD_CM#2.00403012        ;Title = "ArturDents CrackMe#2"
00401167  |.  68 27304000      PUSH AD_CM#2.00403027        ;Text = "Yeah, you did it!"
0040116C  |.  FF75 08          PUSH DWORD PTR SS:[EBP+8];hOwner
0040116F  |.  E8 1C000000      CALL <MessageBoxA>           ;MessageBoxA
```

Posto se ovde prikazuje poruka o tacnom serijskom broju ovo znaci da se negde iznad poredi i racuna serijski broj. Ovo se desava bas ovde:

```
00401154  |> /8A10             /MOV DL,BYTE PTR DS:[EAX]
00401156  |. |2AD1             |SUB DL,CL
00401158  |. |3813             |CMP BYTE PTR DS:[EBX],DL
0040115A  |. |75 18            |JNZ SHORT AD_CM#2.00401174
0040115C  |. |40               |INC EAX
0040115D  |. |43               |INC EBX
0040115E  |.^\E2 F4            \LOOPD SHORT AD_CM#2.00401154
```

Da bismo shvatili kako se racuna serijski broj stavicemo jedan break-point na adresu 00401154 i ponovo cemo u meti pritisnuti dugme "Check it baby!". Posle ovoga ce program normalno zastati na nasem break-pointu. Sada sledi najbitniji deo kod pravljenja keygeneratora. Sledi pazljiv pregled koda koji se izvrsava i nadgledanje sadrzaja registara kako bismo videli kako to uneto ime utice na racunanje serijskog broja. Videcemo da se u registrima nalaze sledece vrednosti:

```
EAX 00403080 ASCII "cracker"          <- Uneto ime
ECX 00000007                          <- Duzina unetog imena
EDX 00140608
EBX 00403280 ASCII "111111"           <- Uneti serijski broj
ESP 0012FB24
EBP 0012FB24
ESI 00000007
EDI 0012FBA4
EIP 00401154 AD_CM#2.00401154
```

Kao sto vidimo u registar DL se na adresi 00401154 stavlja prvo slovo unetog imena. Na sledecoj adresi se od registra DL oduzima registar CL koji sadrzi duzinu unetog imena.

Zanimljiva nam je sledeca adresa, adresa 00401158, jer se na njoj poredi prvo slovo unetog serijskog broja i sadrzaj registra DL. Ako ove dve vrednosti nisu jednake onda ce meta skociti na adresu 00401154, to jest tacno ispod poruke o tacno unetom serijskom broju. Iz ovog razloga cemo dva puta kliknuti na sledeci JNZ skok i unecemo novu komandu, unecemo

NOP. Ovo radimo iz razloga zato sto zelimo da vidimo kako ce se program ponasati ako su unete vrednosti iste. Posle ovog memorijskog patcha program ce izgledati ovako:

```
00401154  |> /8A10          /MOV DL,BYTE PTR DS:[EAX]
00401156  |. |2AD1          |SUB DL,CL
00401158  |. |3813          |CMP BYTE PTR DS:[EBX],DL
0040115A    |90            NOP
0040115B    |90            NOP
0040115C  |. |40          |INC EAX
0040115D  |. |43          |INC EBX
0040115E  |.^\E2 F4        \LOOPD SHORT AD_CM#2.00401154
```

Kao sto vidimo ako nastavimo sa izvrsavanjem programa sa F8 povecace se i EAX i EBX posle cega ce registri izgledati ovako:

```
EAX 00403081 ASCII "racker"
ECX 00000007
EDX 0014065C
EBX 00403281 ASCII "11111"
ESP 0012FB24
EBP 0012FB24
ESI 00000007
EDI 0012FBA4
EIP 0040115E AD_CM#2.0040115E
```

Ovo samo znaci da ce se u sledecem prolazu ovog loopa program porediti sledeci broj unetog serijskog broja i sledeci broj unetog serijskog broja. Dakle vidimo da ce se ovaj loop izvrsavati onoliki broj puta koliko ime ima karaktera. Svaki put kada je broj prolaza manji od broja karaktera program ce se vratiti sa izvrsavanjem na adresu 00401154, i ovo ce raditi sve dok broj prolaza ne bude jednak duzini unetog imena. Dok se ovo ne ispuni ECX registar se smanjuje za 1 i stoga se i CL smanjuje za 1.

Pre nego sto pocnemo da pravimo keygenerator potrebno je samo da pogledamo kod iznad ove provere, da bismo videli da li program ima neke zahteve oko unete duzine serijskog broja i videcemo sledece:

```
00401117  |.  83FE 05       CMP ESI,5
0040111A  |.  7D 18         JGE SHORT AD_CM#2.00401134
```

Kao sto vidimo ispod ovoga se nalazi poruka da duzina imena mora da bude najmanje pet karaktera. Dakle imamo sledeci algoritam:

```
radi petlju onoliko puta koliko ima karaktera u imenu
cl = karakteru prolaza
dl = duzini imena - broju prolaza (broj prolaza ide od 0 do duzine imena - 1)
dl je karakter pravog serijskog broja
vrati se na pocetak loopa ako je broj prolaza manji od duzine imena
```

Posto je ovo lak algoritam bez ikakve dalje analize dacu vam C++ source za keygenerator datog algoritma.

```
unsigned int i,dl,cl;
char name[100]="";
char serial[64]="";
 cl = strlen(name);
for(i=0;i<strlen(name);i++){
 dl = name[i];
 dl = dl - cl;
 serial[i] = dl;
 cl--;
}
```

Naravno ovo je samo isecak koda koji se koristi za generisanje tacnog serijskog broja. Ostaje vam samo da ovaj kod implementirate u svoj source kod ili da ga prepisete u nekom drugom programskom jeziku.

# KeyGen – Beginning #2

Videli smo kako mozemo da modifikujemo neki fajl kako bismo ga pretvorili u svoj keygenerator. Sada cemo pokusati nesto drugacije, pokusacemo da napisemo novi program koji ce za svako uneto ime generisati validan serijski broj. "Meta" ce za pocetak biti veoma jednostavna. Ucitajte program …\Casovi\Cas4\keygenme #1.exe u Olly.

Pre nego sto pocnemo sa keygeningom prvo bismo trebali da skupimo sto je moguce vise informacija o samoj meti. Stoga cemo skenirati metu PeIDom koji ce nam reci da je meta pisana u MASM32 / TASM32u. Posle ovoga cemo uneti lazne podatke u metu da vidimo kako se ponasa prilikom unosa podataka. Ovo je bitno jer treba da vidimo da li je duzina imena, serijskog broja ili neki drugi parametar definisan. Ako jeste program ce nam cesto saopstiti da je ime prekratko ili da neki drugi podatak ne zadovoljava uslove. Unecemo u program sledece podatke: ime ap0x, a kao serijski broj 111111. Kada pritisnemo dugme Register onda ce program izbaciti poruku: OOPS! Wrong serial. Ova poruka ce nam pomoci da nadjemo mesto gde se racuna serijski broj! Potrazicemo ovaj string u fajlu pomocu *Ollyja -> Search for -> All referenced text strings.*

Trazeni string cemo naci na adresi 0040120E. U okolini te adrese se nalazi sledece:

```
004011DE  |> \68 AC304000     PUSH keygenme.004030AC
004011E3  |.  68 D0304000     PUSH keygenme.004030D0
004011E8  |.  E8 13010000     CALL <JMP.&kernel32.lstrcmpA>
004011ED  |.  0BC0            OR EAX,EAX
004011EF  |.  75 16           JNZ SHORT keygenme.00401207
...
00401200  |.  E8 2B010000     CALL <JMP.&user32.MessageBoxA>       ; \MessageBoxA
00401205  |.  EB 14           JMP SHORT keygenme.0040121B
...
00401216  |.  E8 15010000     CALL <JMP.&user32.MessageBoxA>       ; \MessageBoxA
```

Kao sto vidimo poziva se komanda lstrcmpA koja poredi dva stringa. Jedan od ova dva stringa mora biti nas uneti lazni serijski broj dok je drugi skoro sigurno tacan serijski broj. Ovo smo zakljucili jer se u blizini ovog poredjenja nalaze poruke o tacno/pogresno unetom serijskom broju. Posto ovaj CALL pocinje na adresi 00401199, postavicemo break-point na nju. Posle ovoga cemo se vratiti u crackme i ponovo cemo pritisnuti dugme Register posle cega ce program zastati na adresi 00401199. Sa F8 cemo traceovati kroz program sve dok ne dodjemo do prvog CALLa koji se nalazi odmah ispred lstrcmpA komande. Ovaj CALL se nalazi na adresi 004011D2. U ovaj CALL cemo uci sa F7 jer je ocigledno da se ovde, pre poredjenja stringova, mora racunati pravi serijski broj. Ovako izgleda deo tog CALLa:

```
004012C8  |>  8A1C39          /MOV BL,BYTE PTR DS:[ECX+EDI]
004012CB  |.  84DB            |TEST BL,BL
004012CD  |.  74 08           |JE SHORT keygenme.004012D7
004012CF  |.  03C3            |ADD EAX,EBX
004012D1  |.  83C0 10         |ADD EAX,10
004012D4  |.  41              |INC ECX
004012D5  |.^ EB F1           \JMP SHORT keygenme.004012C8
```

Ovaj deo CALLa nam je zanimljiv jer se bas tu racuna tacan serijski broj! Kako ovo znamo? Jednostavno znamo da se serijski brojevi najcesce racunaju u CALLovima koji prethode poredjenju tacnog i laznog serijskog

broja. U ovim CALLovima uvek postoji neki loop koji je zaduzen za racunanje serijskog broja na osnovu unetog imena. Posto se ime sastoji od X karaktera, a serijski se racuna za svaki karakter najmanje jedan loop je potreban.
Ako ste procitali deo posvecen ASMu na pocetku knjige sigurno znate sta se to radi na adresi 004012C8. Ako EDI sadrzi adresu na kojoj se nalazi uneto ime, a ECX sadrzi nulu, a posto je BL 8bitni registar koji moze da ima samo jednobajtnu vrednost, zakljucujemo da se u registar BL smesta jedno po jedno slovo imena. Naravno u registar se ne smestaju slova nego heksadecimalne vrednosti ASCIIa svakog slova.
Na sledecoj adresi 004012CB se poredi registar BL sa samim sobom. Posto se na sledecoj adresi nalazi jedan uslovni skok zakljucujemo da ce se on izvrsiti samo ako je BL jednak 0. Ovo ce se desiti samo u dva slucaja:

1) Kada je jedan karakter imena jednak 0x00 (null)
2) Kada je iskoriscen ceo string

Posto nas prvi slucaj ne interesuje, koncentrisacemo se samo na drugi slucaj.
Na sledecoj adresi 004012CF se na registar EAX dodaje vrednost registra EBX. Ono sto je bitno da primetite da se registar EBX menja kada se menja i registar BL (*vise o ovome u delu posvecenom ASMu na pocetku knjige*). Zbog ovoga i zbog cinjenice sto se EAX nigde u ovom loopu ne brise pretpostavicemo da on sadrzi vrednost koja moze biti pravi serijski broj.
Na sledecoj adresi 004012D1 se u EAX registar dodaje broj 10h (*heksadecimalno*). Posle ovoga ostaje samo da se poveca registar ECX za jedan jer ECX predstavlja brojac slova naseg unesenog imena. ECX se stara da se za svaki prolaz ovog loopa koristi drugo slovo. Evo kako bi izgledao ovaj algoritam napisan u C-u:

```
#include <iostream>
int main(int argc, char *argv[])
{
        char name[50];
        printf("=-=-=-=-=-=-=-=-=-=-\n");
        printf(" KeyGen #2 by Ap0x\n");
        printf("=-=-=-=-=-=-=-=-=-=-\n");
        printf("Unesi ime: ");
        gets(name);
//      printf("-=-=-=-=-=-\n");
        int i;
        int ser;
        ser = 0;
        for(i=0;i< strlen(name);i++){
                ser = ser + (name[i] + 0x10);                   //EAX = EAX + Slovo + 10h
//              printf("+Tmp Serial %x za slovo %c\n",ser,name[i]);
        }
//      printf("-=-=-=-=-=-=-\n");
        printf(" Serijski broj je: %x\n",ser);
        printf("=-=-=-=-=-=-=-=-=-=-\n");
        getchar();
        return 0;
}
```

Vi vas keygenerator mozete napraviti u bilo kom programskom jeziku. Sve dok se drzite formule EAX = EAX + Slovo + 10h uspesno cete napraviti keygenerator. Bez obzira na sve uspesno smo napravili nas drugi keygenerator.

# KeyGen – Beginning #3

Prosli algoritam je bio dosta lak zbog cega cemo sada preci na jedan koji je za nijansu tezi. Meta za ovaj deo poglavlja se zove keygenme #2.exe a nalazi se u folderu Cas04. Ova meta je izabrana ne zbog tezine svoje rutine za generisanje serijskog broja nego zbog njene specificnosti. Ova specificnost se ogleda u delu rutine koji se koristi za "generisanje" validnog serijskog broja. Ako pokusate da resite ovaj problem preko karakteristicnih stringova naci cete se u velikom problemu! Naime ako potrazite string "Good work!":

```
0040733C  |.  3BC1            CMP EAX,ECX                 <- Poredjenje seriskog
0040733E  |.^ 0F85 A3FFFFFF   JNZ keygenme.004072E7
00407344  |.  33C0            XOR EAX,EAX
00407346  |.  6A 40           PUSH 40
00407348  |.  68 7C304000     PUSH keygenme.0040307C
0040734D  |.  68 90444000     PUSH keygenme.00404490      ; |Text = "Good Work! ..."
00407352  |.  6A 00           PUSH 0
00407354  |.  FF15 98104000   CALL user32.MessageBoxA
```

Kao sto se vidi iz ovoga poruka o tacnom serijskom se pojavljuje samo ako su EAX i ECX isti! Iz ovoga sledi da se tacan serijski racuna u istom ovom CALLu. Pogledacemo sam pocetak ovog CALLa:

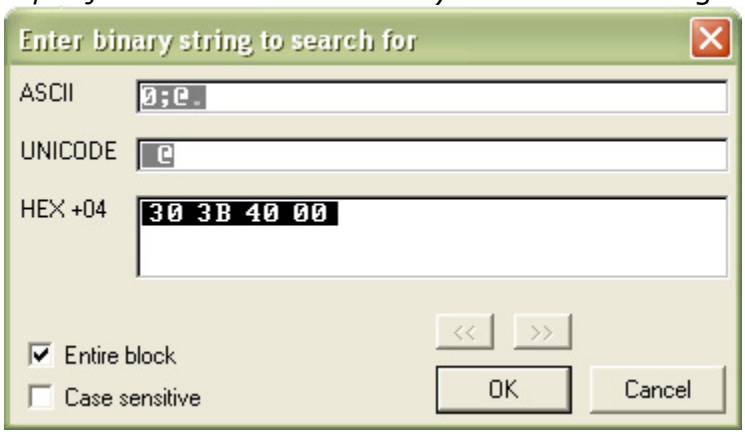
```
004072F0  |$  55              PUSH EBP                    <- Pravi pocetak CALLa
004072F1  |.  8BEC            MOV EBP,ESP
004072F3  |.  8BC8            MOV ECX,EAX
004072F5  |.  BE C8544000     MOV ESI,keygenme.004054C8
004072FA  |.  0FB606          MOVZX EAX,BYTE PTR DS:[ESI]
004072FD  |.  46              INC ESI
004072FE  |.  C1E0 02         SHL EAX,2
00407301  |.  8BD0            MOV EDX,EAX
00407303  |.  0FB606          MOVZX EAX,BYTE PTR DS:[ESI]
00407306  |.  46              INC ESI
00407307  |.  03D0            ADD EDX,EAX
00407309  |.  0FB606          MOVZX EAX,BYTE PTR DS:[ESI]
0040730C  |.  46              INC ESI
0040730D  |.  03C0            ADD EAX,EAX
0040730F  |.  03D0            ADD EDX,EAX
00407311  |.  0FB606          MOVZX EAX,BYTE PTR DS:[ESI]
00407314  |.  6BC0 0B         IMUL EAX,EAX,0B
00407317  |.  03D0            ADD EDX,EAX
00407319  |.  33F6            XOR ESI,ESI
0040731B  |.  8BF2            MOV ESI,EDX
0040731D  |.  0FAFD6          IMUL EDX,ESI
00407320  |.  8BC1            MOV EAX,ECX
00407322  |.  F7E2            MUL EDX
00407324  |.  03C0            ADD EAX,EAX
00407326  |.  81F0 58434000   XOR EAX,404358
0040732C  |.  A3 C85C4000     MOV DWORD PTR DS:[405CC8],EAX
00407331  |.  A1 303B4000     MOV EAX,DWORD PTR DS:[403B30]
00407336  |.  8B0D F4324000   MOV ECX,DWORD PTR DS:[4032F4]
0040733C  |.  3BC1            CMP EAX,ECX                 <- Poredjenje serijskog
0040733E  |.^ 0F85 A3FFFFFF   JNZ keygenme.004072E7
```

I sve bi ovo bilo super samo da nema jednog malog detalja koji ce nas razuveriti od ideje da se ovde racuna tacan serijski broj. Ako pogledamo adresu 0040732C i par adresa ispod zakljucicemo da:

```
0040732C      :: EAX se snima na adresu 00405CC8
00407331      :: U EAX se smesta sadrzaj adrese 00403B30
00407336      :: U ECX se smesta sadrzaj adrese 004032F4
0040733C      :: Porede se EAX i ECX, hmmm...
```

Ovo znaci da se EAX i ne racuna u ovom CALLu nego u nekom drugom. Ovo smo zakljucili na osnovu cinjenice da se sadrzaj adresa 00403B30 i

004032F4 ne menja tokom ovog CALLa. Stoga cemo pronaci mesto na kome se ove dve adrese menjaju. Pritisnucemo desno dugme na samom pocetku programa (*adresa 00407000*) i izabracemo Search for -> Binary string u koji cemo uneti okrenutu adresu 00403B30 (*ovo se okrece jer se binarne adrese zapisuju u reversnom redosledu*). Ovo treba da izgleda kao na sledecoj slici:




Posle ovoga je potrebno samo pritisnuti dugme OK koje ce nas odvesti do ASM komandi koje se koriste za pristupanje i/ili modifikovanje adrese 00403B30. Ovako cemo pronaci sve MOV PTR komande koje se koriste za pristupanje datoj adresi. Posle klika na dugme Ok Olly ce nas odvesti na sledecu adresu:

```
00407141  /$  55              PUSH EBP
00407142  |.  8BEC            MOV EBP,ESP
00407144  |.  BF 303B4000     MOV EDI,keygenme.00403B30          <- Ovde smo
00407149  |.  33DB            XOR EBX,EBX
0040714B  |>  33F6            /XOR ESI,ESI
0040714D  |>  3BDE            |/CMP EBX,ESI
0040714F  |.  74 2D           ||JE SHORT keygenme.0040717E
00407151  |.  33C9            ||XOR ECX,ECX
00407153  |.  53              ||PUSH EBX
00407154  |.  81C3 C8544000   ||ADD EBX,keygenme.004054C8
0040715A  |.  8A03            ||MOV AL,BYTE PTR DS:[EBX]
0040715C  |.  91              ||XCHG EAX,ECX
0040715D  |.  5B              ||POP EBX
0040715E  |.  56              ||PUSH ESI
0040715F  |.  81C6 C8544000   ||ADD ESI,keygenme.004054C8
00407165  |.  AC              ||LODS BYTE PTR DS:[ESI]
00407166  |.  5E              ||POP ESI
00407167  |.  3BC8            ||CMP ECX,EAX
00407169  |.  0F82 03000000   ||JB keygenme.00407172
0040716F  |.  AA              ||STOS BYTE PTR ES:[EDI]
00407170  |.  EB 0C           ||JMP SHORT keygenme.0040717E
00407172  |>  2BC1            ||SUB EAX,ECX
00407174  |.  83C0 FE         ||ADD EAX,-2
00407177  |.  81F0 58434000   ||XOR EAX,404358
0040717D  |.  AA              ||STOS BYTE PTR ES:[EDI]
0040717E  |>  46              ||INC ESI
0040717F  |.  83FE 05         ||CMP ESI,5
00407182  |.^ 7C C9           |\JL SHORT keygenme.0040714D
00407184  |.  43              |INC EBX
00407185  |.  83FB 07         |CMP EBX,7
00407188  |.^ 7C C1           \JL SHORT keygenme.0040714B
0040718A  |.  8BE5            MOV ESP,EBP
0040718C  |.  5D              POP EBP
0040718D  \.  C3              RET
```

Dakle ovo je rutina za generisanje tacnog serijskog broja! Kao sto vidimo rutina pocinje na adresi 0040714B, a zavrsava se na adresi 00407188. Sada nam ostaje samo da analiziramo ovu petlju kako bismo razumeli kako se to generise validan serijski broj. Ovako ce izgledati nasa analiza:

```
0040715A          U AL se stavlja prvo slovo unetog imena
0040715C          U ECX se stavlja prvo slovo iz imena
```

```
00407165        U EAX se stavljaju sva slova iz imena pocevsi od drugog
00407167        Porede se EAX i ECX, to jest prvo slovo i slovo iz prolaza
00407169        Ako je ECX manje od EAX onda:
00407172         EAX = EAX - ECX
00407174         EAX = EAX - 2
00407177         EAX = EAX xor 404358
0040717D        Snimi AL na EDI pointer
```

Kao sto vidimo algoritam je krajnje jednostavan i zasniva se na ponavljanju ove petlje onoliko puta koliko ima slova. Ne dajte da vas zbuni XOR sa brojem 404358, ovaj XOR je ekvivalentan XORu sa brojem 58. Zapamtite da se na adresu 00403B30 snima AL deo EAX registra. Ali bitno je da shvatite da se racuna i deo kada je EAX veci od 00404300. Evo kako bi keygenerator izgledao u Delphiju:

```
procedure ...
 var
 eax,i,ecx:integer;
 name,tmp:string;
begin
 eax := 0;
 tmp := '';
 name := Edit1.Text;
for i := 2 to 5 do begin
 if eax > $404300 then eax := eax + Ord(name[1]) else eax := Ord(name[1]);
 ecx := eax;
 eax := Ord(name[i]);
 if ecx < eax then begin
  eax := eax - ecx;
  eax := eax - 2;
  eax := eax xor $404358;
  eax := eax - $404300;
  tmp := tmp + Chr(eax);
  eax := eax + $404300;
 end
  else tmp := tmp + name[i];
end;
 Edit2.Text := tmp[1] + tmp[2] + tmp[3] + tmp[4];
end;
```

i u C++:

```
unsigned int eax,ecx,i;
        eax = 0;
for (i=1;i<=4;i++){
        if (eax > 0x00404300){
                eax = eax + name[0];
        }else{
                eax = name[0];
        }
        ecx = eax;
        eax = name[i];
        if (ecx < eax){
         eax = eax - ecx;
         eax = eax - 0x02;
         eax = eax ^ 0x00404358;
         eax = eax - 0x00404300;
         wsprintf(buffer,"%c",eax);
         strcat(serial,buffer);
         eax = eax + 0x00404300;
        }else{
         wsprintf(buffer,"%c",name[i]);
         strcat(serial,buffer);
        }
}
```

# KeyGen – Beginning #4

A sada nesto malo komplikovanije, pokusacemo da napravimo keygenerator za nasu "metu" u dva programska jezika. Mislim da su ova dva programska jezika uobicajena i da njih zna najveci deo citalaca ove knjige, stoga su programski jezici koji ce biti obradjivani u knjizi biti Visual Basic i Delphi."Meta" ce biti ista kao i u proslom poglavlju. Ucitajte program …\Casovi\Cas4\Serial2.exe u Olly. Kao sto smo u proslom poglavlju utvrdili serijski broj se generise u petlji koja pocinje na adresi 00407D83. Ono sto cemo sada uraditi je sledece:

1) izvojicemo ovaj deo koda za analizu
2) pokusacemo da ponovo napisemo ovaj algoritam u nekom drugom programskom jeziku

Evo kako izgleda ta petlja izdvojena:

```
00407D83    8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]
00407D86    8A4430 FF        MOV AL,BYTE PTR DS:[EAX+ESI-1]
00407D8A    34 2C            XOR AL,2C
00407D8C    25 FF000000      AND EAX,0FF
00407D91    03C0             ADD EAX,EAX
00407D93    8D0480           LEA EAX,DWORD PTR DS:[EAX+EAX*4]
00407D96    05 00040000      ADD EAX,400
00407D9B    8D55 F0          LEA EDX,DWORD PTR SS:[EBP-10]
00407D9E    E8 9DD5FFFF      CALL Serial2.00405340
00407DA3    8B55 F0          MOV EDX,DWORD PTR SS:[EBP-10]
00407DA6    8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
00407DA9    E8 7EBAFFFF      CALL Serial2.0040382C
00407DAE    46               INC ESI
00407DAF    4B               DEC EBX
00407DB0    75 D1            JNZ SHORT Serial2.00407D83
```

Analizirajmo polako prolazak kroz ovu petlju. Treba da vam je ukljucen Olly, da ste postavili break-point na adresu 00407D83, i da ste kao ime u "metu" uneli ap0x, serijski broj nije bitan.

**Prvi od pet prolaz:**

00407D83 Pri prvom prolazu kroz petlju u EAXu na adresi 00407D83 se nalazi broj 4, a posle izvrsenja koda na adresi 00407D83 nalazi se nase uneto ime to jest ap0x.

00407D86 Ovde se u AL stavlja nulti karakter stringa "ap0x". Kao sto primecujete slovo a je prvi karakter. Sta je onda nulti karakter??? Nulti karakter je isti za sve stringove i iznosi 00h ili samo 0. Uvek je isti.

00407D8A Ovde se izvrsava obicna XOR operacija nad registrom AL sa vrednoscu 2Ch ili 44.

00407D8C Ovde se izvrsava logicko dodavanje 0FFh odnosno 255 na vrednosti EAX. Ovo je obicno svodjenje sa velikih hex brojeva na manje. Shvatite to ovako: ako je EAX 0085005C onda ce posle ovog logickog dodavanja rezultat biti 5C. Posto je posle izvrsenja ove komande EAX jednak registru AL ova komanda je nepotrebna i necemo je koristiti u keygenu.

00407D91 Na ovoj adresi se desava obicno matematicko sabiranje vrednosti iz EAXa sa samom sobom.

00407D93 Ovde EAX dobija sledecu vrednost EAX = EAX + (EAX * 4), ovo je prosto i lako procitati samo treba obratiti paznju na matematicke operacije mnozenja.
00407D96 Opet operacija obicnog matematickog dodavanja 400h = 1024 na vrednost EAXa.
Sve do kraja nam je totalno nebitno jer kako vidimo samo se vrednost iz EAXa samo prebacuje iz brojeva u string i dodaje na novi prazan string. U sledecim prolazima desava se isto to samo se stringovi dodaju jedan na drugi, to jest vrednosti iz svakog prolaza se kao tekst dodaju jedna na drugu. Kao sto vidimo prve cetiri cifre serijskog broja su uvek iste jer se racunaju uvek za isto slovo, to jest za nulu. Ako izracunamo ovo videcemo da su prve cetiri cifre uvek 1464. Svaki sledeci prolaz se razlikuje samo u podatku koji se nalazi u ALu na adresi 00407D86, a on je uvek jednak ASCII kodu slova koje odgovara prolazu kroz petlju. Ako je u pitanju prolaz jedan, AL ce dobiti vrednost ASCII broja prvog slova iz unetog imena, i tako dalje za svako slovo. Nadam se da vam je ovo jasno i da nisam previse zakomplikovao ovo tumacenje koda. Keygenovanje samo po sebi nije jednostavno i zahteva mnogo vezbanja i iskustva, zato ne ocajavajte ako vam sve nije iz prve jasno. Trik kod pravljenja dobrih keygenova je razumevanje svakog reda vezanog za racunanje serijskog broja i njegovo rekreiranje u nekom programskom jeziku. Evo kako bi to izgledalo napisano u Visual Basicu.

```
Private Sub Command1_Click()
 User_name = Text1.Text
 Serial = "1464"
For i = 1 to Len(user_name)
 Al = Asc(Mid$(user_name,i,1))
 Al = Al xor 44
 Al = Al + Al
 Al = Al + (Al * 4)
 Al = Al + 1024
 Serial = Serial & Al
Next i
 Text2.Text = Serial
End Sub
```

Ovaj primer se nalazi vec gotov u folderu …\Casovi\Cas4\Keygen source\VB\
A evo kako bi to izgledalo napisano u Delphiju.

```
procedure TForm1.Button1Click(Sender: TObject);
var
 user_name,serial:string;
 al,i:integer;
begin
 serial := '1464';
 user_name := Edit1.Text;
 for i := 1 to Length(user_name) do begin
  al := Ord(user_name[i]);
  al := al xor 44;
  al := al + al;
  al := al + (al *4);
  al := al + 1024;
  serial := serial + IntToStr(al);
 end;
  Edit2.Text := serial;
end;
```

Ovaj primer se nalazi vec gotov u folderu …\Casovi\Cas4\Keygen source\Delphi\

# KeyGens & Smart Check #1

Do sada smo obradili keygenovanje programa preko Olly-a, a sada je na redu Smart Check. Za potrebe ovog dela skinuo sam primer sa interneta, on se nalazi ovde …\Casovi\Cas3\abexcrackme2.exe, slobodno ga ucitajte u SC i pokrenite sa F5, kada se pojavi Abexov crackme unesite kao ime cracker a kao serijski 111111, pritisnite [ Check ] i izaci ce MessageBox 'Nope, the serial is wrong!'
Sada se u SCu pojavio [+]Click kao oznaka da smo mi kliknuli na dugme, duplim klikom na [+]Click pojavljuje se ovo:

```
<-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><->
  Len(VARIANT:String:"cracker") returns LONG:1242252
  Len(VARIANT:String:"cracker") returns LONG:1242252
  Integer (1) --> Long (1)
  Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
  Asc(String:"c") returns Integer: 99
  Hex(VARIANT:Integer:199)
  Integer (2) --> Long (2)
  Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
  Asc(String:"r") returns Integer: 114
  Hex(VARIANT:Integer:214)
  Integer (3) --> Long (3)
  Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
  Asc(String:"a") returns Integer: 97
  Hex(VARIANT:Integer:197)
  Integer (4) --> Long (4)
  Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
  Asc(String:"c") returns Integer: 99
  Hex(VARIANT:Integer:199)
<-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><->
```

Na osnovu ovoga zakljucujemo da se serijski broj generise od prva 4 slova imena. Za svako slovo se izvaja ASCII broj od koga se racuna Hex broj :) Ali se za racunanje Hex broja dodaje jos 100 na ASCII vrednost svakog slova. To izgleda ovako:

```
serial = ..... Hex(Asc(Mid$(name,1,1)) + 100)
```

Ali sta se dalje radi sa serijskim? Da li se Hex brojevi sabiraju, mnoze, dele, dodaju?

Obelezimo MsgBox i idemo na View -> Show All Events da bi smo imali detaljniji uvid u ono sto se desava. Sada vidimo ovo par redova iznad MsgBox-a:

```
<-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><->
  _VbaVarMove(VARIANT:String:"111111",.....) returns ....          <- Nas lazni serijski broj
  _VbaFreeObj(...)
  _VbaVarTstEq(VARIANT:String:"C7D6C5C7",VARIANT:String:"111111") returns DWORD:0
<- Da li je C7D6C5C7 = 111111
  _VbaVarTstNe(VARIANT:String:"C7D6C5C7",VARIANT:String:"111111") returns DWORD:F
<- Da li C7D6C5C7 nije = 111111
  _VbaVarDup(...)
  _VbaVarDup(...)
   MsgBox(....)                                    <- Poruka o pogresno seriskom
<-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><->
```

Analizirajmo ono sto vidimo. Nas lazni serijski broj se pojavljuje u memoriji, onda se uporedjuje sa nekom C7D6C5C7 vrednoscu i posle toga se pojavljuje poruka o netacnom serijskom broju. Znaci 100% je C7D6C5C7 bas taj serijski broj koji nama treba. Probajmo sada korisnicko ime cracker i seriski broj C7D6C5C7 i to je to, uspeli smo !!!! Program sada izbacuje poruku 'Yap, that`s the right key!' :)  Posto znamo ovo odozgo:

```
<-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><->
   Len(VARIANT:String:"cracker") returns LONG:1242252
   Len(VARIANT:String:"cracker") returns LONG:1242252
   Integer (1) --> Long (1)
   Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
   Asc(String:"c") returns Integer: 99
   Hex(VARIANT:Integer:199)                          <- C7
   Integer (2) --> Long (2)
   Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
   Asc(String:"r") returns Integer: 114
   Hex(VARIANT:Integer:214)                          <- D6
   Integer (3) --> Long (3)
   Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
   Asc(String:"a") returns Integer: 97
   Hex(VARIANT:Integer:197)                          <- C5
   Integer (4) --> Long (4)
   Mid(VARIANT:String:"cracker",long:1,VARIANT:Integer:1)
   Asc(String:"c") returns Integer: 99
   Hex(VARIANT:Integer:199)                          <- C7
<-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><-><->
```

bice nam lako da napravimo keygen. Posto znamo da je serijski za ime cracker - C7D6C5C7, vise je nego ocigledno kako to program generise serijski broj. On se racuna od prva 4 slova imena iz kojih se racuna Ascii kod za svako slovo, koji se koristi za izracunavanje Hex vrednosti. Primeticemo da je Ascii od "c" jednak 99 a da program racuna Hex(199) umesto Hex(99), isto tako za sva 4 slova se na Ascii vrednost slova dodaje 100 i od toga se racuna Hex vrednost od koje se obicnim dodavanje jedne vrednosti na drugu tzv. "lepljenjem" dobija konacni serijski broj. C7 D6 C5 C7 = C7D6C5C7.
Evo kako bi keygen izgledao u Visual Basicu.

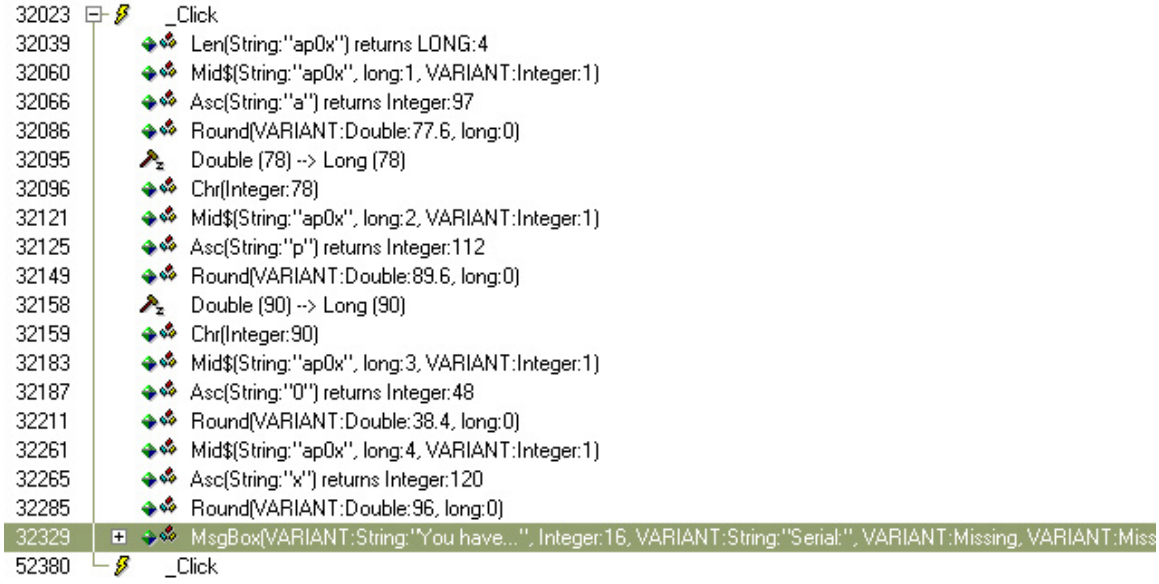
```
ime = Text1.Text
If Len(ime) < 4 Then
 MsgBox "Unesite ime duze od 4 karaktera"
Else
 For i = 1 To 4
  serial = serial & Hex(Asc(Mid$(ime, i, 1)) + 100)
 Next
 Text2.text = serial
End If
```

A evo kako bi izgledao u Delphiju.

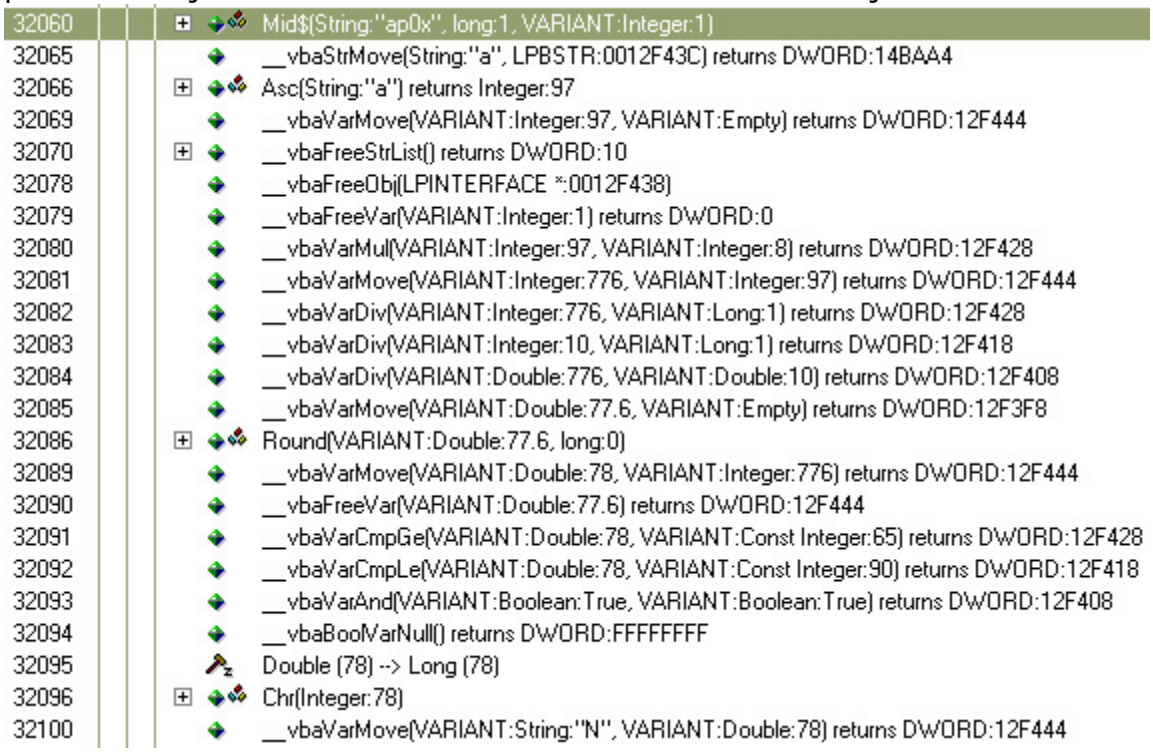
```
serial := '';
ime := Edit1.Text;
If Length(ime) < 4 Then begin
 messagedlg('Unesite ime duze od 4 karaktera',mtError,[mbOK],0);
end
 else
begin
 For i := 1 To 4 do begin
  serial := serial + IntToHex(Ord(name[i]) + 100);
 end;
 Edit2.text := serial;
end;
```

# KeyGens & Smart Check #2

Primer za koji cemo sada napisati keygenerator je vec obradjivan u ovoj knjizi u poglavlju koje se bavi "pecanjem" serijskih brojeva. Taj primer se nalazi ovde ...\Cas03\keygenme03b.exe. Sledicemo standardne korake u trazenju pravog serijskog broj i u nasu metu cemo uneti sledece lazne podatke: ap0x i 111111. Posle pritiska na dugme Check pojavice se dogadjaji




Posto pravimo keygenerator moracemo da detaljno analiziramo ovaj serijski algoritam. Kao sto vidimo program proverava svako slovo unetog imena zbog cega zakljucujemo da se serijski broj racuna u jednom loopu. A njega smo posle selekcije reda 32060 i klika na Show all events izdvojili ovde:



Naravno dovoljno je da analiziramo samo jedan prolaz, to jest jedno slovo imena da bismo zakljucili kako se generise serijski broj. Ovo proistice iz cinjenice da su svi prolazi isti jer se serijski broj racuna u istom loopu. Analizirajmo sta se ovde desava:

***32066*** *- Izdvaja se prvo slovo unetog imena*
***32080*** *- ASCII vrednost prvog slova se mnozi sa 8*
***32084*** *- Dobijena vrednost na 32080 se deli sa 10*
***32086*** *- Dobijena vrednost sa 32084 se zaokruzuje*
***32091*** *- Da li je zaokruzena vrednost sa 32086 veca od 65*
***32092*** *- Da li je zaokruzena vrednost sa 32086 manja od 90*
***32096*** *- Ako je bilo koji od ova dva uslova ispunjen pretvori 32086 u ASCII*
*- Inace samo dodaj ovaj broj kao string na AoC-03B-*

Posto se sve ovo ponavlja za svako slovo nema potrebe ispisivati ceo loop. Ove informacije koje imamo su nam dovoljne da napisemo keygenerator za ovu metu. Evo kako bi on izgledao u VBu:

```
serial = "AoC-03B-"
 For i = 1 To Len(Text1.Text)
  tmp = Asc(Mid$(Text1.Text, i, 1))
  tmp = tmp * 8
  tmp = Round(tmp / 10)
  If tmp >= 65 And tmp <= 90 Then tmp = Chr(tmp)
  serial = serial & tmp
 Next i
Text2.Text = serial
```

gde je Text1.Text polje u koje se unosi ime. Ovako bi taj algoritam izgledao u Delphiju:

```
procedure generate;
var
 serial:string;
 tmp,i:integer;
begin
serial := `AoC-03B-`;
 For i := 1 To Length(Edit1.Text)
  tmp := Ord(Edit1.Text[i]);
  tmp := tmp * 8;
  tmp := tmp div 10;
  If (tmp >= 65) And (tmp <= 90) Then serial := serial + Chr(tmp)
     else serial := serial + IntToStr(tmp);
 end;
 Edit2.Text := serial;
end;
```

gde je Edit1.Text polje u koje se unosi ime.

# 05 CD Checking...

Ovo poglavlje ce detaljno teorijski i prakticno obraditi osnovna pitanja vezana za proveru CD. Ovakve provere se najcesce nalaze u igricama i ne dozvoljavaju vam da ih startujete bez prisustva CDa u uredjaju. Ako ste ovaj problem do sada resavali raznim simulatorima virtualnih uredjaja ovo je pravo stivo za vas. Ne samo da cete ustedeti na prostoru nego cete nauciti i nesto novo vezano za reversni inzenjering.

# CD Checking - *Examples*

Prvo cu vas upoznati sa osnovnom teorijom provere CDova. Treba da shvatite da su CD promene jako slicne uklanjanju NAG ekrana. U 90% slucajeva se radi o jednostavnim porukama koje program izbacuje ako u CD-ROMu ne nadje odgovorajuci CD. Ove poruke mozete veoma lako presresti i izmeniti program tako da se ova poruka nikada ne prikaze. Ovo cete moci uraditi samo sa programima koji nisu pakovani nekim pakerom ili nisu zasticeni nekim programom koji zabranjuje kopiranje CDova (*citaj: SafeDisc, StarForce3 i slicno*). Da bismo odredili da li je neka igra zasticena najlakse je da otvorimo glavni ExE fajl igrice u PeIDu i vidimo da li je pakovan ili ne. Ali ne brinite postoji veliki broj novih i starih igara koji nemaju ovakvu zastitu i dobre su za praksu (*citaj: Quake 3, Worms World Party, GTA: Vice City,...* ), a za one zasticene postoje programi kao sto su UnSafeDisc i slicni koji ce vam "pomoci" da otpakujete ExE fajl i sredite ga normalno kao da nije ni bio pakovan. Ove osnovne zastite koje zahtevaju prisustvo CDa u CD-Romu izgledaju manje ili vise ovako u ASMu:

```
CALL check_for_cd
TEST EAX,EAX
JE no_cd_inserted
MOV EAX,1
...
```

Ovde se poziva zamisljena adresa check_for_cd koja proverava da li postoji neki fajl na CDu, ako taj fajl ne postoji onda je pogresan ili nikakav CD ubacen. U sledecoj liniji se proverava EAX sa samim sobom sto ce za rezultat imati da zero flag (*vrednost koja moze biti samo 0 ili 1, a vrednost joj se uvek i samo dodeljuje posle nekog poredjenja tipa TEST ili CMP*) bude postavljen kao true (1) ili false (0), a ako je zero flag jednak nula onda ce program skociti na zamisljenu adresu no_cd_inserted i prikazace poruku o pogresnom CDu. Naravno ako je CD ubacen onda ce zero flag biti jednak jedan i igrica ce se startovati. Ovo problem mozemo resiti jednostavnim NOPovanjem JE no_cd_inserted skoka i onda ce za program stalno biti ubacen CD. Ali ne dajte se zavarati da ce igrica stvarno raditi bez CDa ako joj je nesto od fajlova stvarno potrebno sa CDa da bi radila.

Recimo da nas interesuje sta se nalazu u CALLu check_for_cd. Tu bismo mogli da vidimo nesto slicno ovome:

```
check_for_cd:
   ...
   MOV ECX,03       {neke ASM funkcije koje su nama nebitne}
   MOV EBX,401234{neke ASM funkcije koje su nama nebitne}
   ...
   TEST EAX,EAX
   JE good_cd
   XOR EAX,EAX
   RET
   ...
good_cd:
   MOV EAX,1
   RET
```

Ovde tri tackice naravno znace neodredjeni i proizvoljni broj linija koda. Prve dve komande MOV nisu ni bitne, tu su samo da predoce neki kod. Ono sto je bitno je neka TEST provera, recimo postojanje odredjenog fajla na CDu. Ako fajl postoji onda ce se JE skok izvrsiti i EAX ce sadrzati broj 1, a ono JE no_cd_inserted se nece izvrsiti. A ako taj fajl ne postoji onda ce se izvrsiti XOR EAX,EAX sto je jednako komandi MOV EAX,0 i onda ce se onaj skok JE no_cd_inserted izvrsiti i prikazace se poruka o pogresnom CDu.

Najcesce se u CALLu koji sluzi za proveru CDa mogu naci i funkcije to jest CALLovi koji sluze za proveru tipa drajva. To jest da li je drajv koji se posmatra CD-ROM, FLOPPY, HARD-DISC, DVD,... Za ovo se koristi windowsova api funkcija GetDriveTypeA. Ona glasi ovako:

```
GetDriveType Lib "kernel32" Alias "GetDriveTypeA" (ByVal nDrive As String) As Long
```

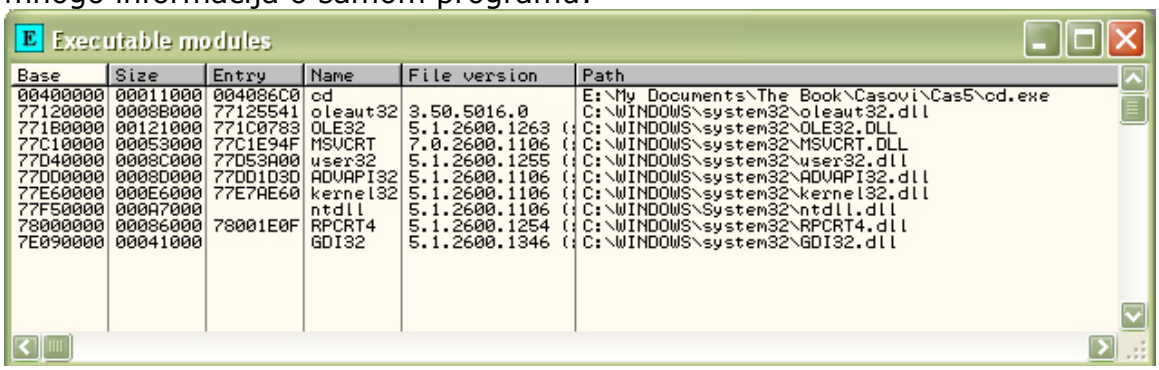
U ASMu bi ovo izgledalo otpilike ovako:

```
check_for_cd:
    ...
    MOV ECX,03      {neke ASM funkcije, koje su nama nebitne}
    MOV EBX,401234{neke ASM funkcije, koje su nama nebitne}
    ...
    ...
    PUSH drive_letter
    CALL GetDriveTypeA
    CMP EAX,00000005
    JNE not_CD_drive
    ...
not_CD_drive:
    XOR EAX,EAX
    RET
```

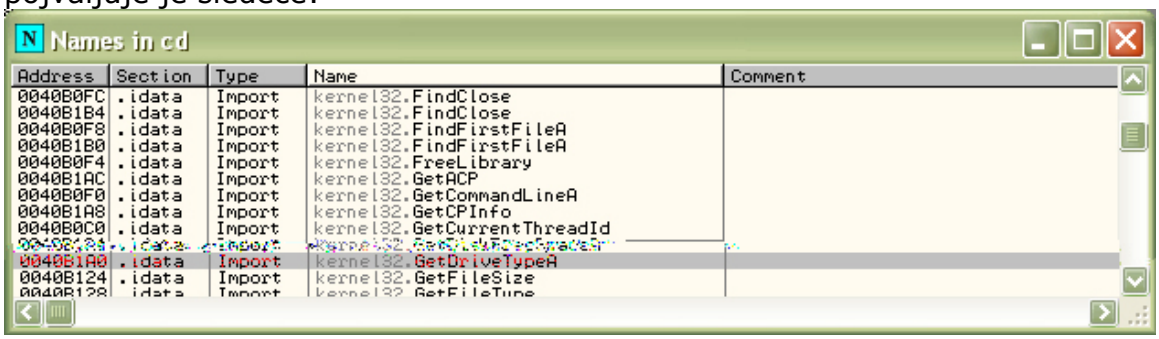
Ovde se desava sledece. Prosledjuju se potrebni parametri funkciji GetDriveTypeA, ona se poziva i vraca rezultat u EAX. Ako u EAXu nije broj 5 onda skoci na deo koji brise EAX (EAX = 0) i vrati se iz ovog CALLa. Primecujete da je identifikacioni broj CD uredjaja 5. Ovo JNE se takodje moze srediti jednim NOPom. Ova funkcija nam moze mnogo pomoci pri trazenju mesta gde se proverava tip uredjaja a samim tim i da li je pravi CD ubacen. Kao sto vidite postoji vise nacina na koje mozemo srediti CD proveru, ali najjednostavniji je sredjivanje skoka odmah posle CALL check_for_cd funkije. Ovo su samo neki teorijski primeri iz moje glave. CD provera uopste ne mora a u vecini slucajeva i ne izgleda ovako, ali najcesce postoji veoma mnogo slicnosti izmedju ovoga sto sam napisao i situacija u “stvarnom svetu”.

# CD Checking - CrackMe

Posle teorije stize praksa. Za potrebe ovog dela knjige je napisan program koji se nalazi …\Casovi\Cas5\cd.exe ovde. Ovo je jedan odlican primer posto tacno prikazuje kako bi radila neka CD zastita u praksi. Program ce prvo proveriti sve CD / CD-RW uredjaje u kompjuteru i na svakom od njih potrazice neki fajl, ako taj fajl postoji i ima sadrzinu koju treba da ima onda je u pitanju pravi CD. Naravno ovo je najjednostavniji primer koji se moze zaobici i virtualnim CDom ali mi to ne zelimo, zar ne ? Startujte program i videcemo da izgleda veoma obicno. Ako pritisnemo dugme :: Run Game :: pojavice se prozor o ne unetom CDu CD uredjaj. Ovo cemo da "popravimo". Ako se secate teorijskog dela ovog poglavlja ovo ce biti izuzetno lako. Ucitajmo ovu "metu" u OllyDBG. Mogli bismo da nadjemo mesto gde se stvara ovaj dijalog ali pokusacemo nesto drugo ovaj put. Znamo da program sigurno mora da koristi Windows API funkciju GetDriveTypeA kako bi odredio koji od uredjaja u kompjuteru je CD-ROM. Sve sto treba da uradimo je da nadjemo gde se poziva ova funkcija. Sreca nasa sto koristimo Olly jer je trazanje ovakvih API poziva izuzetno lako. Pritisnimo dugme E iz gornjeg toolbara ili ALT + E na tastaturi. Ovaj prozor koji trenutno vidimo daje nam mnogo informacija o samom programu:

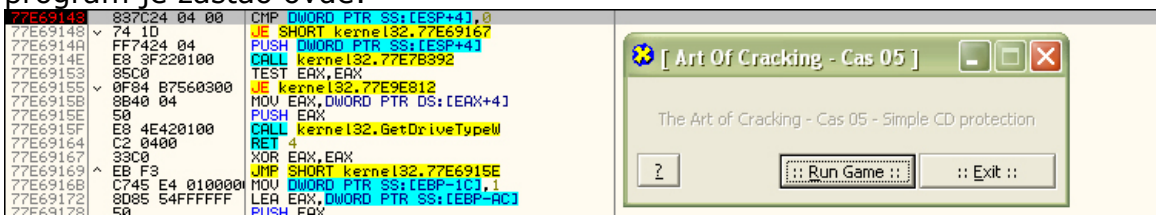



U ovom prozoru mozemo videti koji su sve fajlovi u direktnoj upotrebi od strane ovog programa, naravno Olly ce prikazati samo odredjene programske module (.vxd,.ocx,.dll i sl.) koji sadrze odredjeni ASM kod koji ovaj program poziva. Ono sto mi zelimo da saznamo je koje API pozive vrsi ovaj program. Kliknucemo desnim dugmetom misa na prvi red (on je ujedno i putanja do fajla pa i sam taj fajl) i kliknucemo na View Names. Ono sto se sada pojvaljuje je sledece:




Spisak svih API poziva je podeljen po sekcijama i dll fajlovima u kojima se nalaze. Da biste lakse nasli API koji nama treba mozete da kucate na

tastaturi njegovo ime. Sada kada smo ga nasli i selektovali postavicemo break-point na njega, pritisnite desno dugme na tom APIju i selektujete Toggle breakpoint on import, sto znaci da ce program stati kada program prvi put pozove ovaj API. Sada pokrenite program i pritisnite ::Run Game:: i program je zastao ovde:




Ako pogledate ovo cudno mesto gde smo sada nista vam nece biti jasno. Videcete samo da EAX sadrzi string A: i da ovaj deo koda najverovatnije sluzi sa proveru uredjaja A: Ali ovde nam je nesto cudno, pogledajte te adrese sa strane 77??????, kao da se nalazimo u nekom DLLu ? Ako pogledate ime prozora Ollyja videcete da u njemu pise [CPU main thread, module kernel32] sto znaci da mi i jesmo u DLLu i to u kernel32.dll Skinimo ovaj break-point sa 77E69143 (*moze se razlikovati na vasem kompjuteru*) i pritiskajuci F8 sve dok se ne izvrsi i RET 4 komanda. Sada smo se iz kernel32.dll-a vratili u nasu metu. Nalazimo se u jednoj malo duzoj petlji odmah ispod:

```
00408403  |.  50              |PUSH EAX                       ; /RootPathName
00408404  |.  E8 93C7FFFF     |CALL JMP.&kernel32.GetDriveTypeA   ; \GetDriveTypeA
```

a odmah iznad

```
00408409  |.  83F8 05         |CMP EAX,5 <- Ovde smo
0040840C  |.  0F85 98000000   |JNZ cd.004084AA
00408412  |.  8D85 20FEFFFF   |LEA EAX,DWORD PTR SS:[EBP-1E0]
00408418  |.  B9 2C854000     |MOV ECX,cd.0040852C            ;  ASCII "\main.xxx"
0040841D  |.  8B55 F8         |MOV EDX,DWORD PTR SS:[EBP-8]
00408420  |.  E8 FBB9FFFF     |CALL cd.00403E20
00408425  |.  8B85 20FEFFFF   |MOV EAX,DWORD PTR SS:[EBP-1E0]
0040842B  |.  E8 E4D5FFFF     |CALL cd.00405A14
00408430  |.  84C0            |TEST AL,AL
00408432  |.  74 76           |JE SHORT cd.004084AA
```

Ono sto iz ovoga moze da se zakljuci a sto znamo iz teorije je da ako se EAX poredi sa 5, a sve ovo se desava ispod nekog poziva GetDriveTypeA APIja to znaci da se proverava da li je drajv CD-ROM ili ne. Ako se u EAXu ne nalazi 5 nego neka druga vrednost program ce skociti na adresu 004084AA, to jest ovde:

```
004084AA  |> \43              |INC EBX
004084AB  |.  83FB 5B         |CMP EBX,5B
004084AE  |.^ 0F85 08FFFFFF   \JNZ cd.004083BC
```

Primecujemo da se ovde EBX poredi sa 5Bh ili sa 91 decimalno, ako je broj veci ili jednak program ce nastaviti sa izvrsavanjem i izaci ce iz ove petlje. Ovo znaci da ce se pretraziti svi uredjaji u sistemu od A: do Z: (90 je ASCII kod za Z) u potrazi za CD-ROMom. Ok sada smo saznali dosta o ovoj zastiti postavimo break-point na 00408409 da saznamo sta se dalje desava posle provere CMP EAX,5. Ako pritisnemo F8 program ce izvrsiti ovu komandu i preci ce na izvrsavanje sledece. Ono sto ne zelimo da se desi je da se izvrsi skok na kraj petlje pa cemo NOPovati adresu 0040840C. Izvrsavamo kod sa F8 sve dok ne stignemo do adrese 0040842B kada nam sadrzaj EAXa izgleda jako zanimljivo. EAX je sada jednak "A:\main.xxx". Sta ovo znaci ? Ovo znaci da program ili proverava postojanje fajla "A:\main.xxx" na "CDu" ili

pokusava da otvori taj fajl. Ako fajl ne postoji ili ne moze biti otvoren onda ce se skok na adresi 00408432 izvrsiti i program ce proveriti neki drugi uredjaj. Pogledajmo sta bi se to izvrsilo kada bi fajl postojao:

```
00408434  |.  8D85 1CFEFFFF   |LEA EAX,DWORD PTR SS:[EBP-1E4]
0040843A  |.  B9 2C854000     |MOV ECX,cd.0040852C              ;  ASCII "\main.xxx"
0040843F  |.  8B55 F8         |MOV EDX,DWORD PTR SS:[EBP-8]
00408442  |.  E8 D9B9FFFF     |CALL cd.00403E20
00408447  |.  8B95 1CFEFFFF   |MOV EDX,DWORD PTR SS:[EBP-1E4]
0040844D  |.  8D85 2CFEFFFF   |LEA EAX,DWORD PTR SS:[EBP-1D4]
00408453  |.  E8 0CA5FFFF     |CALL cd.00402964
00408458  |.  8D85 2CFEFFFF   |LEA EAX,DWORD PTR SS:[EBP-1D4]
0040845E  |.  E8 9DA2FFFF     |CALL cd.00402700
00408463  |.  E8 B0A1FFFF     |CALL cd.00402618
00408468  |.  8D55 FC         |LEA EDX,DWORD PTR SS:[EBP-4]
0040846B  |.  8D85 2CFEFFFF   |LEA EAX,DWORD PTR SS:[EBP-1D4]
00408471  |.  E8 86A7FFFF     |CALL cd.00402BFC
00408476  |.  8D85 2CFEFFFF   |LEA EAX,DWORD PTR SS:[EBP-1D4]
0040847C  |.  E8 E7A7FFFF     |CALL cd.00402C68
00408481  |.  E8 92A1FFFF     |CALL cd.00402618
00408486  |.  8B45 FC         |MOV EAX,DWORD PTR SS:[EBP-4]
00408489  |.  BA 40854000     |MOV EDX,cd.00408540              ;  ASCII "CD_main_xxx"
0040848E  |.  E8 85BAFFFF     |CALL cd.00403F18
00408493  |.  75 05           |JNZ SHORT cd.0040849A
00408495  |.  BE 01000000     |MOV ESI,1
0040849A  |   8D85 2CFEFFFF   |LEA EAX,DWORD PTR SS:[EBP-1D4]
004084A0  |.  E8 7BA5FFFF     |CALL cd.00402A20
004084A5  |.  E8 6EA1FFFF     |CALL cd.00402618
```

Ovo i ne izgleda kao nesto posebno, nema nikakvih poredjenja i uslova sem jednog skoka na adresi 00408493 koji kada se ne izvrsi ESI postaje jednak 1. Hmmm ovo bi mozda moglo biti zanimljivo? Ako pogledamo odmah ispod ove petlje videcemo ovo:

```
004084B4  |.  85F6            TEST ESI,ESI
004084B6  |.  75 18           JNZ SHORT cd.004084D0
```

A ako se ovaj skok izvrsi vodi nas pravo na poruku o ubacenom CDu u CD-ROM i do "startovanja igre". Ako pogledamo pre same petlje videcemo da je pocetno stanje ESIa jednako 0.

```
004083B5  |.  33F6            XOR ESI,ESI
004083B7  |.  BB 41000000     MOV EBX,41
```

Analizom ove petlje dosli smo do zakljucka da ce program ispitati sve uradjaje u kompjuteru od A: do Z: u potrazi za CDom koji na sebi ima fajl "main.xxx", i ako je ovo sve ispunjeno i taj fajl sadrzi nesto sto se u njemu trazi (najverovatnije string "CD_main_xxx"), ako je ovo sve ispunjeno onda ce ESI biti jednak 1 kao znak da se CD sa igricom nalazi u sistemu. Ovaj problem mozemo resiti na razlicite nacine.

1) Mozemo promeniti skok na adresi 004084B6 u JMP tako da se igra uvek startuje (*ovo je ujedno i naljaksi nacin*)
2) Mozemo umesto JNE (jnz) na adresi 004084AE uneti MOV ESI,1 tako da je ESI pre same provere (TEST ESI,ESI) sigurno jednak 1
3) Mozemo izmeniti adresu 004083B2 i uneti MOV ESI,1 umesto postojeceg koda i tako cemo pre same petlje ESIju dodeliti vrednost 1 i izbrisati ono XOR ESI,ESI koje ESIi dodeljuje vrednost 0.
4) Postoji mnogo nacina, vi kao vezbu uz ovo poglavlje smislite neki koji ja nisam ovde nabrojao.

# 06 Code Hacking...

Prvi deo ovog poglavlja ce vas nauciti kako da uzmete neki deo tudjeg koda i stavite ga u svoj. Ovo ce biti veoma interesantno za Delphi programere koji su se suocili sa problemom reversovanja veoma dugackog algoritma i zeleli bi da ubrzaju proces rekodiranja tudjeg koda. Ova tehnika vam moze dobro doci kod pravljenja keygeneratora ali i kod drugih stvari.

Drugi deo ovog poglavlja ce vas nauciti kako da ubacite svoj kod u tudji program, kako da naterate bilo koji program da se ponasa kako vi zelite, da ubacite svoje poruke, dijaloge u programe.

# Delphi and ASM

Kao veoma obican i lak primer sam za ovaj sesti cas pripremio Bangalyev CrackMe #2. On kao i ostali primeri koje ja nisam napisao mogu se preuzeti sa adrese www.crackmes.de Ako "propustimo" ovaj primer kroz PeID videcemo da je on pisan u TASMu / MASMu. Ok otvoricemo Olly da vidimo sta se desava unutar ovog programa. Pretpostavicu da znate da pronadjete mesto gde se generise serijski broj pa cemo odmah preci na stvar. Znaci imamo ovaj deo koda koji nam je interesantan i ne zelimo da ponovo pisemo ovu rutinu:

```
00401304  |.  B8 01000000     MOV EAX,1
00401309  |>  8B15 38304000  /MOV EDX,DWORD PTR DS:[403038]
0040130F  |.  8A90 37304000  |MOV DL,BYTE PTR DS:[EAX+403037]
00401315  |.  81E2 FF000000  |AND EDX,0FF
0040131B  |.  8BDA           |MOV EBX,EDX
0040131D  |.  0FAFDA         |IMUL EBX,EDX
00401320  |.  03F3           |ADD ESI,EBX
00401322  |.  8BDA           |MOV EBX,EDX
00401324  |.  D1FB           |SAR EBX,1
00401326  |.  03F3           |ADD ESI,EBX
00401328  |.  2BF2           |SUB ESI,EDX
0040132A  |.  40             |INC EAX
0040132B  |.  49             |DEC ECX
0040132C  |.^ 75 DB          \JNZ SHORT Key-Crac.00401309
0040132E  |.  56             PUSH ESI
```

Mozemo da postavimo break-point na adresu 00401309 i da prodjemo par puta kroz ovaj loop da vidimo sta se to desava ovde. Videcemo da se u prva tri reda u EDX stavlja prvo,drugo,... slovo iz imena, u ostalim se vrse neke racunske operacije i ovaj loop se ponavlja vise puta, odnosno onoliko puta koliko ime ima slova. Ono sto vidimo je da se svi registri sem ESI resetuju to jest da dobijaju neke nove vrednosti. Ovo ujedno znaci da ce na kraju loopa ESI sadrzati pravi serijski broj. Ako ovo odavde ne mozete da zapazite imate i deo koda malo ispod koji vam upravo ovo govori:

```
0040133A  |.  3BC6           CMP EAX,ESI
0040133C  |.  75 15          JNZ SHORT Key-Crac.00401353
```

Ono sto cemo mi uraditi je da cemo iskoristiti ovaj ASM kod da napravimo Delphi keygenerator, ali to necemo raditi prevodjenjem ovog ASM koda u Delphi nego malom modifikacijom ovog ASM koda i njegovim ubacivanjem u Delphi program. Sve od adrese 0040131B pa do adrese 00401328 cemo ostaviti kako jeste, dok cemo ostatak prilagoditi. Znaci ovo cemo malo prilagoditi:

```
MOV EBX,EDX
IMUL EBX,EDX
ADD ESI,EBX
MOV EBX,EDX
SAR EBX,1
ADD ESI,EBX
SUB ESI,EDX
```

Ono sto moramo da uradimo je da pre izvrsavanja ovoga u EDX stavimo vrednost ASCII koda nekog slova iz imena, a na kraju da ESI sacuvamo u neku promenljivu.Znaci kod cemo izmeniti u ovo:

```
asm             {Pocetak ASM bloka}
 XOR ESI,ESI    {Dodali smo ovu ASM komandu}
 MOV ESI,serial{U delphiju i unutar ASM bloka mozemo koristiti sve promenljive}
 MOV EBX,slovo{koje su definisane unutar procedure koju koristimo}
 XOR EBX,EBX
 MOV EBX,EDX
 IMUL EBX,EDX
 ADD ESI,EBX
 MOV EBX,EDX
 SAR EBX,1
 ADD ESI,EBX
 SUB ESI,EDX
 MOV serial,ESI {ESI registar brise po zavrsetku bloka pa ga moramo sacuvati u promenljivoj}
end;            {Kraj ASM bloka}
```

Ono sto smo ovde dodali je brisanje ESIa i dodeljivanje vrednosti promenljive serial ESIju, a na kraju dodeljivanje vrednosti ESIja promenljivoj serial. Posto ovo treba da se ponavlja za svako slovo iz imena, to ce u Delphiju izgledati ovako:

```
procedure TForm1.Button1Click(Sender: TObject);
 var
 slovo,i,serial:longint;
 ime:string;
begin
 serial := 0;
 ime := Edit1.Text;
for i := 1 to Length(ime) do begin
 slovo := Ord(ime[i]);
asm
  XOR ESI,ESI
  MOV ESI,serial
  MOV EDX,slovo
  XOR EBX,EBX
  MOV EBX,EDX
  IMUL EBX,EDX
  ADD ESI,EBX
  MOV EBX,EDX
  SAR EBX,1
  ADD ESI,EBX
  SUB ESI,EDX
  MOV serial,ESI
 end;
end;
  messagedlg('Vas serijski broj je: ` + IntToStr(serial) + ` !', mtInformation,[mbOK],0);
end;
```

Ovo je jako jednostavan primer ali vam pokazuje kako mozete iskoristiti deo tudjeg koda da napravite u ovom slucaju keygenerator. U folderu …\Casovi\Cas6\keygen-source\ se vec nalazi ovaj gotov primer i source code za njega.

**NAPOMENA:** Ovo poglavlje je zamisljeno da bude od velike koristi Delphi programerima koji se bave reversnim inzenjeringom i zeleli bi da reversuju neke algoritme bez njihovog rekodiranja u Delphi sintaksu, ostalima ovaj deo poglavlja nece biti od velike vaznosti pa ga mozete preskociti.

# VC++ and ASM

Vec smo videli da kako se u Delphiju koristi ASM, a sada ostaje da iskoristimo ovo znanje i da ga primenimo na VC++. Za ovo cemo iskoristiti isti primer koji smo iskoristili i za Delphi. Evo kako bi to izgledalo u C++u:

```
#include <stdio.h>
#include <string.h>

int main(void){
 unsigned int i,serial,ln;
 char* ime;
 char name[50];
 printf("Enter name: ");
 gets(name);
 ln = strlen(name);
 ime = name;
 serial = 0;
 __asm{
   MOV ECX,ime
   MOV EAX,0h
  lop:
   movzx EDX,byte ptr[ECX+EAX]
   MOV EBX,EDX
   IMUL EBX,EDX
   ADD serial,EBX
   MOV EBX,EDX
   SAR EBX,1
   ADD serial,EBX
   SUB serial,EDX
   INC EAX
   CMP EAX,ln
  jne lop
 }
 printf("Serial: %d \n",serial);
 gets(name);
 return 0;
}
```

Analiziracemo malo ovaj kod. Kao sto vidimo prvo cemo konzolno uneti nase ime. Prvo cemo videti da je ASM blok odvojen od ostalog koda pomocu komande *__asm{}* koja oznacava blok komandi. Takodje cete primetiti da ime mora biti tipa const char jer ce se u protivnom desiti Access exception i program ce se srusiti. Sledeci problem predstavlja prosledjivanje slova, jedno po jedno, samoj petlji? Ovo cemo resiti pomocu sledecih komandi:

```
MOV ECX,ime   ...   MOVZX EDX,BYTE PTR[ECX+EAX]
```

Gde prva komanda u ECX registar prosledjuje const char ime (*unutar ASM bloka se mogu koristiti sve varijable definisane u programu),* a druga EDX registru dodeljuje HEX vrednost svakog slova. Ovo se radi preko pointera koji prevashodno prvo pokazuje na ECX, sto predstavlja prvo slovo unetog imena. Posto se EAX povecava u svakom prolazu tako i EDX dobija drugu vrednost sve dok se ne iskoriste sva slova i EAX dobije vrednost duzine unetog imena. Od ostalih specificnosti bitno je znati da se labele oznacavaju isto kao i u MASMu to jest kao *ime:*. Ka ovim labelama se moze usmeravati program pomocu bilo koje standardne JMP komande i njenih varijanti. Labele se takodje mogu pozivati i u smislu CALL komande ali se onda kao i kod ASM programiranja mora navesti izlazak (*RET komanda*) iz CALLa koji pocinje sa labelom.

# Adding functions #1

Ovo poglavlje ce vas nauciti kako da dodate najobicniji NAG ekran u neki program i da se program posle toga moze startovati :) Pogledajte primer CRC32.exe koji se nalazi u folderu Cas06. Primeticete da je to jedan obican CRC32 kalkulator i da ako ga skeniramo sa PeIDom videcemo da je pisan u Visual C++. Ova informacija je jako bitna jer se ova tehnika dodavanja funkcija moze primeniti samo na Delphi, C++ i ASM programe.

Ono sto cemo mi u ovom poglavlju uraditi je dodavanje obicnog NAG ekrana pre samog pocetka programa. Ovaj NAG ce biti obican Message Box koji ce govoriti da smo mi dodali funkciju u program. Otvoricemo program pomocu Ollyja i zapamticemo kako izgleda OEP. Trebace nam samo prvih par linija koda pa cemo ih iskopirati u Notepad. Trebaju nam ove linije:

```
00401580 > $  55              PUSH EBP
00401581   .  8BEC            MOV EBP,ESP
00401583   .  6A FF           PUSH -1
00401585   .  68 00404000     PUSH CRC32.00404000
0040158A   .  68 88264000     PUSH CRC32.00402688
```

Zasto smo uzeli samo ove linije??? Zato sto cemo zameniti ovaj OEP sa nekom nasom komandom ali posle izvrsavanja naseg koda ovaj OEP moramo i da vratimo nazad pa smo ga backupovali.

Prvo trebamo naci mesto gde cemo ubaciti nas NAG ekran. To mesto uvek trazimo tamo gde ima puno praznih, neiskoriscenih 00 ili 90 bajtova. Posto ce nam za ubacivanje NAGa i vracanje OEPa trebati oko 50 bajtova onda cemo mesto za ubacivanje koda traziti na kraju fajla gde po pravilu ima mnogo slobodnih 00 bajtova. Kao sto vidimo od adrese 00403356 pa do 00403FFF ima vise nego dovoljno mesta za ovo:

```
00403356    00                DB 00
...
00403FFF    00                DB 00
```

Tacnije ovde ima 3241 bajt slobodnog prostora. Pre ubacivanja samog MessageBoxA moramo da znamo da li se u fajlu nalazi API poziv ka MessageBoxA apiju i ako se nalazi koji su mu parametri. Mozete slobodno da odete u module i proverite da se API MessageBoxA stvarno nalazi u fajlu i da se API moze koristiti. Sama API funkcija ima sledece parametre:

```
MessageBoxA(hwnd, Text, Naslov, MB_TIPMESSAGEBOXA);
```

gde su:

- hwd; vlasnik messageboxa, moze biti nula
- Text; Tekst koji ce pisati unutar messageboxa
- Naslov; Naslov prozora messageboxa
- Tip; Da li je messagebox informacija ili upozorenje ili....

A ovo bi u ASMu izgledalo ovako:

```
PUSH Tip
PUSH Naslov
PUSH Text
PUSH 0
CALL MessageBoxA
```

Pre nego sto unesemo ovaj CALL negde u slobodnom prostoru prvo moramo da unesemo sve tekstove koji se pojavljuju u messageboxu. Ovo radimo zato sto se u PUSH komandama za naslov i text ne nalaze ASCII vrednosti nego

adrese do ASCII vrednosti koje se koriste. Selektovacemo adresu 00403358 i pritisnucemo desno dugme -> Binary -> Edit... Ako vam je otkaceno Keep Size iskljucite ga. Pa cemo uneti sledeci tekst bez navodnika “Cracked:”. Kada pritisnemo OK program ce pokazati ovo:

```
00403358    43              INC EBX
00403359    72 61           JB SHORT CRC32.004033BC
0040335B    636B 65         ARPL WORD PTR DS:[EBX+65],BP
0040335E    64:3A00         CMP AL,BYTE PTR FS:[EAX]
```

Posto ovo nije pravo stanje stvari pritisnucemo CTRL+A da bismo analizirali fajl i da bismo videli pravo stanje stvari. Ono sto se pojavilo je ovo:

```
00403358  .  43 72 61 63 6>ASCII "Cracked:",0
```

I to je ono sto treba ovde da se nalazi. Predpostavljam da vas interesuje sta znaci ova nula iza Stringa Cracked:. Ona predstavlja kraj stringa, to jest prvo se ispisuju sve ASCII vrednosti slova iz stringa pa se onda taj string zatvara sa 0x00 bajtom. Ovo C++ koderi sigurno znaju. Dalje cemo selektovati adresu 00403361 i na isti nacin cemo uneti string “The art of cracking was here!!!”, takodje bez navodnika. Sada imamo ovo:

```
00403358  .  43 72 61 63 6>ASCII "Cracked:",0
00403361     00          DB 00 <- Ostavio sam nulu pre unosenja drugog stringa, vi ne morate
00403362  .  54 68 65 20 6>ASCII "The art of Crack"
00403372  .  69 6E 67 20 7>ASCII "ing was here!!!",0
```

Odmaci cemo se malo od ovog teksta i unecemo NAG pocevsi od adrese 00403384. Selektujte tu adresu i pritisnite Space da biste presli u Assembly mod i unesite sledece:

```
PUSH 40 <- Zato sto je 40 Hex vrednost za MB_ICONINFORMATION, a moze da se unese i PUSH
0, sasvim je svejedno, rec je samo o ikoni dialoga.
```

Pritisnite Assemble i negasenjem Assembly prozora samo unesite novi red:

```
PUSH 00403358 <- jer se na adresi 00403358 nalazi naslov messageboxa
```

Pritisnite Assemble i negasenjem Assembly prozora samo unesite novi red:

```
PUSH 00403362 <- jer se na adresi 00403362 nalazi tekst messageboxa
```

Pritisnite Assemble i negasenjem Assembly prozora samo unesite novi red:

```
PUSH 0 <- jer je hwnd jednak 0
```

Pritisnite Assemble pa Cancel jer nam treba malo pomoci pri unosenju sledeced reda. Znate da sledeca komanda koja sledi je CALL MessageBoxA ali nam za njeno pozivanje treba tacna adresa API MessageBoxA pa cemo skociti do prozura Modules (ALT+E) i otici na desno dugme na fajl CRC32.exe i selektovati View Names. U prozoru koji se sada pojavio potrazite MessageBoxA. Videcete da se u tom redu nalazi i adresa na kojoj se nalazi sam API. Dakle vidimo ovo:

```
Names in CRC32, item 34
 Address=004081B4
 Section=.idata
 Type=Import  (Known)
 Name=USER32.MessageBoxA
```

A adresa APIja je 004081B4. Posto znamo podatak koji nam je potreban vraticemo se na adresu 00403390 i unecemo sledece:

```
CALL DWORD[004081B4] <- Pozovi adresu na kojoj se nalazi API
```

I sada smo uspesno ubacili NAG u program. Sada preostaje samo da prepravimo OEP tako da se umesto prve linije koda izvrsava ovaj red koji prikazuje NAG screen. Ovo se moze uraditi na dva nacina.

**NACIN 1:**
Prvi nacin je izuzetno lak i ne zahteva nikakve dodatne alate sem Ollyja. Posto smo backupovali OEP u Notepad znamo tacno kako on izgleda i mozemo ga izmeniti i posle vratiti. Dakle u Notepadu imamo ovo:

```
00401580 > $  55              PUSH EBP
00401581   .  8BEC            MOV EBP,ESP
00401583   .  6A FF           PUSH -1
00401585   .  68 00404000     PUSH CRC32.00404000
0040158A   .  68 88264000     PUSH CRC32.00402688
```

Otici cemo na adresu 00401580 i izmeniti je tako da ona vodi direktno do adrese 00403384 gde pocinje prikazivanje NAGa. Izmenicemo adresu 00401580 u jedan obican skok:

```
00401580 >   /E9 FF1D0000    JMP CRC32.00403384
00401585   .|68 00404000     PUSH CRC32.00404000
0040158A   .|68 88264000     PUSH CRC32.00402688
```

Dakle uneli smo jmp 00403384 ali smo na racun toga izgubili par redova OEPa koje moramo negde nadoknaditi. Izgubljeni redovi su:

```
00401580 > $  55              PUSH EBP
00401581   .  8BEC            MOV EBP,ESP
00401583   .  6A FF           PUSH -1
```

Pa cemo njih uneti odmah ispod NAGa koji pocinje na adresi 00403384. Otici cemo dole i na adresi 00403398 cemo uneti delove koji nedostaju.

```
00403398   .  55              PUSH EBP
00403399   .  8BEC            MOV EBP,ESP
0040339B   .  6A FF           PUSH -1
```

i dodacemo jedan skok koji ce vratiti program na prvu neizmenjenu adresu odozgo, na 00401585.

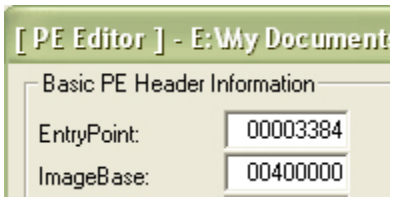
```
0040339D   .^\E9 E3E1FFFF     JMP CRC32.00401585
```

I time smo zavrsili sa dodavanjem obicnog NAG ekrana u program. Sada ostaje samo da snimimo promene i pokrenemo snimljeni fajl. Fajl crc32_added.exe pokazuje kako dodati NAG treba da izgleda.

**NACIN 2:**
Prvi nacin je izuzetno lak ali drugi nacin moze biti jos laksi ali zahteva upotrebu dva alata: Ollyja i LordPe-a. Kada smo uneli NAG treba odmah ispod NAGa da dodamo skok koji ce nas vratiti na OEP posto se posle izvrsavanja NAGa program vraca na pocetak. Dodajte red:

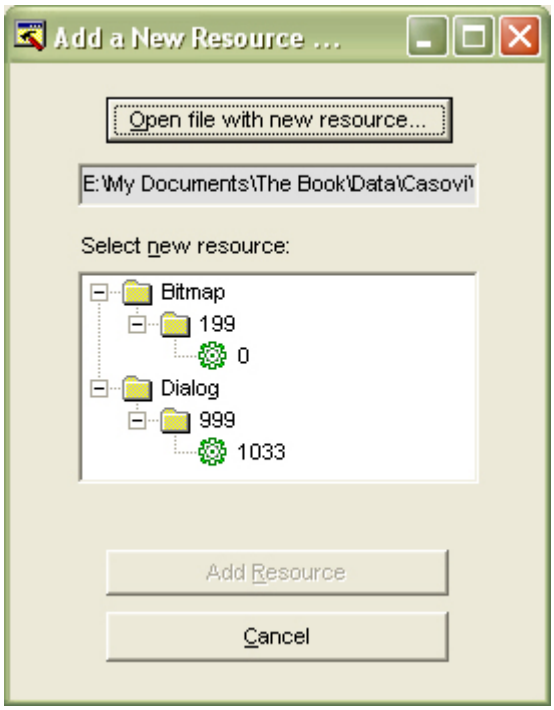
```
00403398   .^\E9 E3E1FFFF     JMP CRC32.00401580
```

Ovo radimo zato sto cemo promeniti OEP programa sa 00401580 na 00403384 tako da ce se prvo izvrsiti NAG pa tek onda sve ostalo. Kada smo dodali i ovaj red snimimo promene. Da bismo promenili OEP fajla otvoricemo novosnimljeni fajl pomocu LordPe-a. Kliknite na PE-Editor dugme i izaberite novosnimljeni fajl i u njemu cemo izmeniti OEP u novu adresu 00403384 – ImageBase 00400000 = 00003384. Ovo treba da izgleda kao na slici. Posle promena mozemo da pritisnemo Save, da zatvorimo LordPE i da startujemo fajl sa izmenjenim OEPom. Videcemo da smo uspeli i da program radi bez ikakvih problema.



# Adding functions #2

Prvi deo poglavlja je objasnio dodavanje MessageBoxa u program. U drugom delu ovog poglavlja cu vam pokazati kako da dodate Dijalog u neki program i kako da ga nacinite 100% funkcionalnim. Zvuci li zanimljivo? Pa da pocnemo. Otvoricemo metu original.exe koja se nalazi u folderu Cas6, pomocu ResHackera. Zasto? Zato sto moramo da dodamo nove resurse u sam exe fajl a to cemo najlakse uraditi pomocu ResHackera. Sada vidite zasto sam




rekao da ResHacker ima jos mnogo korisnih funkcija. Kao sto vidimo na slici pored izabracemo iz glavnog menija Action -> Add new resource i kada se pojavi dijalog sa slike izabracemo fajl addonDialog.res i dodacemo jedan po jedan oba resursa koja se nalaze u tom fajlu. Ne ja ovde nemam nameru da objasnim kako se prave .res fajlovi. Za ovo cete sami morati da pregledate Microsoftovu dokumentaciju o Resource Compileru. Posle ovog dodavanja moramo da mapiramo, to jest da zapisemo sve IDove koji se nalaze u novododatom Dijalogu. ID samog dijaloga je 999 (3E7h), dugme ? ima ID 3007 (BBFh), dugme Visit Web page ID 3005 (BBDh), a dugme Exit ID 3006 (BBEh). Ovo nam je potrebno kako bismo povezali sve ove dugmice sa funkcijama koje ce raditi svoj posao. Sada mozemo da snimimo promene u ovom fajlu, snimicemo ovaj fajl kao AddOn.exe. Mozemo da zavrsimo sa ResHackerom i da otvorimo novosnimljeni fajl Ollyjem. Pre nego sto pocnemo sa dodavanjem funkcija u fajl prvo moramo da odlucimo gde cemo ih dodati i kako. Posto moramo da prikazemo novi dijalog na ekranu backupovacemo sledece linije ASM koda u recimo Notepad, jer ce nam kasnije trebati:

```
00408722  .  6A 00              PUSH 0                          ; /lParam = NULL
00408724  .  68 C8854000        PUSH AddOn.004085C8             ; |DlgProc =
00408729  .  6A 00              PUSH 0                          ; |hOwner = NULL
0040872B  .  6A 64              PUSH 64                         ; |pTemplate = 64
0040872D  .  FF35 4CA84000      PUSH DWORD PTR DS:[40A84C]      ; |hInst = 0
00408733  .  E8 C4C4FFFF        CALL DialogBoxParamA            ; \DialogBoxParamA
```

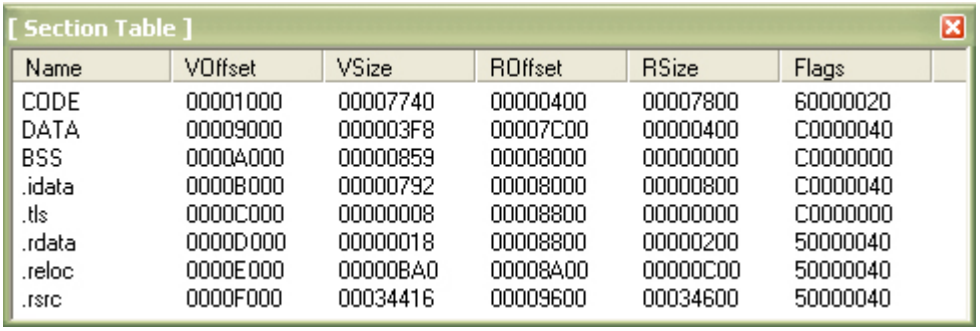
dalje treba da backupujemo poruku koja ce se pojaviti kada pritisnemo dugme ?. Dakle treba da backupujemo ovo:

```
00408611  |.  6A 40             PUSH 40                         ; /Style = MB_OK
00408613  |.  A1 90924000       MOV EAX,DWORD PTR DS:[409290]   ; |
00408618  |.  50               PUSH EAX                        ; |Title => "About..."
00408619  |.  A1 94924000       MOV EAX,DWORD PTR DS:[409294]   ; |
0040861E  |.  50               PUSH EAX                        ; |Text => "U "
0040861F  |.  53               PUSH EBX                        ; |hOwner
00408620  |.  E8 07C6FFFF       CALL MessageBoxA                ; \MessageBoxA
```

Ovo radimo jer cemo cemo umesto pritiska na dugme ? uraditi da se pojavi nas novi ubaceni dijalog. Kada se pojavi novi dijalog onda cemo ubaciti ovu poruku. Dakle plan izgleda ovako:

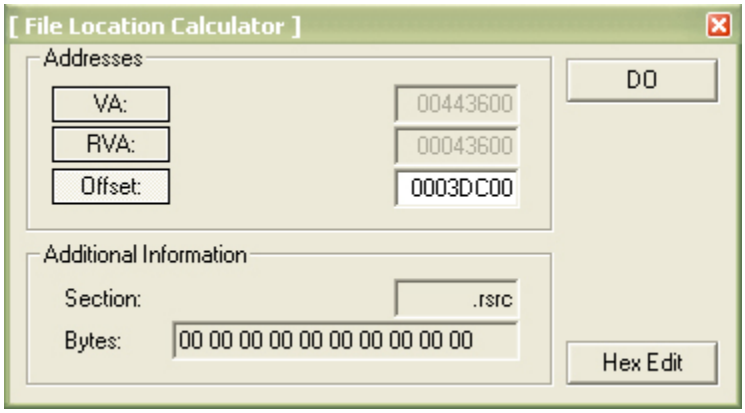
1) Dodati dijalog pomocu ResHackera
2) Osloboditi mesto za kod koji cemo ubaciti
3) Napisati kod koji ce se kaciti na message loop za obradu poruka
4) Napisati kod za svako dugme novog dijaloga
5) Dodati kod za vracanje iz dodatog koda u message loop

Ufff... ovo je mnogo posla a imamo i jedan veliki problem. Moramo prvo da oslobodimo dovoljno mesta da bismo mogli da ubacimo novi kod koji ce uraditi sve sto smo zamislili. Za ovo ce nam trebati jedan dobar Hex Editor i LordPe, ja sam koristio staru verziju Hex WorkShop jer mi je jedina bila pri ruci, ali vi mozete koristiti bilo koji Hex Editor. Prvo cemo otvoriti AddOn.exe sa LordPe-om i naci cemo sve sekcije.




| Name | VOffset | VSize | ROffset | RSize | Flags |
|---|---|---|---|---|---|
| CODE | 00001000 | 00007740 | 00000400 | 00007800 | 60000020 |
| DATA | 00009000 | 000003F8 | 00007C00 | 00000400 | C0000040 |
| BSS | 0000A000 | 00000859 | 00008000 | 00000000 | C0000000 |
| .idata | 0000B000 | 00000792 | 00008000 | 00000800 | C0000040 |
| .tls | 0000C000 | 00000008 | 00008800 | 00000000 | C0000000 |
| .rdata | 0000D000 | 00000018 | 00008800 | 00000200 | 50000040 |
| .reloc | 0000E000 | 00000BA0 | 00008A00 | 00000C00 | 50000040 |
| .rsrc | 0000F000 | 00034416 | 00009600 | 00034600 | 50000040 |

Uobicajeno je da prosirimo poslednju sekciju za odredjeni broj bajtova. Dakle otvoricemo isti ovaj fajl pomocu Hex Editora, otici cemo na poslednji bajt i dodacemo jos 1000 0x00 bajtova. Pocetni bajt ce biti 3DC00 a poslednji 3DEE0. Kada ovo uradimo vraticemo se u LordPe i izmenicemo fizicku i virtualnu velicinu .rsrc sekcije sa 00034416 na 000348B0 jer smo dodali 1000 bajtova. Moracemo jos i da promenimo karakteristike (*flags*) same sekcije tako da bismo mogli i da ubacimo kod u nju na F00000E0. Sada moramo da sracunamo gde se to nalazi virtualna adresa 3DC00. Za ovo cemo iskoristiti FLC opciju iz LordPe-a.



Ova RVA adresa je 00443600 a ovo predstavlja prvu RVA adresu od 1000 novih dodatih bajtova. Ovde cemo dodavati nas kod. Sada pocinje prava zabava :)
Prvo cemo dodati sve ASCII vrednosti koje su nam potrebne. Selektovacemo adresu 00443600 i pritisnucemo desno dugme i izabracemo Binary -> Edit... U novootvorenom prozoru cemo u ASCII polje uneti http://www.google.com a primeticete da Keep size checkbox ne sme da bude otkacen.
Drugi string cemo ubaciti na adresi 00443616, tako da cemo selektovati 00443615 i izmeniti drugi bajt i tri iza njega u string "open" - bez navodnika. Zasto ovako razmacinjemo dva stringa??? Zato sto se svaki string mora zavrsavati sa po jednim 0x00 bajtom.
Dalje cemo presresti loop koji je zaduzen za prevodjenje poruka i njihovu interpretaciju. Ovaj loop se najlakse nalazi tako sto cemo naci mesto gde se nalaze WM_nesto komande. Taj CALL pocinje ovde 004085C8, a vec je definisan kada se pojavio default dijalog na ekranu. Pogledajte ovu adresu:

```
00408724   .  68 C8854000      PUSH original.004085C8          ; |DlgProc = original.004085C8
```

NOPovacemo sve od 00408611 pa do 00408624 jer nam taj dao koda ne treba. Na mesto prikazivanja poruke kada se pritisne dugme ? cemo ubaciti pojavljivanje novog dijaloga i obradu poruka. Selektovacemo adresu 00408611 i na to mesto cemo ubaciti sledeci kod:

```
JMP 00443620
```

a ovo ce nas odvesti do novog koda koji cemo dodati na toj adresi.Na adresi 00443620 dodajte kod:

```
CMP ESI,0BBF
JNZ 00443643
PUSH 40                                 {
MOV EAX,DWORD PTR DS:[409290]
PUSH EAX
MOV EAX,DWORD PTR DS:[409294]                   Kod za prikazivanje MsgBoxA
PUSH EAX
PUSH EBX
CALL 00404C2C                           }
JMP 00408625
NOP
NOP
```

Sta se ovde desava ??? Prvo se ESI (sadrzace ID objekta koji je pritisnut) uporedjuje sa 0BBF, to jest proveravamo da li je pritisnuto drugo (*novo*) dugme ?, a ako jeste onda cemo prikazati backupovani messagebox sa starog *(00408611)* ? Posle ovoga se proverava ako ovo nije tacno program ce skociti na neku dalju adresu, a ako je ESI 0BBF onda ce se prikazati MessageBox na ekranu. Ako se pokaze MessageBox na ekranu onda cemo se posle prikazivanja MessageBoxa vratiti u loop za proveru poruka pomocu JMPa. Dalje cemo selektovati adresu 00443643 i unecemo sledece:

```
CMP ESI,0BBE
JE 00408374
```

Ovde poredimo ESI sa dugmetom za gasenje programa, EXIT. Ako je pritisnuto ovo dugme program ce otici na mesto iznad message loopa koje je zaduzeno za gasenje programa. Ovo smo ranije odredili da se nalazi na adresi 00408374. Dalje cemo selektovati adresu 00443655 i unecemo:

```
CMP ESI,0BBD
JNZ 00443673
PUSH 1                                  {
PUSH 0
PUSH 0
```

```
PUSH 443600                          Kod za otvaranje web-strane
PUSH 443616
PUSH 1
CALL 00408388                        }
JMP 00408625
```

Prvo cemo uporediti ESI sa 0BBD, to jest sa dugmetom Visit Web site. Ako je pritisnuto ovo dugme onda cemo ga posetiti pomocu ShellExecuteA funkcije. Napominjem da adresu na kojoj se nalazi ova funkcija u fajlu morate naci u View Names opciji u modulima. I poslednje, ali i najvaznije je da se postaramo da se novi dijalog i pojavi na ekranu. Selektovacemo adresu 00443673 i unecemo sledece:

```
CMP ESI,0BBA
JNZ 00408625
PUSH 0                               {
PUSH 4085C8
PUSH 0
PUSH 3E7                             Kod za prikazivanje dijaloga
PUSH DWORD PTR DS:[40A84C]
CALL 00404BFC                        }
JMP 00408625
```

Ovde smo prvo uporedili ESI sa 0BBA, odnosno sa starim dugmetom ?, ovo nas je podsetilo da ovakav CMP nismo NOPovali gore pa cemo se posle vratiti na to. Ako je pritisnuto staro dugme ? onda cemo prikazati dijalog na ekran. Ovo je samo copy / paste backupovanog koda sa 00408722 koji sluzi za prikazivanje prvog dijaloga na ekran. Posle ovoga se jednostavno vracamo u gornji loop za proveru WM poruka sa obicnom JMP komandom. Sada cemo snimiti nase promene, selektovacemo kod od 00443600 do 0044369D (*jer smo taj deo koda dodali*) i pritisnucemo desno dugme -> Copy to executable -> Section i snimicemo ovaj fajl kao Addon2.exe, pa cemo otvoriti ovaj novi fajl pomocu Ollyja.

Sada cemo se vratiti da povezemo stari loop sa novim loopom poruka. NOPovacemo sve od 00408611 pa do 00408624 jer nam taj dao koda ne treba. Na njegovo mesto cemo ubaciti pojavljivanje novog dijaloga i obradu poruka. Selektovacemo adresu 00408611 i na to mesto cemo ubaciti sledeci kod:

```
JMP 00443620
```

Posle ovoga nam ostaje samo da ubijemo staru proveru za dugme ?, BBA. Zato cemo NOPovati adrese:

```
00408609  |> \81FE BA0B0000          CMP ESI,0BBA
0040860F  |.  75 14                  JNZ SHORT AddOn2.00408625
```

Sada kada smo sve zavrsili mozemo da snimimo promene pritiskom na desno dugme -> Copy to executable -> All modifications -> Copy All -> desno -> Save file... i mozemo probati fajl. Ako ste nesto propustili ili niste dobro uradili program ce vam se srusiti, dobro uradjen primer imate u istom folderu pod imenom AddOn3.exe, pa mozete videti kako to treba da izgleda.

**NAPOMENA:** Ovo je izuzetno teska tehnika i zahteva zaista dosta prakse. Zbog ovoga ne ocajavajte ako ne razumete sve sada ili ne mozete da naterate program da se startuje. Posle kad sakupite dosta reverserskog iskustva vratite se na ovo poglavlje.

# Adding functions #3

Prvi deo poglavlja je objasnio dodavanje MessageBoxa u program, drugi deo ovog poglavlja vam je pokazao kako da dodate Dijalog u neki program, a u trecem cu vam pokazati kako da dodate API koji vam je potreban za reversing u IAT tabelu bilo kog programa.

Ovaj deo posla, ubacivanje u IAT, je do skora bio jako tezak i zahtevao jer ili rucnu modifikaciju PE Sekcija, dodavanje nove sekcije, pravljenje validnog IAT headera, ili ubacivanje API poziva preko LoadLibraryA funkcije i trazenje APIja po njegovom ordinalnom broju. Iako su oba nacina veoma korisna ja cu vam ovde objasniti kako se ovo veoma lako radi pomocu samo jednog alata: IIDKinga.


Ovaj alat se koristi izuzetno jednostavno:

1) Selektujte aplikaciju koju dodajete API
2) Ako zelite da ubacite dodatani kod u .exe unesite dodatnu velicinu sekcije (u ovom slucaju 1000)
3) Unesite ime .dll fajla
4) Unesite ime API poziva
5) Pritisnite + da dodate API i taj .dll-a
6) Pritisnite Add them da biste snimili fajl

Imajte na umu da sada mozete dodavati API pozive iz bilo kog broja .dll fajlova. Posle pritiska na dugme Add them !! IIDKing ce napraviti .txt fajl sa adresama do svih dodatih API poziva. IIDKing program mozete naci u folderu Cas6.

# 07 Getting caught

Ovo poglavlje sadrzi osnove detekcije debuggera i osnovne trikove detekcije modifikacije ili pokusaja modifikacije memorije programa kao i programa samog. Ove tehnike ce biti opisane detaljno tako da ce koristi od ove oblasti imati i software developeri i reverseri. Prvi da unaprede zastitu na svom software-u a drugi da saznaju kako je jednostavno ukloniti primitivne zastite.

# SoftICE detection

Evo jednog primera za Delphi programere. Ovaj deo koda je zasebna funkcija koja se koristi za detekciju aktivnog SoftICEa. Ovo je standardna detekcija SoftICEa i kao ovakva se nalazi u 90% programa koji koriste takozvanu Anti-SoftICE funkciju.

```
function SoftIce95: boolean;
var hfile: Thandle;
begin
 result:=false;
 hFile:=CreateFileA('\\.\SICE', GENERIC_READ or GENERIC_WRITE, FILE_SHARE_READ or
FILE_SHARE_WRITE, nil, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
 if(hfile<>INVALID_HANDLE_VALUE) then
 begin
  CloseHandle(hfile);
  result:=true;
 end;
end;
```

Prisustvo debuggera mozete proveriti uz pomoc malog Delphi programa koji se nalazi ovde …\Casovi\Cas7\Debugger check.exe. Posto znamo kako se zastita implementira u neki program stoga znamo kako i da zaobidjemo ovu proveru. Zastita koju u ovom primeru hocemo da zaobidjemo je provera aktivnog SoftICEa i SoftICEa na NTu. Neznatna je razlika izmedju ove dve provere i one uvek u programima idu zajedno. Jedina razlika izmedju NT i ostalih windowsa je u tome sto je prvi parametar za CreateFileA API \\.\NTICE umesto \\.\SICE. Posto su ovde u pitanju dva string parametra koja se prosledjuju API funkciji stoga se one sigurno u fajlu nalaze kao stringovi i mogu se naci uz pomoc String Reference prozora u W32Dasmu ili pomocu odgovarajuce komande u Olly-u. Dakle provericemo ovu teoriju u praksi. Otvorite ovu "metu" u Ollyju, pronadjite pomenute stringove i postavite break-pointe na njih. Sa F9 startujte program i pritisnite SoftICE NT dugme. Ono sto vidite je sledeca

```
004523C4  /$  53              PUSH EBX
004523C5  |.  33DB            XOR EBX,EBX
004523C7  |.  6A 00           PUSH 0                      ; /hTemplateFile = NULL
004523C9  |.  68 80000000     PUSH 80                     ; |Attributes = NORMAL
004523CE  |.  6A 03           PUSH 3                      ; |Mode = OPEN_EXISTING
004523D0  |.  6A 00           PUSH 0                      ; |pSecurity = NULL
004523D2  |.  6A 03           PUSH 3                      ; |ShareMode = …
004523D4  |.  68 000000C0     PUSH C0000000               ; |Access = …
004523D9  |.  68 F4234500     PUSH Debugger.004523F4      ; |FileName = "\\.\NTSICE"
004523DE  |.  E8 4543FBFF     CALL CreateFileA            ; \CreateFileA
004523E3  |.  83F8 FF         CMP EAX,-1
004523E6  |.  74 08           JE SHORT Debugger.004523F0
004523E8  |.  50              PUSH EAX                    ; /hObject
004523E9  |.  E8 1A43FBF      CALL CloseHandle            ; \CloseHandle
004523EE  |.  B3 01           MOV BL,1
004523F0  |>  8BC3            MOV EAX,EBX
004523F2  |.  5B              POP EBX
004523F3  \.  C3              RET
```

funkcija koja proverava da li je SoftICE aktivan. Ono sto vidimo je jednostavna funkcija koja vraca 1 (true) ako je SICE detektovan ili 0 (false)

ako SICE nije detektovan. Reversing ove funkcije mozete raditi ili na ulazu u nju ili na izlazu iz nje, a mozete je raditi i na sredini. Bez nekog detaljisanja (*dobar deo ovoga je vec vise puta objasnjen u knjizi*) preci cu na moguca resenja ovog problema.

1) Ako zelimo mozemo da promenimo sam pocetak:

```
004523C4  /$  53              PUSH EBX
004523C5  |.  33DB            XOR EBX,EBX
004523C7  |.  5B              POP EBX
004523C8  |.  C3              RET
```

2) Mozemo da izmenimo i samo jedan skok:

```
0045232F  |.  83F8 FF         CMP EAX,-1
00452332  |.  EB 08           JMP SHORT Debugger.0045233C
00452334  |.  50              PUSH EAX                          ; /hObject
00452335  |.  E8 CE43FBFF     CALL<JMP.&kernel32.CloseHandle>   ; \CloseHandle
0045233A  |.  B3 01           MOV BL,1
```

3) Ili mozemo da resetujemo EBX na samom kraju:

```
0045233C     33DB            XOR EBX,EBX
0045233E  |.  5B              POP EBX
0045233F  \.  C3              RET
```

Resenja je mnogo, mozete i sami smisliti neko. Moze se desiti da je string koji trazimo (\\.\SICE) maskiran ili enkriptovan u fajlu i da ne mozemo da ga nadjemo. U tom slucaju mozemo ovo mesto naci pomocu API funkcije CreateFileA koja sluzi za pravljenje "fajla" \\.\SICE. API funkcije mozemo naci tako sto cemo otvoriti Executables Module window (ALT + E) i tu u glavnom exe fajlu pronaci nasu API funkciju pod opcijom View Names (CTRL + N). Ono na sta morate da pazite kada stavljate kondicionalni, importni ili obican break-point na API funkciju je da ce program uvek zastati u dll fajlu u kome se nalazi ta API funkcija (kernel32.dll, user32.dll itd.). Ono sto moramo da uradimo da bismo se vratili u exe koji poziva tu API funkciju je da izvrsimo kod do prve sledece RET funkcije, posle cega cemo se vratiti u ExE kod odakle je pozvana ta funkcija.

Iako je ovo standardan nacin detekcije SICEa ali postoje i druge metode koje se zasnivaju na detekciji aktivnih procesa i otvorenih prozora. Za ovo se koriste druge API funkcije. Neke od njih mogu biti FindWindowA, FindWindowExA, Process32First, Process32Next, MessageBoxA (*prikaz poruke o aktivnom debuggeru*), itd. Ove zastite zaobilazite zavisno od slucaja do slucaja ali najcesce se resavaju pomocu poruke o aktivnom debuggeru koju programeri najcesce ostavljaju u programu. Ovo znaci da nam skoro u 90% preostalih slucajeva programeri preko obicnog message boxa prikazuju poruku da program nece biti startovan jer je neki od debuggera aktivan.

**NAPOMENA:** Primeticete da sam u prvom primeru reversinga SICE funkcije izmenio 3 i 4 liniju u POP EBX i RET. Ovo RET je logicno jer ono sto se desava je jednostavno vracanje iz CALLa a ono sto moramo uraditi pre vracanja je da vratimo ulazne parametre funkcije pomocu komande POP (*ulazni parametri se ne moraju uvek nalaziti pre CALLa nego se mogu naci i unutar samog CALLa na njegovom pocetku*). Posto je jedini ulazni parametar EBX vracamo samo njega. Nema preterane potrebe da ovo pamtite, ovo se moze jednostavno prepisati sa kraja CALLa ali je bitno da imate na umu prilikom menjanja koda.

# Windows check debugger API

Evo jos jednog primera za Delphi programere. Ovaj deo koda je zasebna funkcija koja se koristi za detekciju bilo kog aktivnog debuggera. Ova funkcija ce detektovati da li je program startovan uz pomoc nekog debuggera ili ne. Ovo je bar koliko je meni poznato nedokumentovana API funkcija. Slucajno sam naisao na nju prilikom reversovanja programa koji je bio pakovan i zasticen AsProtectom. Posle dosta razmatranja dosao sam do zakljucka da AsProtect koristi windowsovu API funkciju IsDebuggerPresent koja se nalazi u kernel32.dll-u koja nema ulaznih parametara a vraca kao rezultat 1 ili 0 u zavisnosti od toga da li je debugger detektovan. Kada se sve ovo sroci u jednu Delphi funkciju to izgleda upravo ovako:

```
function IsDebuggerPresent: Integer; stdcall;  external 'kernel32.dll' name
'IsDebuggerPresent';
```

Kao sto je za SICE objasnjeno reversing ove funkcije se radi veoma jednostavno. Otvorite isti primer koji smo koristili u proslom delu poglavlja. Ako postavimo break-point na IsDebuggerPresent pokrenemo program i pritisnemo dugme Test Debugger API zavrsicemo u kernel32.dll-u a posle izvrsenja RET komande naci cemo se ovde:

```
0045229A    8BC0            MOV EAX,EAX
0045229C  . E8 F3FFFFFF     CALL IsDebuggerPresent  <- Ovde smo
004522A1  . 48              DEC EAX
004522A2  . 75 16           JNZ SHORT Debugger.004522BA
```

Ako pogledamo sadrzaj EAXa videcemo da on sadrzi broj 1 jer smo program pokrenuli uz pomoc debuggera (Ollyja). Ovo mozemo promeniti na veoma lak nacin tako da program uvek “misli” da nije startovan pomocu debuggera.

```
0045229A  .  8BC0           MOV EAX,EAX
0045229C  .  B8 00000000    MOV EAX,0
004522A1  . 48              DEC EAX
004522A2  . 75 16           JNZ SHORT Debugger.004522BA
```

Sta se ovde desava ? Umesto poziva funkciji IsDebuggerPresent mi cemo samo postaviti da je EAX uvek jednak nula (MOV EAX,0 ili XOR EAX,EAX isto je). Zasto nula ? Zato sto imamo kondicioni skok JNZ na adresi 004522A2 koji ce se izvrsiti samo ako je EAX jednak FFFFFFFF (-1) a pre toga imamo jednu DEC EAX komandu (EAX = EAX – 1) pa stoga pre ove dve komande EAX uvek mora biti jednak nula.

Ovo je jedna zaista obicna i doista jednostavna zastita koja se lako implementira a jos lakse zaobilazi. Ali i bez obzira na to moze se sresti i u komercijalnim zastitama (*AsProtect*) pa je korisno znati i ovaj API.

# Memory modification check

Ovo je jedna retko (*skoro nikada*) koriscena tehnika. Ova tehnika je jako korisna kao zastita od raznih vrsta loadera, memory patchera ili bilo kakve manipulacije memorijom. Ja licno nisam sreo ovu tehniku u "metama" ali nije na odmet znati kako dodatno zastititi vasu aplikaciju. Posebno napisana aplikacija koja se nalazi ovde …\Casovi\Cas7\Memory Checker.exe. Malo uputstvo pre nego sto pocnemo sa reversingom ove aplikacije. U samom programu cete videti 00 koje ce se posle par sekundi promeniti u broj 1727. Ovaj broj predstavlja checksum vrednost zasticenog dela koda. Svakih par sekundi odredjeni deo koda (*memorije*) se proverava i ako mu je checksum razlicit od 1727 onda ce se umesto ove poruke pojaviti poruka o modifikaciji memorije. Ova poruka ce se pojaviti samo ako modifikacija memorije ne sadrzi ni jednu NOP komandu. Ako je deo memorije NOPovan program ce se sam zatvoriti. Posmatrani deo memorije je segment od 20 bajtova koji sluzi za prikazivanje NAG poruke svakih 20 sekundi. Ovaj deo koda odnosno memorije moramo da promenimo tako da se ova NAG poruka ne pojavljuje a da program i dalje misli da je sve u redu sa sadrzajem memorije. Ovaj zadatak iako izgleda veoma komplikovan nije toliko tezak.

Uredu, pa da pocnemo. Otvorite ovu metu pomocu Ollyja. Ono sto mozete da pokusate je da nadjete gde se to pojavljuje poruka o menjanju memorije. Da bismo saznali kako ova poruka izgleda moramo da prvo ubijemo onu NAG poruku. Nadjimo je i postavimo jednostavnu RET komandu na adrese 004507F4 i 004507F5 (*zapamtite, ne sme biti ni jedna NOP komanda*).

```
004507F4    6A 40           PUSH 40 <- Izmeniti u RET
004507F6    68 08084500     PUSH Memory_C.00450808          ; ASCII "Error"
004507FB    68 10084500     PUSH Memory_C.00450810          ; ASCII "This NAG will
be shown every 20 seconds, KILL IT !!!"
00450800    6A 00           PUSH 0
00450802    E8 45FEFFFF     CALL <JMP.&user32.MessageBoxA>
00450807  . C3              RET
```

Program ce posle par sekundi izbaciti poruku Error – Modified. Sada mozemo da potrazimo ovaj string u fajlu. Naci cemo ga ovde:

```
Text strings referenced in Memory_C:CODE, item 1825
 Address=004507E0
 Disassembly=ASCII "Error - Modified"
```

Duplim klikom na ovaj red zavrsicemo ovde:

```
004507D8  . FFFFFFFF        DD FFFFFFFF
004507DC  . 10000000        DD 00000010
004507E0  . 45 72 72 6F 7>ASCII "Error - Modified"
004507F0  . 00              ASCII 0
004507F1    00              DB 00
```

Posto ovo ne predstavlja nikakav kod nego samo referencu ka stringu (*ovo se koristi tako sto se kada zelimo da se ovaj string pojavi na ekranu funkciji zaduzenoj za pojavljivanje ne prosledjujemo string nego adresu na kojoj se on nalazi*), selektovacemo adresu 004507E0 i postavicemo *desno dugme -> Breakpoint -> Memory, on access* breakpoint na nju. Posle malo cekanja zavrsicemo ovde usled izvrsavanja breakpointa:

```
00404687  |. 8B1F           MOV EBX,DWORD PTR DS:[EDI]
00404689  |. 38D9           CMP CL,BL
0040468B  |. 75 41          JNZ SHORT Memory_C.004046CE
```

Ono sto sada moramo da uradimo je da izvrsimo sav kod dok se ne vratimo u deo koda koji je pristupio ovoj memoriji. To u principu znaci da treba (u

ovom slucaju) da se vratimo iz 2 CALLa to jest da izvrsimo dve RET komande. Kako ovo znamo? Ne znamo, nego se jednostavno vracamo iz jednog po jednog CALLa sa F8 dok ne dodjemo do koda koji izgleda kao funkcija za prikazivanje ovog stringa na ekranu. Za ovo je potrebno malo vezbe. Kada se konacno vratimo iz drugog CALLa videcemo ovo:

```
0045077E     /74 13             JE SHORT Memory_C.00450793
00450780  |. |8B45 FC           MOV EAX,DWORD PTR SS:[EBP-4]
00450783  |. |8B80 04030000     MOV EAX,DWORD PTR DS:[EAX+304]
00450789 |.  |BA E0074500       MOV EDX,Memory_C.004507E0                ;  ASCII "Error -
Modified"
0045078E  |. |E8 C1F1FDFF       CALL Memory_C.0042F954
00450793  |> \33C0              XOR EAX,EAX
00450795  |.  5A                POP EDX
```

Ovo vec izgleda kao deo koda koji sluzi za prikazivanje poruke na ekranu. Ono sto treba da uradimo da se ova poruka nikada ne prikaze na ekranu je da skok na adresi 0045077E promenimo iz JE u JMP i to je to. Program ce uspesno umesto poruke o modifikaciji memorije prikazivati poruku o novom memory checksumu. Sada mozemo snimiti ove promene pritiskom na *desno dugme -> Copy to executable -> All modifications -> Save...*

**Napomena:**

Ovaj problem moze biti resen i mnogo jednostavnije ako znate koji je to Win32API zaduzen za citanje memorije. Taj API je ReadProcessMemory.

**Vezba:**

Ova nasa modifikacija ce raditi samo u slucaju da smo promenili NAG pomocu RET komande. Probajte isto ovo samo umesto RET komande unesite RET pa NOP. Naravno program ce se zatvoriti. Na vama je da otkrijete gde se vrsi provera da li je neka adresa NOPovana i da je zaobidjete.

**Resenje:**

Promenite skok na adresi 0045070C iz JNZ (*JNE*) u JMP.

# Reversing CRC32 checks

Jedna od veoma cesto koriscenih anti-cracking tehnika je provera CRC checksuma nekog fajla. Sta je to CRC? CRC je tehnika koja omogucava proveru sadrzine nekog fajla. Ona jednostavno uzme neki fajl, napravi njegovu "sliku" i to pretvori u osmocifreni heksadecimalni broj koji predstavlja unikatnu "sliku" posmatranog fajla. Posto CRC zavisi od sadrzine fajla to znaci da razliciti fajlovi imaju razlicite CRCove. CRC je broj koji se nalazi u rasponu od 00000000 pa do FFFFFFFF. Ovo ujedno znaci da ako promenimo neki fajl patchovanjem promenicemo mu i CRC pa ce sam program jednostavnom proverom znati da je modifikovan. Imajte na umu da ako je ova zastita dobro implementirana u program bice vam potrebno izuzetno mnogo vremena i zivaca da pronadjete gde se to tacno nalazi CRC provera u programu koji reversujete. Primer koji sam ja za ovu priliku napravio je veoma lak i necete imati nikakvih problema da ga reversujete. Primer se nalazi ovde: …\Casovi\Cas7\CRC\crc32.exe, a sadrzi i pomocni fajl crc32table.txt koji u stvari cuva enkriptovani CRC32 checksum crc32.exe fajla. Primecujete da se u  txt fajlu cuva CRC checksum koji program koristi da ga uporedi sa svojim CRCom i pitate se zasto je to ovako? Zasto program ne cuva svoj CRC u samom sebi? Pa odgovor je veoma jednostavan: Ako bi program cuvao svoj CRC u samom sebi CRC bi se uvek menjao, ako kompajlujemo program koji cuva u sebi jedan CRC, zbog tog broja CRC bi se promenio, a ako bismo taj broj promenili u novi CRC, onda bi se CRC opet promenio pa se zbog toga CRC cuva u posebnom fajlu ili u Registryju. Ako startujete ovaj primer videcete da se pojavljuje standardan NAG ekran pre samog startovanja i zatvaranja aplikacije. Ono sto moramo da promenimo u ovom primeru je da ubijemo ovaj NAG. Ovo je veoma lako i to, ako ste citali knjigu lepo sa razumevanjem, morate da znate da uradite sami. Kada ovo uradite snimite fajl kao crc32.exe a stari reimenujte u crc32_1.exe. Ako sada startujete crc32.exe fajl prikazace se poruka o pogresnom CRCu. Mozete i nju naci i promeniti. Ovo je isto lako:

```
00451104  |.  E8 DB3AFBFF     CALL crc32.00404BE4
00451109  |.  74 40           JE SHORT crc32.0045114B
0045110B  |.  6A 40           PUSH 40;  MB_OK|MB_ICONASTERISK|MB_APPLMODAL
0045110D  |.  68 C0114500     PUSH crc32.004511C0               ; |Title = "Error"
```

samo treba promeniti skok na adresi 00451109 iz JE u JMP i to je to. Kao sto rekoh ovo je veoma laka "zastita", dok ce vas lepo implementirana CRC provera posteno namuciti. Pogledajmo malo ovu CRC zastitu koja pocinje na adresi 00450ED0. Jednostavnim pregledom stringova videcemo da program otvara fajl crc32table.txt i kopira crc32.exe u bak.bak da bi najverovatnije na novom bak.bak fajlu proverio CRC. Ovo je zanimljivo posto kako vidimo program koristi specificno ime crc32.exe a ne neku promenljivu u slucaju da se ime exe fajla promeni. I ovo mozemo da iskoristimo. Iskopirajmo crc32.exe u novi fajl new.exe i u new.exe fajlu samo ubijmo NAG. Ako sada startujemo new.exe videcemo da nema NAGa ali da nema ni poruke o pogresnom CRCu jer se CRC32 proverava na kopiji crc32.exe fajla a ne na samom sebi, u ovom slucaju new.exe-u. Naravno ako u praksi nadjete ovako glupu zastitu posaljite mi email programera koji ju je napisao da ga castim pivom :) A posto sam ovako glupu zastitu napisao ja sam, mislim da cu

morati sam sebe da castim pivom :) Bilo kako bilo, ova CRC zastita definitivno zasluzuje nagradu sa +Fravia sajta:


*"Da da znam, zasluzio sam nagradu posteno :)" – Ap0x*

Salu na stranu ono (*pametno*) sto ce ovaj primer da vas nauci je kako da dekriptujete onaj crc32table.txt fajl i tako razbijete CRC zastitu. Otvorite originalni crc32.exe fajl u Ollyju postavite obican break-point na adresu 00450ED0. Sa F9 startujte program i sacekajte da se Olly zaustavi na ovom break-pointu. Sa F8 polako analizirajte kod. Na adresi 00450ED8 imamo stvarno dugacak loop (*cak 41h puta se ponavlja*) koji ne radi nista bitno za nas. Idemo dalje... Kada stignemo do adrese 00450F28 videcemo da se u EAX smesta cela putanja do crc32table.txt fajla. Ovo moze biti zanimljivo. Sa F8 idemo dalje kroz kod. Nista bitno se ne desava sve dok ne dodjemo do adrese 00451037 kada program kopira samog sebe u bak.bak fajl. Nista bitno za enkripciju, pa nastavljamo dalje. Stigli smo do 004510C9 i sada imamo neki loop koji se izvrsava nekoliko puta. Ovaj loop mora biti jako zanimljiv posto se nalazi odmah iznad poruke o pogresnom CRCu. Da vidimo sta se desava u njemu.

```
004510CE  |> /8D45 FC         /LEA EAX,DWORD PTR SS:[EBP-4]
004510D1  |. |E8 1A3CFBFF     |CALL crc32.00404CF0
004510D6  |. |8B55 FC         |MOV EDX,DWORD PTR SS:[EBP-4]
004510D9  |. |8A541A FF       |MOV DL,BYTE PTR DS:[EDX+EBX-1]
004510DD  |. |80F2 2C         |XOR DL,2C
004510E0  |. |885418 FF       |MOV BYTE PTR DS:[EAX+EBX-1],DL
004510E4  |. |43              |INC EBX
004510E5  |. |4E              |DEC ESI
004510E6  |.^\75 E6           \JNZ SHORT crc32.004510CE
```

Prodjimo par puta kroz njega i videcemo da se uzima jedno po jedno slovo iz nekog stringa, mozda onog koji se nalazi u samom crc32table.txt fajlu, i da se njegova ASCII vrednost xoruje sa 2Ch ili sa 44 decimalno. Kada se ovaj loop zavrsi, to jest kada stignemo do adrese 004510E8 onda cemo u EAXu imati string **EAX 008B213C ASCII "7CC85525"** koji odgovara dekriptovanom CRCu iz fajla crc32table.txt. Ako je taj CRC jednak CRCu fajla bak.bak onda fajl nije modifikovan. Ono sto nas interesuje je kako da reversujemo enkripciju pa da dobijemo drugi enkriptovani CRC, za izmenjeni NAG free program, koji bi smestili u crc32table.txt fajl. Ovo mozemo resiti na dva nacina.

**Prvi nacin:**

Pretpostavimo za trenutak da ne znamo kako program generise enkriptovani string koji sadrzi CRC vrednost crc32.exe fajla. Ono sto cemo da uradimo je da otvorimo crc32table.txt fajl i da iz njega iskopiramo onaj string. Jedino potrebno podesavanje Notepada je da promenite font u Terminal. **[Hint:** *Format -> Font -> Terminal -> Regular***]**, ono sto nam je jos potrebno je ASCII tablica koju mozete preuzeti sa www.asciitable.com (*opciono ali veoma korisno*) i spremni smo da pocnemo.

Prvo sto treba da uradimo je da da sve simbole iz crc32table.txt pretvorimo u ASCII kodove. U ovom slucaju je ovo jako tesko izvesti, jer su slova prilicno ne standardna, pa cemo napraviti program koji ce to uraditi za nas. Kod Visual Basic programa koji to radi bi izgledao bas ovako:

```
Text2.Text = ""
for i = 1 to Len(Text1.Text)
 Text2.Text = Text2.Text & Asc(Mid$(Text1.Text,i,1))
 if i < Len(Text1.Text) then Text2.Text = Text2.Text + ","
next i
```

Text2 sadrzi rezultat a Text1 sadrzi string za koji se racuna ASCII. Gotov primer se nalazi u istom folderu pod imenom ascii.exe. Otvorimo ovaj program i u njega unesimo string iz fajla crc32table.txt. Ono sto ce program pokazati je:

**27,111,111,20,25,25,30,25**

Sada treba da patchujemo program crc32.exe i snimimo patchovan program kao new.exe. Patchovacemo 00450E88 adresu u ovo:

```
00450E88    C3        RET
00450E89    90        NOP
```

Reimenujte originalni crc32.exe u crc32_1.exe, a new.exe u crc32.exe. Sada cemo pomocu getCRC32.exe dobiti CRC vrednost fajla crc32.exe (*new.exe fajla*) fajla, dobicemo novi CRC koji iznosi 0D2541F9 (*zapamtite CRC mora imati 8 cifara*). Sada cemo i ovaj CRC pretvoriti u ASCII pomocu ascii.exe-a. Dobicemo ovo:

**48,68,50,53,52,49,70,57**

Pored ovog nam je poteban i ASCII kod originalnog crc32.exe fajla, koji se trenutno zove crc32_1.exe. Posto znamo da ovaj CRC iznosi 7CC85525, i njega cemo pretvoriti u ASCII. To izgleda ovako:

**55,67,67,56,53,53,50,53**

Sada treba da otkrijemo kako treba da enkriptujemo novu CRC vrednost - 0D2541F9. Za ovo su nam potrebna dva podatka. ASCII enkriptovanog CRCa i ASCII samog CRCa. Ovi podaci su:

**27,111,111,20,25,25,30,25 i 55,67,67,56,53,53,50,53**

Ono sto je bitno da se svaka prosta a i komplikovana enkripcija zasniva na obicnom XOR-ovanju. Ovo znaci da je ASCII kod jednog karaktera enkriptovan (xorovan) nekom vrednoscu i tako je dobijena nova ASCII vrednost. Mi treba da saznamo kojom vrednoscu je XORovan svaki karakter originalnog CRCa - 7CC85525 kako bi bio dobijen enkriptovani string koji se nalazi u crc32table.txt fajlu. Ovo cemo dobiti reversnim xorovanjem, to jest uporedjivanjem enkriptovanog stringa i neenkriptovanog originalnog CRCa crc32.exe fajla. Znaci **27 xor ?? = 55**, **111 xor ?? = 67**,... Ono sto je bitno kod xor funkcije je da je ona reverzibilna to jest da je **x xor y = z** ali i da je **z xor x = y i z xor y = x,** i tako cemo otkriti vrednost ??. Znaci **55 xor 27 = 44, 67 xor 111 = 44**,... i tako smo dobili magicnu xor vrednost za svako slovo CRCa. Sada treba samo da enkriptujemo novi CRC pomocu vrednosti 44, i da dobijenu ASCII vrednost pretvorenu u slova i simbole snimimo u crc32table.txt fajl. Ovo cemo uraditi pomocu novog programa koji cemo sami napisati. Evo kako to izgleda u Visual Basicu:

```
 Text2.Text = ""
for i = 1 to Len(Text1.Text)
 Text2.Text = Text2.Text & Chr(Asc(Mid$(Text1.Text,i,1)) xor 44)
next i
 open "crc32table.txt" for output as #1
  print #1, Text2.Text
 close #1
```

Gotov primer se nalazi u istom folderu pod imenom XORascii.exe. Otvorimo ovaj program, unesimo novi CRC u njega (*0D2541F9*) i program ce sam generisati novi crc32table.txt fajl. Ako probamo da startujemo crc32.exe sada videcemo da se poruka o modifikaciji ne menja, to jest da je jos tu. Mora da smo negde pogresili !!! Probajmo samo da umesto 0D2541F9 unesemo D2541F9 u XORascii.exe i startujmo program sada. Radi, znaci ipak je sve ok. Ova tehnika moze biti "malo" komplikovana ali je odlicna ako se svaki karakter XORuje drugacijim brojem, tako da ovu vrstu dekripcije nije lose znati. Istu ovakvu enkripciju passworda mozete naci i u programu Trillian, pa za vezbu mozete napraviti program koji dekriptuje njegov password.

**Drugi nacin:**

Drugi nacin se zasniva na reversovanju samog loopa za dekripciju enkriptovanog CRCa iz crc32table.txt fajla. Ovo je veoma lako (*lakse od prvog nacina*) i dacu vam odmah source za Visual Basic:

```
 Text2.Text = "" `izlazni text box
for i = 1 to Len(Text1.Text) `ulazni text box
 Text2.Text = Text2.Text & Chr(Asc(Mid$(Text1.Text,i,1)) xor 44)
next i
 open "crc32table.txt" for output as #1
  print #1, Text2.Text
 close #1
```

i za Delphi:

```
var
 wFile:TextFile;
 crc:string;
 i:integer;
begin
 crc := '';
for i := 1 to Length(Edit1.Text) do begin //ulazni text box
 crc := crc + Chr(ORD(Edit1.Text[i]) xor 44);
end;
 AssignFile(wFile,'crc32table.txt');
  writeln(wFile,crc);
 CloseFile(wFile);
end;
```

# "Not Getting Caught :)" - *Exerecise*

Ako niste primetili pored primera …\Casovi\Cas7\Debugger check.exe postoji i primer …\Casovi\Cas7\Debugger Check2.exe. Ovaj drugi je napravljen da bude vezba za izbegavanje detekcije debuggera. Ako ste pazljivo citali prvi i drugi deo ovog poglavlja bez problema cete zaobici sve ove zastite i resicete uspesno i ovu vezbu. Pokusajte sami da resite ovaj problem a ako ne budete mogli vratite se ovde da utvrdite gradivo.

**Resenje:**

Ako citate ovaj deo pretpostavljam da niste sami mogli da se oslobodite provere debuggera u primeru. Pretpostavljam da ste krenuli logickim redosledom tako sto ste zapamtili poruku o detektovanom debuggeru i probali ste da je potrazite u fajlu. Ali, desilo se nesto jako cudno, nema takve niti bilo kakve slicne poruke u fajlu :) Ono sto sam namerno uradio za ovu vezbu je enkripcija stringova unutar samog fajla. To znaci da su stringovi i dalje tu samo nisu isti u fajlu i u memoriji. Ovo znaci da se stringovi pre upotrebe dekriptuju u svoj pravi oblik. Naravno vi ne znate kakav sam ja algoritam koristio za enkripciju stringova tako da nikako ne mozete znati kako koja poruka izgleda enkriptovana. Ono sto cemo mi uraditi je veoma jednostavno, koristicemo drugi metod za pronalazak mesta gde se proverava da li je program otvoren uz pomoc debuggera. Trazicemo ova mesta pomocu API poziva. Otvoricemo ovaj program uz pomoc Ollyja i postavicemo break-point na MessageBoxA API posto se pomocu njega prikazuje poruka o aktivnom debuggeru. Ovo se radi u Executable modulima (*ALT + E*), pod opcijom View Names (*CTRL + N*); Toggle break-point on import, ako ste zaboravili. Sada pokrenimo ovaj primer i sacekajmo da program dodje do break-pointa. Uklonimo ovaj break-point pritiskom na F2. Posto se nalazimo u dll fajlu sa F8 cemo izvrsavati kod polako dok ne dodjemo do RET komande i dok se i ona ne izvrsi. Sada, kada smo se vratili iz funkcije, nalazimo se na adresi 00450D7F. Ovo izgleda kao funkcija koja prikazuje poruku o prisustvu debuggera. Ovo cemo ukloniti jednostavnim postavljanjem obicne RET komande na adresi 00450D44. Resetujmo program sa CTRL + F2 i izmenimo adresu 00450D44 u RET (C3). Startujmo program i videcemo da se poruka opet pojavila. Hmmm. Resetujmo program opet, izmenimo adresu 00450D44 i postavimo opet break-point na MessageBoxA. Program je opet stao u nekom dll-u i posle izvrsavanja RET komande vraticemo se u deo koda koji poziva MessageBoxA API. Sada se nalazimo na adresi 00450847. Hmmm ovaj CALL izgleda isto kao onaj koji pocinje na adresi 00450D44. Da li postoje dve vrste provere ? Zapamtimo dve adrese 00450D44 i 0045080C posto su one pocetne adrese ovih CALLova. Resetujmo program sa CTRL + F2 i pomocu Go To adress opcije (*prvo dugme na levo pored dugmeta L u toolbaru*) otici cemo na obe ove adrese. Kada odemo na prvu i selektujemo je videcemo detalje o njoj:

**Local call from 00450DED**

Ovo znaci da se ovaj CALL poziva samo sa adrese 00450DED. A kada odemo na adresu 0045080C videcemo sledece:

```
Local calls from 004509BA, 00450A13, 00450A45, 00450A67, 00450A89, 00450AAB, 00450ACD, 00450AEF, 00450B11
```

to jest da se on poziva sa vise adresa. Na svaku od ovih adresa mozemo da odemo pritiskom da desno dugme na ovoj liniji i selekcijom na koju to adresu zelimo da odemo. Ako odemo samo na neke od tih adresa bice nam jasno sta se tu desava. Otici cemo samo na adrese 00450DED i na adresu 00450B11. Na adresi 00450DED imamo:

```
00450DC3  .  27 20 37 25 2 ASCII "' 7%%'"
00450DC9  .  30 62 24 2D 3 ASCII "0b$-7,&nb#22.+!#"
00450DD9  .  36 2B 2D 2C 6 ASCII "6+-,b5+..b',&l",0
00450DE8  .  E8 87F9FFFF   CALL IsDebuggerPresent   ; [IsDebuggerPresent]
00450DED  .  E8 52FFFFFF   CALL Debugger.00450D44
00450DF2  .  C3            RET
```

Ovaj prvi kod prikazuje poziv poruci o detektovanom debuggeru odmah posle CALLa IsDebuggerPresent APIja. Ok sada znamo da program prisustvo debuggera proverava uz pomoc API funkcije. Ovo je prvi deo provere. Drugi deo provere se nalazi na adresi 00450B11. Na toj adresi imamo ovo:

```
00450B09  |.  50             PUSH EAX                 ; /Title
00450B0A  |.  6A 00          PUSH 0                   ; |Class = 0
00450B0C  |.  E8 0F62FBFF    CALL FindWindowA         ; \FindWindowA
00450B11  |.  E8 F6FCFFFF    CALL Debugger.0045080C
00450B16  |.  33C0           XOR EAX,EAX
```

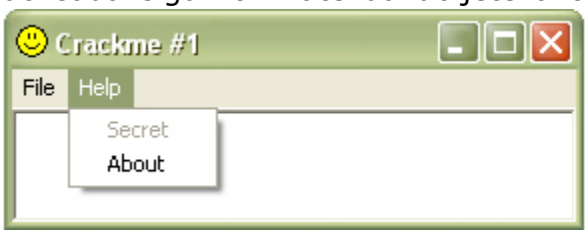
odakle zakljucujemo da je druga provera FindWindowA. Resenje obe ove provere je izuzetno lako. Mozemo jednostavno staviti RET komande na adresama 00450D44, 0045080C i problem je resen. Ako zelimo da vidimo koje to prozore ovaj program trazi mozemo jednostavno postaviti break-pointe na sve CALLove ka FindWindowA APIju. Imajte na umu da se break-pointi ne postavljaju na sam CALL FindWindowA nego na PUSH EAX pre toga posto ce se tada u EAXu nalaziti naslov (title) prozora koji se trazi.

# 08 Cracking it...

Ovo poglavlje sadrzi osnove ukljucivanja iskljucenih menija i dugmadi, pronalazenje izgubljenih passworda, pobedjivanje time-trial programa, crackovanje .dll fajlova. Ova tehnika je podcenjena zbog postojanja velikog broja raznih programa koji ovo mogu uraditi za vas brzo i jednostavno. Nas jednostavno zanima kako se ovo radi rucno i kako bismo to mogli izvesti u nedostatku alata poput ReSHackera.

# ReEnable buttons - *ASM*

Ovaj deo poglavlja ce vas nauciti kako da ukljucite neke iskljucene opcije u programima. Ovo je narocito korisno ako je neki program zasticen bas na ovaj nacin. Na primer, moze se desiti u nekoj "meti" da je recimo Save opcija zatamnljena i da se ne moze ukljuciti. Naravno vecina vas je naucila da ovakve probleme resava preko raznih ReSHackera i slicnih alata. Nemam ja nista protiv takvog pristupa ali morate i sami priznati da je skroz lame. Primer koji sam specijalno za potrebe ovog poglavlja "iskopao" na netu pristupa ovoj problematici na sasvim drugaciji nacin. Naime fajl crackme.exe koji se nalazi u folderu …\Casovi\Cas8\ je specifican po tome sto ne koristi resurse nego sam programski pomocu Win32Apija pravi elemente koji se nalaze u njemu. Kao sto vidite ovde ReSHacker ne pomaze :) Naravno NAG do sada sigurno znate da ubijete a ono sto je predmet razmatranja ovog poglavlja je kako ukljuciti iskljucen meni. Posto se ovaj "tajni" meni pravi preko API komandi iskoristicemo to da pronadjemo mesto odakle se poziva procedura koja pravi ovaj meni. Otvoricemo ovaj program u Ollyju i potrazicemo sve string reference ka stringu Secret. Evo spiska svih referenci koje se nalaze u tom crackmeu.




**Text strings referenced in crackme:.text**

| Address | Disassembly | Text string |
|---|---|---|
| 0040104E | PUSH crackme.0040315C | ASCII "Secret Unrevealed!" |
| .... | | |
| 004010F2 | PUSH crackme.00403018 | ASCII "&Secret" |
| 00401103 | PUSH crackme.00403010 | ASCII "&About" |
| 00401112 | PUSH crackme.00403008 | ASCII "&Help" |
| .... | | |
| 004011D7 | PUSH crackme.00403190 | ASCII "Crackme #1" |
| 004011F0 | PUSH crackme.004031C0 | ASCII "Error!" |
| 004011F5 | PUSH crackme.00403170 | ASCII "Aishh..msgbox takleh create!" |

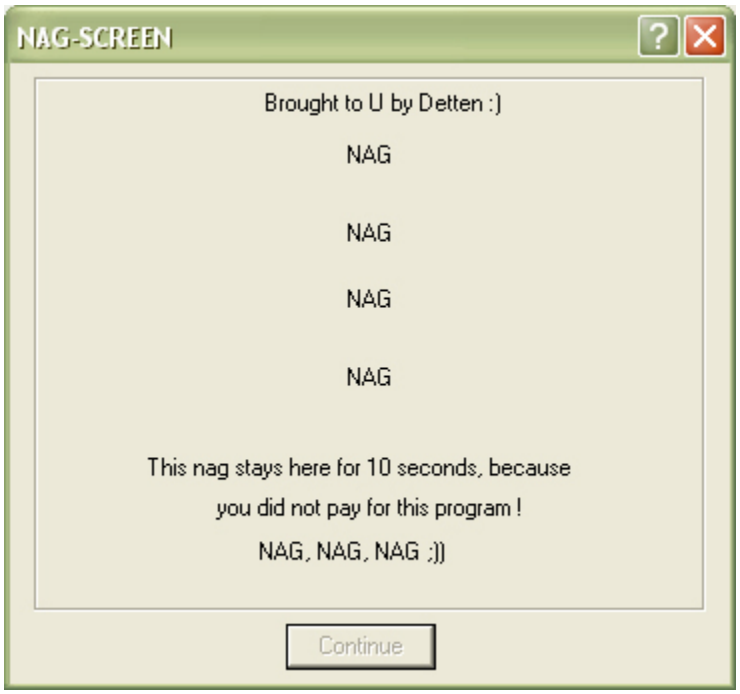
Kao sto vidimo samo zeleni red sadrzi string koji je istovetan onom koji vidimo u crackmeu. Ne dajte da vas zbuni ono & ispred stringa. Kada su meniji u pitanju ono & oznacava da je prvo slovo stringa podvuceno. Ako 2x kliknete na taj red docicete dovde:

```
.  FFD7        CALL EDI                  CreatePopupMenu
.  68 18304000 PUSH crackme.00403018     pItem = "&Secret"
.  8BF8        MOV EDI,EAX
.  68 2B230000 PUSH 232B                 ItemID = 232B (9003.)
.  6A 01       PUSH 1                    Flags = MF_BYCOMMAND|MF_GRAYED|MF_STRING
.  57          PUSH EDI                  hMenu
.  FFD6        CALL ESI                  AppendMenuA
.  68 10304000 PUSH crackme.00403010     pItem = "&About"
.  68 2A230000 PUSH 232A                 ItemID = 232A (9002.)
.  6A 00       PUSH 0                    Flags = MF_BYCOMMAND|MF_ENABLED|MF_STRING
.  57          PUSH EDI                  hMenu
.  FFD6        CALL ESI                  AppendMenuA
```

Kao sto primecujete osim razlike u prvoj PUSH komandi, koja nam je nebitna jer predstavlja samo ID menija, imamo razliku i u drugoj PUSH komandi. Kod Secret menija imamo PUSH 1 a on je iskljucen a kod About menija imamo PUSH 0 a on je ukljucen. Dolazimo do zakljucka treba samo da promenimo PUSH 1 u PUSH 0 da bi i Secret meni bio ukljucen. Kao sto vidite ovaj problem je bilo veoma lako resiti.

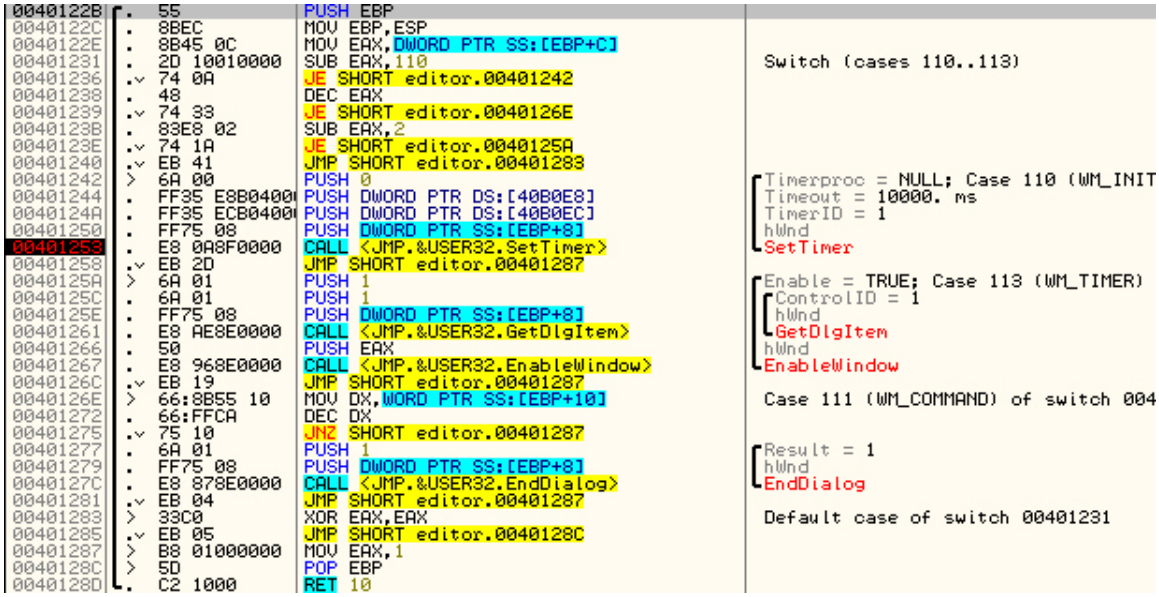
# ReEnable buttons - API

Ovaj deo poglavlja ce vas nauciti kako da pronadjete tacno mesto odakle se iskljucuju / ukljucuju opcije. Naravno posto je Microsoft-ova dokumentacija veoma obimna spisak svih raspolozivih API funkcija nije ni malo lako prelistati i naci ono sto vam treba. Naravno kao i svaki drugi veoma nestrpljivi cracker odlucio sam da batalim API reference i da nadjem neki prakticni primer kojim cu lako otkriti kako se to iskljucuju / ukljucuju dugmici, meniji, formovi i sl... Slucajno sam pri ruci imao savrsen primer za ovaj problem. Primer …\Casovi\Cas8\editor.exe izgleda ovako:




Kao sto se vidi na ovoj slici u pitanju je jos jedan NAG ekran koji treba ukloniti. Jedina prednost ovog primera nad svim drugima je to sto on ima iskljuceno dugme koje se posle deset sekundi ukljuci propustajuci vas dalje u program. Ovo je bitna karakteristika koju cemo iskoristiti da bismo nasli koja je to API funkcija zaduzena za ukljucivanje tog dugmeta. Da bismo ovo uradili mozemo da koristimo dva nacina: laksi nacin i moj nacin :) Jedini problem sa laksim nacinom je da on pretpostavlja da znate API funkcije koje se koriste za simuliranje tajmera. To jest da znamo kako to program napravi pauzu od 10 sekundi, a ako vam ovo nije poznato mozete uvek da koristite moj nacin :)

**Nacin 1:**
Laksi nacin pronalaska mesta odakle se poziva funkcija koja ukljucuje iskljuceno dugme je da postavimo break-point na svaku referencu ka API funkcijama koje sluze za pravljenje tajmera i / ili pauziranje programa. Ovo rade dve API funkcije: **kernel32.Sleep** i **user32.SetTimer** pa cemo pronaci koja se od ove dve koristi u ovom primeru. Preko Ollyja pogledajte koji se to importi nalaze u module -> names prozoru. Od ove dve spomenute funkcije pronaci cete samo API SetTimer. Naravno da se vreme moze pronaci i preko drugih APIja kao sto su GetLocalTime, GetSystemTime i slicnih ali je najbrze preko SetTimer APIja. Dakle postavimo break-point on every reference na ovaj API i onda startujmo program. Program ce zastati na adresi 00401253 odakle se poziva SetTimer API. Ako odskrolujemo malo gore i pregledamo ceo ovaj CALL videcemo kako se to startuje timer i mozda cemo naci API funkciju koja ukljucuje dugme. Ono sto vidimo prikazano je na sledecoj slici:




Dakle na adresi 00401253 se poziva t.j. kreira tajmer sa IDom 1 i vremenom ponavljanja od 10000 ms odnosno 10 sekundi. Ako analiziramo malo ovaj deo koda primeticemo da posto se kreira tajmer, izvrsava se neki JMP skok koji preskace par API funkcija (*GetDlgItem, EnableWindow, EndDialog*). Primeticemo da se i iznad API funkcije za kreiranje tajmera nalazi par kondicionih JE skokova i je jedan nekondicioni JMP skok. Mozemo da postavimo break-pointe na njih i da startujemo program sa F9 kada dodje do njih. Tada cemo primetiti da se tajmer kreira samo jednom a da se ostali skokovi ne izvrsavaju dok ne prodje 10 sekundi, kada se izvrsava skok na adresi 0040123E koji vodi do adrese 0040125A koja prvo vraca handle (*adresu kontrole koja se trazi u memoriji*) neke kontrole uz pomoc GetDlgItem API da bi se toj istoj kontroli poslala komanda Enable = True preko APIja EnableWindow. Izgleda da se API EnableWindow koristi za ukljucivanje iskljucenih dugmadi, menija i sl. I posle malo mucenja pronasli smo API funkciju koja se koristi za ukljucivanje dugmica. Ona glasi ovako:

**EnableWindow(hwnd,1)**

gde umesto hwnd ide handle dugmeta ili nekog drugog objekta u prozoru a umesto 1 ide 1 za ukljucivanje i 0 za iskljucivanje. Hwnd dugmeta mozemo dobiti pomocu druge API funkcije GetDlgItem koja se poziva ovako:

**GetDlgItem(hwnd,ControlID)**

gde je hwnd handle prozora u kojem se nalazi dugme a control ID je Id koji smo dodelili nasem dugmetu u .res fajlu, u ovom slucaju ControlID je 1 jer je tako Daten (*autor programa*) deklarisao ID u .res fajlu. Naravno ako ste za razliku od mene procitali Microsoft-ove API reference ovo ste znali i bez primene reversnog inzenjeringa. Ja sam ovo morao da naucim na malo tezi nacin :)

**Nacin 2:**

Ovo je malo tezi (*ili malo laksi???*) nacin od prethodnog i ja sam ga prvobitno upotrebio da pronadjem API koji nam treba. Otvorio sam ovaj crackme pomocu ResHackera da pronadjem ID (*jedinstveni identifikacioni broj*) pomocu

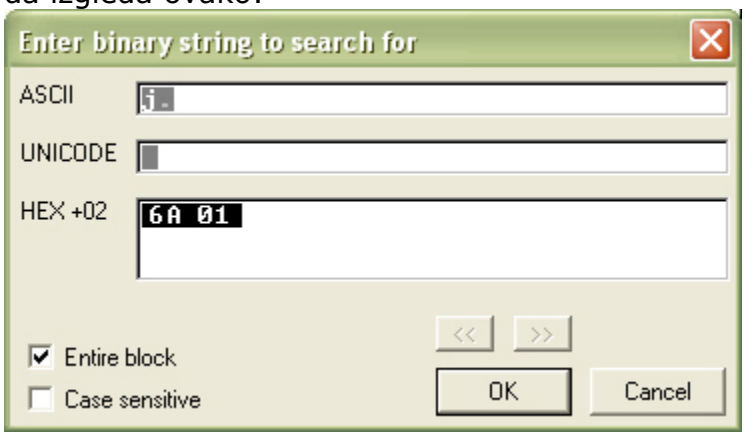
kojeg se program "obraca" ovoj kontroli. Kada otvorimo crackme pomocu ResHackera imacemo ovo:

```
900 DIALOG 0, 0, 240, 191
STYLE DS_MODALFRAME | DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "NAG-SCREEN"
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
FONT 8, "MS Sans Serif"
{
  CONTROL "&Continue", 1, BUTTON, BS_PUSHBUTTON | BS_CENTER | WS_CHILD |
WS_VISIBLE | WS_DISABLED | WS_TABSTOP, 92, 172, 50, 14
  CONTROL "NAG", 101, STATIC, SS_CENTER | WS_CHILD | WS_VISIBLE, 20, 48, 200, 10
  CONTROL "This nag stays here for 10 seconds, because", 102, STATIC, SS_CENTER |
WS_CHILD | WS_VISIBLE, 20, 120, 192, 8
  CONTROL "you did not pay for this program !", 103, STATIC, SS_CENTER | WS_CHILD |
WS_VISIBLE, 20, 132, 200, 9
  CONTROL "NAG, NAG, NAG ;))", 104, STATIC, SS_CENTER | WS_CHILD | WS_VISIBLE, 40,
146, 148, 13
...
}
```

iz cega smo saznali mnogo... Saznali smo da je ID cele forme koja se pojavljuje na ekranu 900 (*decimalno*) i saznali smo da je ID dugmeta Continue jednak 1. Ovo je bitno jer kada program na ASM nivou zeli da uradi nesto sa nekim objektom on prvo mora da zna na koji objekat se data komanda odnosi. Dakle ono sto je sigurno je da ce se kao jedan od parametara API funkciji koja ukljucuje dugme sigurno proslediti i ID dugmeta. To bi moglo da izgleda bas ovako:

```
PUSH ukljuci_dugme
PUSH Id_controle
PUSH neki_drugi_parametri
CALL UkljuciDugmeAPI
```

Kao sto vidimo moraju se koristiti PUSH komande, a PUSH 1 kako bi trebalo da izgleda prosledjivanje IDa dugmeta bi u hex obliku izgledalo ovako 6A01. Sada samo treba da potrazimo ovaj binarni string u fajlu i videcemo odakle se to poziva API za ukljucivanje dugmeta. Pritisnite CTRL + B u CPU prozoru Ollyja da nadjete ovaj string. Kada unesete 6A01 prozor koji se pojavio treba da izgleda ovako:



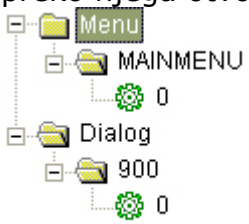

Pritiskom na dugme OK program ce nas odvesti na adresu 0040111B a ocigledno je da je to pogresna adresa jer CreateFileA sluzi za nesto drugo. Pritisnucemo onda CTRL+L (search again) i zavrsicemo na adresi 0040125A gde se nalazi par zanimljivih redova:

PUSH 1 // enable = true,

red PUSH 1 // ControlID i red CALL EnableWindow pa je ocigledno da se API EnableWindow koristi za ukljucivanje dugmeta a da su mu parametri handle kontrole i true ili false switch.

# ReEnable buttons - ResHacker

Ova neverovatno lame vrsta "crackovanja" je veoma zastupljena kod pocetnika i onih koji jednostavno ne zele da se "zamaraju" kopanjem po ASM kodu. Daleko od toga da ReSHacker nije koristan i pravim reverserima i da se ne ustrucavaju da ga koriste. Naravno pravi reverseri upotrebljavaju ovaj izuzetan program za kreiranje RES fajlova za modifikaciju sopstvenih programa a ne za beznacajno ukljucivanje ukljucenih dugmica. Bez obzira na sve ovo naucicu vas kako da koristite ovaj program za resavanje NAG problema. Kada otvorite Datenov program (*pogledaj prosli deo poglavlja*) i preko njega otvorite nasu metu videcete sledece drvo u levoj strani ekrana:




Naravno jasno je sta sve ovo znaci... U .res sekciji .exe fajla postoji jedan Meni i jedan Dialog. Ako selektujemo ./Menu/MAINMENU/0 videcemo glavni meni programa koji se pojavljuje kada se iskljuci NAG screen. Jedini dijalog koji se nalazi u .res sekciji je bas taj NAG screen koji zelimo ili da uklonimo ili da dugme Continue u njemu uvek bude ukljuceno. Da bismo ovo uradili otvoricemo ./Dialog/900/0 i pogledacemo sta se nalazi tamo:

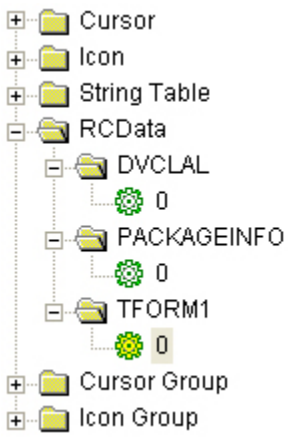
```
900 DIALOG 0, 0, 240, 191
STYLE DS_MODALFRAME | DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "NAG-SCREEN"
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
FONT 8, "MS Sans Serif"
{
  CONTROL "&Continue", 1, BUTTON, BS_PUSHBUTTON | BS_CENTER | WS_CHILD |
WS_VISIBLE | WS_DISABLED | WS_TABSTOP, 92, 172, 50, 14
  CONTROL "NAG", 101, STATIC, SS_CENTER | WS_CHILD | WS_VISIBLE, 20, 48, 200, 10
  CONTROL "This nag stays here for 10 seconds, because", 102, STATIC, SS_CENTER |
WS_CHILD | WS_VISIBLE, 20, 120, 192, 8
  CONTROL "you did not pay for this program !", 103, STATIC, SS_CENTER | WS_CHILD |
WS_VISIBLE, 20, 132, 200, 9
  CONTROL "NAG, NAG, NAG ;))", 104, STATIC, SS_CENTER | WS_CHILD | WS_VISIBLE, 40,
146, 148, 13
  CONTROL "Frame1", 105, STATIC, SS_ETCHEDFRAME | WS_CHILD | WS_VISIBLE, 8, 4, 224,
164
}
```

Ovo sto tu pise nam je skroz nebitno posto sve mozemo da odradimo pomocu samog programa ResHacker, bez imalo kucanja. Ako NAG dijalog nije prikazan pritisnite Show Dialog dugme. Potom selektujete iskljuceno dugme, pritisnite desno dugme misa na njemu i selektujte Edit Control. U novootvorenom prozoru pronadjite WS_DISABLED i iskljucite ga. Pre nego sto snimite promene potrebno je da kliknete na OK a onda na Compile Script da bi program ucitao promene u memoriju. Sada mozete pritisnuti Save... Ovako ce dugme uvek biti ukljuceno jer program ne proverava stanje dugmeta pre nego sto mu posalje komandu da se ukljuci.

Sam NAG Dialog se moze ukloniti na par nacina: Mozete selektovati ./Dialog/900/0 i klikom na desno dugme ili reimenovati Dialog 900 u nesto drugo ili ga jednostavno obrisati. Tako se NAG dijalog nece nikada vise pojavljivati. **NAPOMENA:** Brisanje ili reimenovanje dijaloga nije preporucljivo a u vecini slucajeva nece ni raditi kako treba pa ce se program srusiti.

# ReEnable buttons -ResHacker & Delphi

Prica vezana za ResHacker se malo razlikuje kada je Delphi u pitanju. Delphi je specifican jer u novijim Delphi verzijama, Delphi ne pravi klasicne .res sekcije unutar exe fajla nego je ova sekcija specificna samo za njega. Kada otvorimo program pomocu …\Casovi\Cas8\EnableMe.exe ResHackera videcemo u drvetu sa strane ovo:



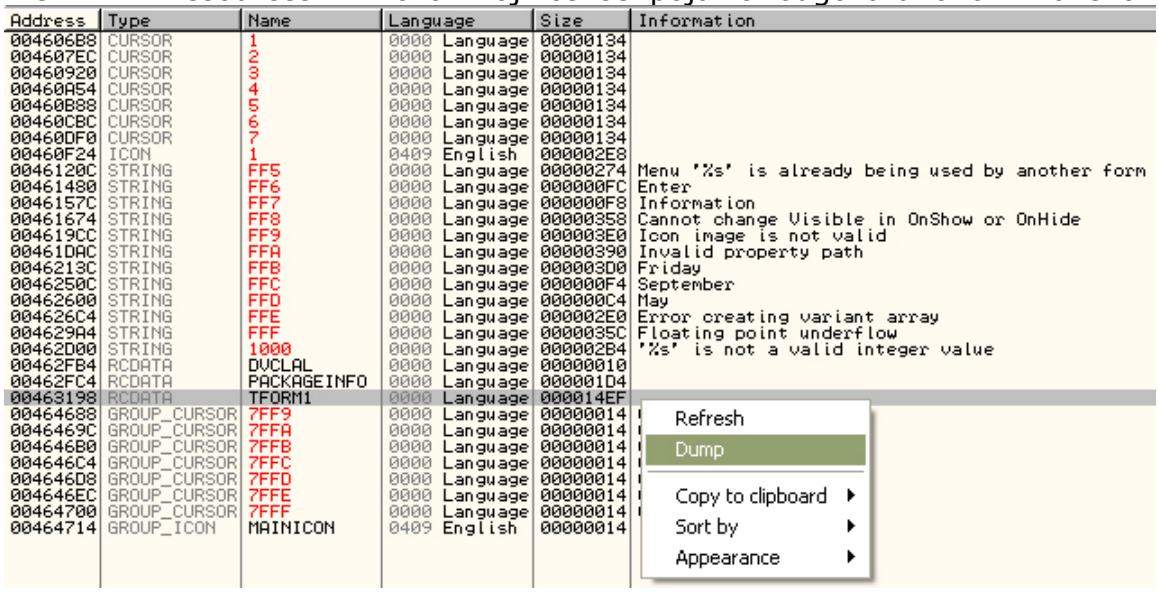

Za Delphi je karakteristicno da svoje resurse smesta u posebnom obliku. Ono sto vidite na slici je putanja do glavne forme u kojoj se nalaze ukljuceni objekti. Kada selektujemo ovu putanju ./RCData/TFORM1/0 imacemo pregled svih objekata koji se nalaze u selektovanoj formi i odakle je moguce modifikovati sve objekte koji se nalaze u formi. Primera radi odskrolovacemo na kraj ovog prozora da bismo na kraju pronasli tri iskljucena objekta. Prvi iskljuceni objekat je dugme, drugi iskljuceni objekat je EditBox a treci je glavni meni. Svaki objekat je definisan pocetkom i krajem koji se oznacava kao END te kontrole. Izmedju pocetka i kraja se nalaze sve vrednosti koje taj objekat ima i koje se mogu odavde modifikovati.

```
object Button1: TButton
  Left = 8
  Top = 56
  Width = 177
  Height = 33
  Caption = 'The Art Of Cracking - Enable ME'
  Enabled = False
  TabOrder = 0
  OnClick = Button1Click
end
object Edit1: TEdit
  Left = 48
  Top = 24
  Width = 129
  Height = 21
  Enabled = False
  TabOrder = 1
  Text = 'Enable this'
end
object MainMenu1: TMainMenu
  Left = 88
  Top = 48
  object File1: TMenuItem
  Caption = 'File'
  Enabled = False
  object Exit1: TMenuItem
  Caption = 'Exit'
  OnClick = Exit1Click
end
```

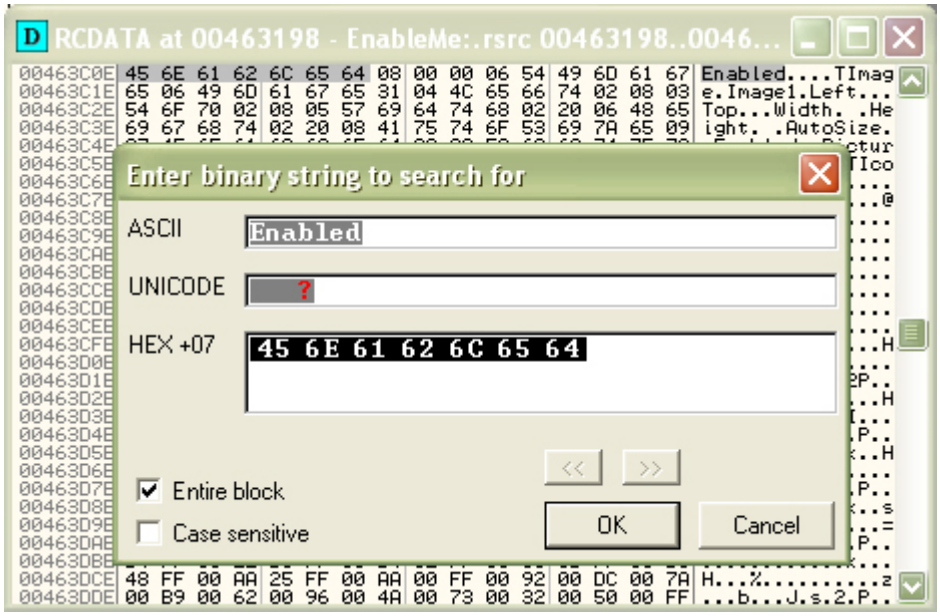
Ono sto je nama bitno za svaki objekat je vrednost svake kontrole koja se nalazi uz Enabled. Ako je Enabled = False onda je taj objekat iskljucen a ako se uz Enabled nalazi True umesto False onda ce objekat biti ukljucen. Kada izmenimo False u True za sve objekte koji nam trebaju onda cemo pritisnuti Compile Script dugme, a sa Save snimamo promene.

# ReEnable buttons – Olly & Delphi

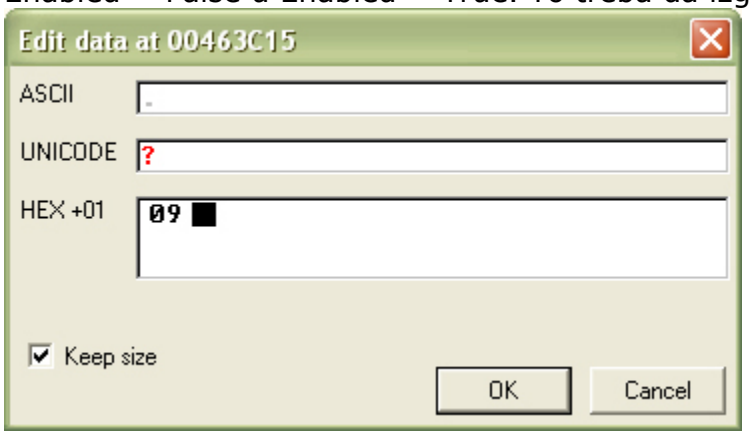
Videli smo da se format Delphi resursa dosta razlikuje od standardnog nacina zapisa resursa. U ovom delu poglavlja cu vas nauciti kako da iskoristite ovo da bez upotrebe ResHackera ukljucite iskljucene objekte. Naravno da se pitate zasto biste ovo uopste ucili ako su vam pri ruci alati kao sto su ResHacker, VBReformer i slicni. Naravno da ne morate da ucite ovu tehniku ali postoji veci broj prednosti ako patchujete programe ovako. Naime kada ukljucite neko dugme recimo pomocu ReSHackera on umesto da jednostavno promeni samo jedan bajt on promeni veci deo koda i zbog toga se promeni velicina fajla koji ste patchovali. Sada shvatate zasto je ovo lose. Ako koristite ResHacker necete moci da uporedite dva fajla jer su im velicine razlicite, a samim tim cete biti prinudjeni da ako distribuirate vase patcheve (*ne radite ovo sa komercijalnim shareware programima jer je nezakonito*) bicete prinudjeni da saljete cele .exe fajlove a ne male patchere na internet. Zato je najbolje koristiti ovaj prilicno 1337 (*elite*) nacin :) Otvorite program …\Casovi\Cas8\EnableMe.exe pomocu Ollyja. Predjite u Executable Modules Window (ALT + E) i tu pritisnite desno dugme na glavni exe fajl i izaberite View ALL Resources... Prozor koji ce se pojaviti odgovara onom na slici:




Od svih ovih resursa interesuje nas samo sadrzaj glavnog Forma, forma 1 i zato cemo kliknuti desnim dugmetom na nju i izvrsicemo Dump kao na slici iznad. U novootvorenom prozoru moramo da potrazimo objekte koje treba da ukljucimo. Imajte na umu da se ovo moze raditi i preko obicnog Hex editora samo ja preferiram da sve zavrsavam preko Ollyja koji se u ovom slucaju ponasa bas kao obican Hex Editor. Ono sto cemo traziti u ovom delu koda kao obican binary string je rec Enabled jer je ona specificna za Delphi objekte i oznacava da li je neki objekat ukljucen ili ne. Pritisnite desno dugme -> Search for -> Binary string i u polje ASCII unesite string Enabled koji cemo traziti. Klikom na OK Olly ce nas odvesti do prvog pojavljivanja stringa Enabled to jest do adrese 00463C0E a to vidimo i na slici koja se nalazi ispod.




Ako pogledate malo iznad selektovanog teksta videcete na koji objekat se odnosi nadjeni string Enabled. Taj objekat je **TLabel.Label1** na kojem pise sledeci tekst ***Enable this label two***, a kao sto smo videli pri startovanju mete ovaj tekst je iskljucen i treba ga ukljuciti. Kako ovo radimo ??? Selektovacemo prvi broj koji se nalazi iza nadjenog teksta Enabled (*prvu tacku ako posmatrate tekst Enabled a ne hex vrednosti*), dakle selektovacemo 08 i pritisnucemo desno dugme -> binary -> edit ili samo CTRL + E. Ovaj hex broj 08 treba da promenimo u 09 da bismo promenili stanje ovog objekta iz Enabled = False u Enabled = True. To treba da izgleda bas ovako:




Posle pritiska na OK dugme program ce patchovati selektovani bajt u 09 a mi cemo pritiskom na CTRL+L nastaviti potragu za iskljucenim objektima. Da li je objekat iskljucen ili ne mozete prepoznati jednostavnim pogledom na HEX broj iza stringa Enabled. Ako je taj broj 08 onda je objekat iskljucen a ako je broj 09 onda je objekat ukljucen. Izmenite sve iskljucene objekte u ukljucene u ovom primeru. Ako startujete program sa F9 videcete da su svi dugmici ukljuceni i da program pritiskom na dugme The Art Of Cracking – Enable Me govori da je uspesno crackovan. Jedini problem nastaje kada zelite da snimite ove promene. Iz nekog razloga ovako izmenjeni podaci se ne mogu snimiti. Ovo i nije neki problem, jednostavno zapisite adrese na kojima se nalaze 08 bajtovi i izmenite ih u CPU prozoru na isti nacin kao i ovde. Iz CPU prozora je dozvoljeno snimanje klikom desno dugme -> Copy to executable -> All modifications -> Copy All -> Save

# ReEnable buttons – Olly & VB

Videli smo da Delphi ima svoj nacin zapisivanja resursa, a ono sto sledi iz toga je da i drugi programi mogu imati nestandardne nacine zapisivanja. Visual Basic je takodje jedan od programa sa nestandardim nacinom zapisivanja resursa. Postoje programi koji mogu da urade slicne stvari, npr. VBReformer, ali nacin koji cu vam ja ovde predstaviti je 1337 :)

Posto postupak pronalaska tacnog mesta u fajlu gde se to definise da li je dugme ili neki drugi objekat ukljucen ili iskljucen zahteva da imate instaliran Visual Basic alat, ja sam vec uradio uporedjivanje dva fajla sa iskljucenim i ukljucenim dugmicima i menijima. Prilikom toga dosao sam do zakljucka da ako imamo recimo meni koji treba da ukljucimo treba da pronadjemo naziv menija u fajlu pri vrhu samog fajla, to jest prvi od vrha i da izmenimo treci bajt iza ovog stringa. Ako je meni ukljucen onda je taj bajt FF a ako je iskljucen onda je taj bajt 00. Evo kako to izgleda:

```
00401386  . 4C 65 76 65 6>ASCII "Level02",0
0040138E    05          DB 05
0040138F    00          DB 00 <- Ovo je treci bajt jer se racuna i nula iza Level02
00401390    FF          DB FF <- Taj bajt je 00 sto znaci da je meni iskljucen
```

Ista prica se ponavlja i sa dugmicima samo sto se pozicija menja. To jest kod dugmica nije u pitanju treci nego dvanaesti bajt. Evo kako to izgleda:

```
00401303  . 45 6E 61 62 6>ASCII "Enable Me  - Lev"
00401313  . 65 6C 30 31 0>ASCII "el01",0
00401318    04          DB 04
00401319    78          DB 78                       ;  CHAR 'x'
0040131A    00          DB 00
0040131B    78          DB 78                       ;  CHAR 'x'
0040131C    00          DB 00
0040131D    4F          DB 4F                       ;  CHAR 'O'
0040131E    0B          DB 0B
0040131F    67          DB 67                       ;  CHAR 'g'
00401320    02          DB 02
00401321    08          DB 08
00401322    00          DB 00 <- Ovo je dvanaesti bajt jer se racuna i nula iza Level012
00401323    11          DB 11 <- Taj bajt je 00 sto znaci da je dugme iskljuceno
00401324    00          DB 00
00401325    00          DB 00
00401326    FF          DB FF
00401327    03          DB 03
```

Naravno u pitanju je obicno kompajlovanje VB programa a ne PCODE. Ovo je zapazanje vezano samo za obicne VB programe, i vazi samo za dugmice i menije. Sto se tice ostalih elemenata koji se mogu naci u VB programima, to cete morati sami da ispitate ali ce logika (*FF je jednako true, a 00 false*) sigurno biti ispunjena. Objasnio sam vam kako treba da pronadjete bajtove koje treba da izmenite kako bi ukljucili menije i dugmice. Za vezbu uradite program …\Casovi\Cas8\Crackme08.exe

**Resenje:**
0040138F promeniti 00 u FF
00401322 promeniti 00 u FF
004013C6 promeniti 00 u FF

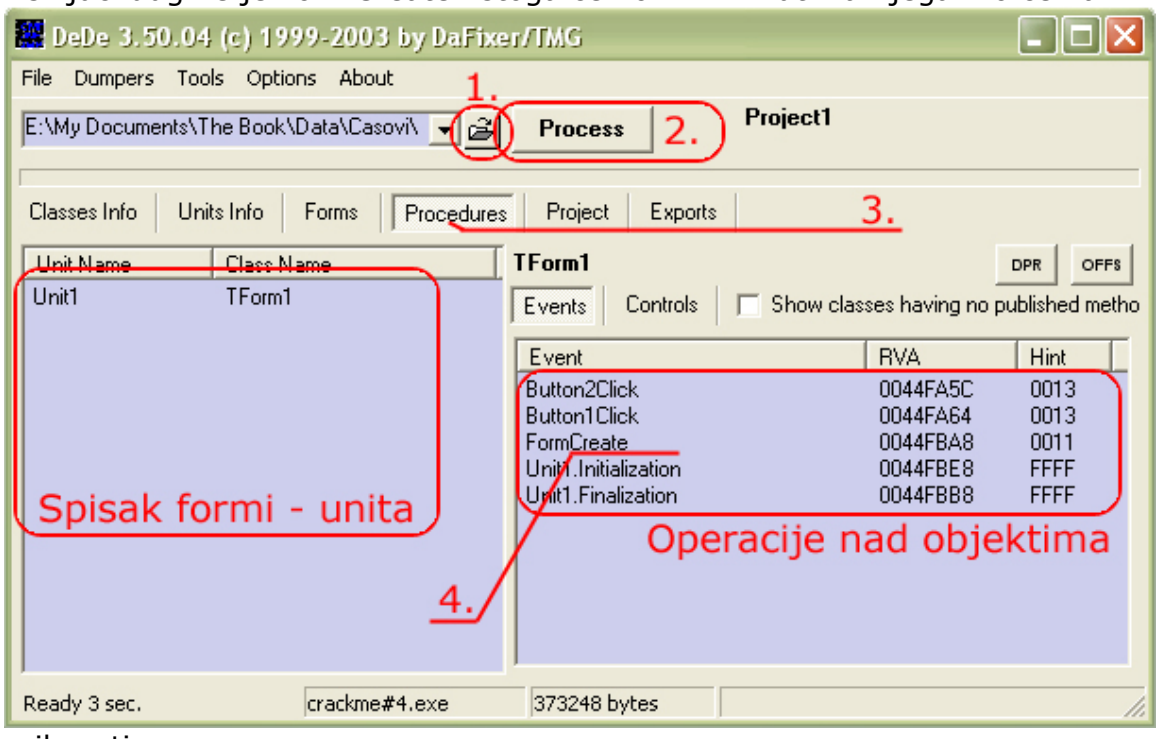
# ReEnable buttons – DeDe & Delphi

Vec sam pomenuo da postoje specificni programi koji se bave analizom posebnih kompajlera. Jedan od takvih kompajlera je i Delphi a specijalizovani program koji se bavi samo Delphijem i njegovim podverzijama je DeDe, program koji je napisao DaFixer. Meta u kojoj je iskljuceno dugme preko neke procedure (*tipa Object.Enabled := False;*) se nalazi u folderu Cas12 a zove se crackme#4.exe i nju cemo izmeniti tako da dugme check bude uvek ukljuceno. Mozemo prvo da pogledamo resurse u programu da bismo zakljucili kako je ovo dugme iskljuceno:

```
object Button1: TButton
   Left = 176
   Top = 9
   Width = 81
   Height = 24
   Caption = 'Check'
   TabOrder = 0
   OnClick = Button1Click
end
```

kao sto se vidi iz ResHackera dugme je stalno ukljuceno tako da se njegovo iskljucivanje sigurno radi preko procedure u obliku Button1.Enabled := False; Preko DeDe-a cemo pronaci gde se ovakva procedura izvrsava. Otvoricemo DeDe i ucitacemo metu u njega pomocu komande Open a njenu analizu cemo izvrsiti pomocu komande process.

Posle ovoga DeDe ce vas pitati da li da izvrsi standardno VCL prepoznavanje, na sta cete odgovoriti potvrdno. Kada se zavrsi ovo ispitivanje izaberite opciju procedures i u njoj cete videti spisak svih operacija nad objektima. Posto ih ima malo vidi se da jedini koji moze da iskljuci dugme je FormCreate i stoga cemo 2x kliknuti na njega. To ce nam



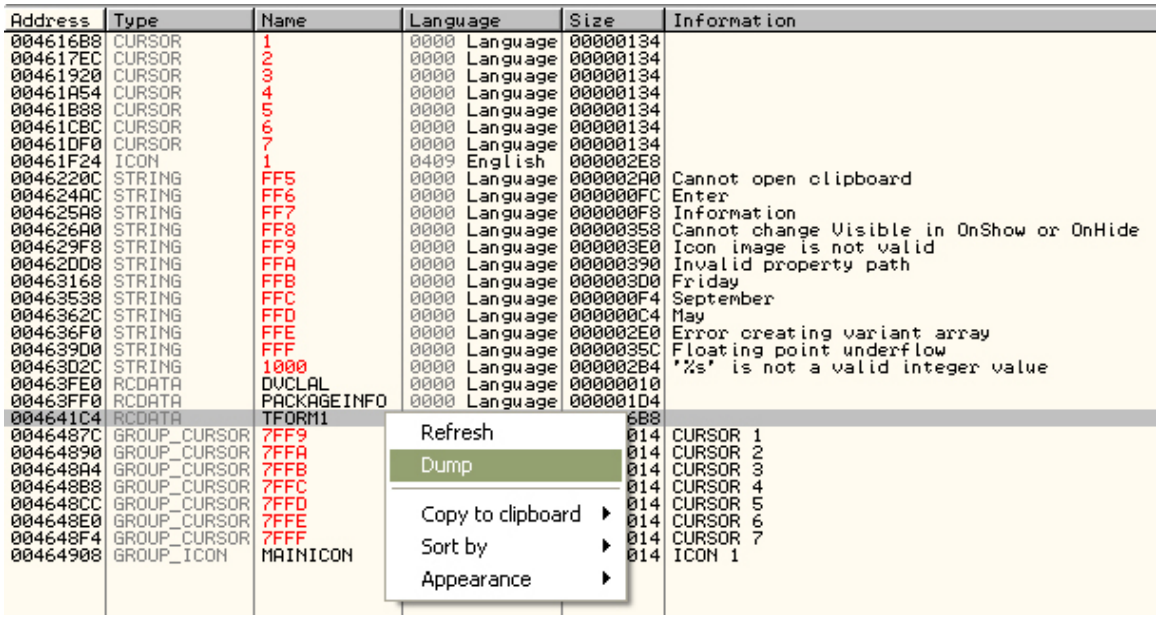

prikazati ovo

```
* Reference to method TButton.SetEnabled(Boolean)
0044FBB2  FF5164              call   dword ptr [ecx+$64]
0044FBB5  C3                  ret
```

a ovo znaci da se CALL na adresi 0044FBB2 koristi za iskljucivanje dugmeta. Da bi dugme stalno bilo ukljuceno potrebno je samo NOPovati ovaj CALL.

# Passwords – Olly & Delphi

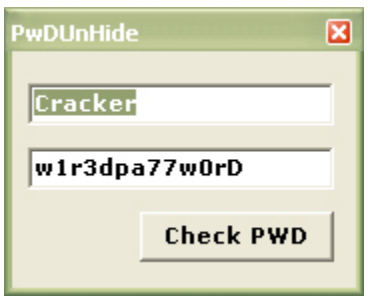
Videli smo da Delphi ima svoj nacin zapisivanja resursa, a ono sto nas sada interesuje je kako da provalimo u password polja i vidimo sta se nalazi zapisano unutra. Primer za ovo se nalazi u folderu Cas8 a zove se pwdMe.exe. Ovaj fajl cemo otvoriti pomocu Ollyja. Vec smo naucili kako da pronadjemo resurse u Delphi exe fajlu pa to necu ponavljati. Dumpovacemo jedinu formu, TFORM1.

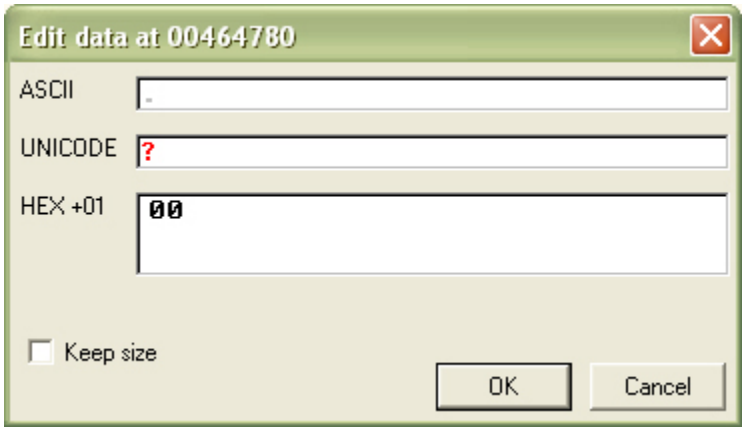



Ono sto je specificno za Delphi je da unutar resursa svako Delphi polje, odnosno Edit ima parametar PasswordChar koji oznacava koji je to karakter kojim ce biti zamenjeno svako uneto slovo u taj Edit. Ovaj karakter je najcesce zvezdica *. Stoga cemo pronaci PasswordChar u dumpu koji ce se pojaviti. Taj string cemo naci ovde:

**00464772 50 61 73 73 77 6F 72 64 43 68 61 72 06 01 2A 08 PasswordChar..*.**

Ako ste nekada koristili Delphi videli ste da je Default karakter za polje Edit ako ono nije Password tipa #0 ili 00 hex. Stoga cemo samo zameniti nasu zvezdicu 2A hex sa 00 i pretvoricemo polje za unos iz Password polja u obicno polje. Evo kako to izgleda posle patchovanja tog bajta:



# Passwords – Olly & VB

Visual Basic kao uvek je prica za sebe pa ce i uklanjanje zvezdica sa password polja u VB programima biti prica za sebe. Meta za ovaj deo knjige se nalazi u folderu Cas8 a zove se pwdMe2.exe. Otvorite ovu metu pomocu Ollyja. Sada pocinju muke, treba pronaci kako se zove forma u kojoj se nalazi, nepoznati textbox sa passwordom i kako se konacno sklanjaju zvezdice. Otici cemo u string reference i tamo cemo pronaci sto vise informacija o samoj meti. Videcemo koji se to objekti pojavljuju u exe fajlu. To su:

```
004013F8  DD pwdMe2.00401410          ASCII "Form1"
00401814  DD pwdMe2.00401474          ASCII "Form"
0040183C  DD pwdMe2.00401500          ASCII "Command1"
00401864  DD pwdMe2.0040151C          ASCII "Text2"
0040188C  DD pwdMe2.00401524          ASCII "Text1"
```

Znaci imamo samo jednu formu sa dva Textboxa i jednim dugmetom. Posto se objektima dodeljuju vrednosti na samom pocetku fajla kliknucemo na sadrzaj jednog od textboxova iz istog forma. Kliknucemo 2x na ovo:

```
Text strings referenced in pwdMe2:.text, item 7
 Address=00401297
 Disassembly=ASCII "CrackeR",0
```

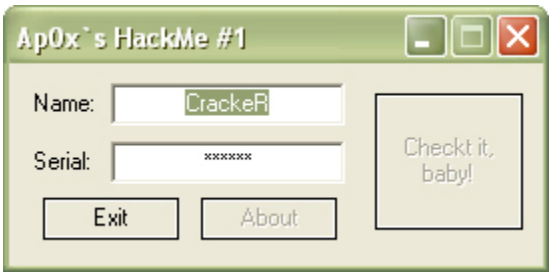
jer ne znamo sta je sadrzaj password polja a ono se nalazi dovoljno blizu. Ovo iznad nam izgleda mnogo sumljivo:

```
00401260  > \1D 010B0B00          SBB EAX,0B0B01
00401265  . 70 77                 JO SHORT pwdMe2.004012DE
00401267  . 64:55                 PUSH EBP                    ; Superfluous prefix
00401269  . 6E                    OUTS DX,BYTE PTR ES:[EDI]   ; I/O command
0040126A  . 48                    DEC EAX
0040126B  . 696465 4D 650>        IMUL ESP,DWORD PTR SS:[EBP+4D],1120065
00401273  . 0022                  ADD BYTE PTR DS:[EDX],AH
00401275  . 0100                  ADD DWORD PTR DS:[EAX],EAX
00401277  . 2A00                  SUB AL,BYTE PTR DS:[EAX]
00401279  . 2803                  SUB BYTE PTR DS:[EBX],AL
0040127B  . FF03                  INC DWORD PTR DS:[EBX]
0040127D  . 26:0000               ADD BYTE PTR ES:[EAX],AL
00401280  . 0001                  ADD BYTE PTR DS:[ECX],AL
00401282  . 05 00546578           ADD EAX,78655400
00401287  . 74 31                 JE SHORT pwdMe2.004012BA
00401289  . 0002                  ADD BYTE PTR DS:[EDX],AL
0040128B  . 04 78                 ADD AL,78
0040128D  . 00F0                  ADD AL,DH
0040128F  . 0007                  ADD BYTE PTR DS:[EDI],AL
```

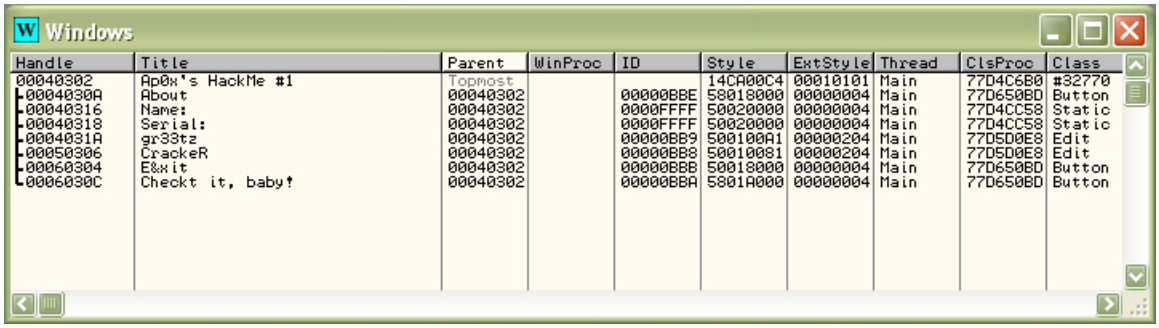
Da li je ovo podatak koji nam treba ??? Ako pretrazimo ovaj kod naci cemo tacno jedan 2A bajt. Ovo bi u ASCIIju znacilo *, tako da to moze biti bas ono sto trazimo. Selektovacemo adresu 401277 i kliknucemo na desno dugme -> Binary -> Edit. I izmenicemo ovaj bajt 2A u 00 vodeci se onim sto smo naucili od Delphija. Startovacemo program sa F9 i videcemo da u password polju sada pise: pwdUnHideMe. Dakle uspeli smo provaliti smo u VB passworde. Naravno postoji laksi nacin da nadjemo ovaj bajt pomocu nekog Hex editora cemo samo potraziti 2A bajtove i pogledacemo gde se oni nalaze. Ako se nalazi blizu nekog textboxa moguce je da je on bas taj bajt koji trazimo. Ako postoji vise ovakvih polja moracete malo da pogadjate na koji se textbox odnosi 2A bajt i da li on uopste pripada nekom textboxu.

# Passwords – Olly & ASM

Za razliku od Delphija, ASM, C++ programi, imaju specijalan nacin zapisivanja to jest koriscenja ovog password polja. Ako otvorite neku metu sa ResHackerom videcete da polje ima atribut ES_PASSWORD, koji mozemo da uklonimo i polje ce postati vidljivo, to jest nece biti vise password tipa nego ce biti obicno. Ovo moze u nekim slucajevima da pomogne (*ako program koristi resurse i nema CRC32 proveru*) ali nacin koji cu vam ja sada pokazati vazi za svaki programski jezik i omogucava vam da bez ikakvih password unhider programa procitate sta se to nalazi unutar nekog password polja. Meta za ovaj deo knjige se nalazi u folderu Cas8 a zove se pwdMe3.exe, i nju cemo otvoriti pomocu Ollyja.
Pokrenite nasu metu pomocu Ollyja, to jest pritisnite F9. Pojavice se nasa meta u svom punom sjaju.




Treba da shvatimo da iako se preko nepoznatog teksta nalaze zvezdice on je jos tu i stoga mozemo saznati sta pise u tom polju. Imamo srece sto u samom Ollyju postoji opcija koja nam moze prikazati sve trenutno aktivne prozore debugovane aplikacije, kao i sadrzaj svakog polja za unos. Ova opcija se nalazi u toolbaru i obelezena je sa W. Kada pritisnemo to dugme videcemo sledece:




Vidimo da se u aktivnom prozoru nalaze samo dva Edit dugmeta i da su njihovi sadrzaji: CrackeR i gr33tz. Posto vidimo samo CrackeR znaci da je sadrzaj password polja gr33tz, i ovo je nas trazeni password. Ovo je zgodan i najbrzi nacin da pronadjete skriveni password u bilo kom programu. Izgleda da password polja i nisu bas toliko sigurna.

# Time-Trial

Postoji veliki broj programa pisan u svim programskim jezicima cije je koriscenje ograniceno na jedan, a najcesce veoma kratak, vremenski period. Jedna takva aplikacija se nalazi u folderu Cas08 a zove se timetrial.exe.


Primeticete da se aplikacija moze koristiti samo tri dana ili se moze startovati samo nekoliko puta. Ovo cemo resiti tako da aplikacija nikada ne istekne. Upalite Olly i ucitajte ovu aplikaciju. Videcete niz API poziva: CreateFileA, MessageBoxA, ReadFileA, GetSystemTime.... Bez nekakve velike mudrosti vidi se da program otvara fajl DATA.DET, a ako on ne postoji prikazuje poruku o tome na ekran. Ako fajl postoji iz njega se cita 50 bajtova, posle cega se poziva API GetSystemTime koji vraca vreme na vasem kompjuteru. Proci cemo kroz sve ove API pozive sa F8 sve dok se ne nadjemo ovde:

```
00401081  |.  803B 00                CMP BYTE PTR DS:[EBX],0
```

Primeticemo da se JNZ skok nece izvrsiti pa cemo sigurno doci u sledeci loop:

```
0040108B  |> /66:8B81 E4304>         /MOV AX,WORD PTR DS:[ECX+4030E4]
00401092  |. |66:35 6969              |XOR AX,6969
00401096  |. |66:8981 AB304>         |MOV WORD PTR DS:[ECX+4030AB],AX
0040109D  |. |83C1 02                |ADD ECX,2
004010A0  |> |83F9 08                |CMP ECX,8
004010A3  |.^\76 E6                  \JBE SHORT timetria.0040108B
```

Posto ja nemam zelju da prolazim kroz njega postavicu break-point na prvu adresu ispod loopa (004010A5) i pritisnucu F9. Ali objasnicu vam sta se ovde desava: Program jednostavno proverava kada je fajl snimljen, ako je to 2001. prva godina onda ce samo skinuti jedno otvaranje programa i updateovace datum koji se nalazi na .DET fajlu. Naravno sve ovo ce biti snimljeno negde dole, ovde gore u loopu se samo proverava datum. Zbog ovoga ce se samo prvi put ici u ovaj loop, a svaki sledeci put ne. Posto nas ovaj prvi put i ne zanima onda cemo pritisnuti F9 da bismo startovali program i updateovali datum .DET fajla. Primeticemo da nam je program smanjio broj startovanja programa za jedan. Restartovacemo Olly i ici cemo sa F8 kroz kod sve dok ne dodjemo do:

```
00401084  |. /75 22                  JNZ SHORT timetria.004010A8
```

primetite da se sada ovaj skok izvrsava i da cemo se naci ovde:

```
004010A8  |> \8B0D AB304000          MOV ECX,DWORD PTR DS:[4030AB]
004010AE  |.  81F1 69696969          XOR ECX,69696969
004010B4  |.  A1 E4304000            MOV EAX,DWORD PTR DS:[4030E4]
004010B9  |.  3BC1                   CMP EAX,ECX
004010BB  |.  0F85 85000000          JNZ timetria.00401146
004010C1  |.  66:8B0D B1304>         MOV CX,WORD PTR DS:[4030B1]
004010C8  |.  66:81F1 6969           XOR CX,6969
004010CD  |.  66:A1 EA30400>         MOV AX,WORD PTR DS:[4030EA]
004010D3  |.  66:2BC1                SUB AX,CX
004010D6  |.  66:83F8 03             CMP AX,3
004010DA  |.  77 6A                  JA SHORT timetria.00401146
004010DC  |.  2805 00304000          SUB BYTE PTR DS:[403000],AL
004010E2  |>  A0 B5304000            MOV AL,BYTE PTR DS:[4030B5]
004010E7  |.  34 69                  XOR AL,69
004010E9  |.  3C 00                  CMP AL,0
```

Kao sto vidimo neke vrednosti se racunaju i porede. Ako bilo koji od ovih uslova nije ispunjen skocice se ovde:

```
00401146  |> \6A 30                PUSH 30
00401148  |.  68 97304000          PUSH timetria.00403097; |Title = "Too Bad ;)"
0040114D  |.  68 76304000          PUSH timetria.00403076 ;|Text = "Sorry, this crackme”
00401152  |.  6A 00                PUSH 0                 ; |hOwner = NULL
00401154  |.  E8 A1000000          CALL <MessageBoxA>     ; \MessageBoxA
```

Naravno moguce je promeniti skokove tako da se ovo nikada ne izvrsi ali posto postoje programi koji proveravaju da li su im modifikovani odredjeni skokovi mi cemo naterati ovaj program da snimi .DET fajl tako da program nikada ne istekne. Prvo imamo ovu proveru:

```
004010A8  |> \8B0D AB304000        MOV ECX,DWORD PTR DS:[4030AB]
004010AE  |.  81F1 69696969        XOR ECX,69696969
004010B4  |.  A1 E4304000          MOV EAX,DWORD PTR DS:[4030E4]
004010B9  |.  3BC1                 CMP EAX,ECX
004010BB  |.  0F85 85000000        JNZ timetria.00401146
```

Posto se porede EAX i ECX u ECX cemo staviti vrednost iz EAXa jer ce tako ova dva broja uvek biti jednaka. Promenicemo gornji kod u ovo:

```
004010A8  |> \8B0D E4304000        MOV ECX,DWORD PTR DS:[4030E4]
004010AE  |.  90                   NOP
004010AF  |.  90                   NOP
004010B0  |.  90                   NOP
004010B1  |.  90                   NOP
004010B2  |.  90                   NOP
004010B3  |.  90                   NOP
004010B4  |.  A1 E4304000          MOV EAX,DWORD PTR DS:[4030E4]
004010B9  |.  3BC1                 CMP EAX,ECX
004010BB  |.  0F85 85000000        JNZ timetria.00401146
```

Primetite da sam program modifikovao tako da ce se u EAX i ECX smestati vrednosti sa iste adrese 004030E4. Primeticete i da sam obrisao (NOPovao) onu XOR ECX komandu, jer bi ona poremetila vrednost ECXa. Idemo na sledeci niz provera:

```
004010C1  |.  66:8B0D B1304>       MOV CX,WORD PTR DS:[4030B1]
004010C8  |.  66:81F1 6969         XOR CX,6969
004010CD  |.  66:A1 EA30400>       MOV AX,WORD PTR DS:[4030EA]
004010D3  |.  66:2BC1              SUB AX,CX
004010D6  |.  66:83F8 03           CMP AX,3
004010DA  |.  77 6A                JA SHORT timetria.00401146
```

Ovde je ocigledno da se nesto racuna pa se AX poredi sa 3... Hmmm sa brojem dana koji nam je dozvoljen da koristimo program. Ovo cemo izmeniti u ovo:

```
004010C1  |.  66:8B0D EA304>       MOV CX,WORD PTR DS:[4030EA]
004010C8  |.  90                   NOP
004010C9  |.  90                   NOP
004010CA  |.  90                   NOP
004010CB  |.  90                   NOP
004010CC  |.  90                   NOP
004010CD  |.  66:A1 EA30400>       MOV AX,WORD PTR DS:[4030EA]
004010D3  |.  66:2BC1              SUB AX,CX
004010D6  |.  66:83F8 03           CMP AX,3
004010DA  |.  77 6A                JA SHORT timetria.00401146
```

Primeticete da smo kao i gore CXu i AXu dodelili iste vrednosti, da smo izbrisali XOR. Ovo je zgodno jer ako pogledate skok ispod videcete da ce se on izvrsiti samo ako je AX vece od 3. Posto AX i CX imaju iste vrednosti posle oduzimanja SUB AX,CX vrednost koja ce se nalaziti u AX je 0, pa stoga nikada nece biti veca od 3. Idemo dalje i stizemo do poslednje provere:

```
004010DC  |.  2805 00304000        SUB BYTE PTR DS:[403000],AL
004010E2  |>  A0 B5304000          MOV AL,BYTE PTR DS:[4030B5]
004010E7  |.  34 69                XOR AL,69
```

```
004010E9  |.  3C 00                 CMP AL,0
004010EB  |. /74 59                 JE SHORT timetria.00401146
```
A ovde se sigurno proverava da li je broj preostalih startovanja programa nula ili ne. Stoga cemo ovu proveru promeniti u ovo:
```
004010DC  |.  2805 00304000         SUB BYTE PTR DS:[403000],AL
004010E2  |>  B0 90                 MOV AL,90
004010E4  |.  90                    NOP
004010E5  |.  90                    NOP
004010E6  |.  90                    NOP
004010E7  |.  34 69                 XOR AL,69
004010E9  |.  3C 00                 CMP AL,0
004010EB  |.  74 59                 JE SHORT timetria.00401146
```
Posto se AL xoruje sa 69, jedina vrednost koja ce posle xorovanja dati nulu je ako se u AL nalazi bas 69. Dakle 69 xor 69 jednako 0. Ovo cemo resiti jednostavno tako sto cemo umesto MOV AL, BYTE PTR DS:[4030B5] ubaciti MOV AL,90. Broj 90 je proizvoljan i mozete uzeti bilo koji broj koji je veci od 69. Naravno da smo mogli i ovde da obrisemo XOR ali nema potrebe, ovako je lakse. Ako pogledamo dole sta ce se izvrsiti ako se svi ovi skokovi ne izvrse, a nece posto smo ih modifikovali tako da nikada nece:
```
004010ED  |.  FEC8                  DEC AL
004010EF  |.  A2 01304000           MOV BYTE PTR DS:[403001],AL
004010F4  |.  34 69                 XOR AL,69
004010F6  |.  A2 B5304000           MOV BYTE PTR DS:[4030B5],AL
004010FB  |.  6A 00                 PUSH 0                          ; /Origin = FILE_BEGIN
004010FD  |.  6A 00                 PUSH 0                          ; |pOffsetHi = NULL
004010FF  |.  6A 00                 PUSH 0                          ; |OffsetLo = 0
00401101  |.  FF35 14314000         PUSH DWORD PTR DS:[403114]      ; |hFile = NULL
00401107  |.  E8 18010000   CALL <JMP.&KERNEL32.SetFilePointer>     ; \SetFilePointer
0040110C  |.  6A 00                 PUSH 0                          ; /pOverlapped = NULL
0040110E  |.  68 E0304000           PUSH timetria.004030E0
00401113  |.  6A 0B                 PUSH 0B                         ; |nBytesToWrite = B
00401115  |.  68 AB304000           PUSH timetria.004030AB          ; |Buffer = 004030AB
0040111A  |.  FF35 14314000         PUSH DWORD PTR DS:[403114]      ; |hFile = NULL
00401120  |.  E8 05010000           CALL <WriteFile>                ; \WriteFile
```
Videcemo da se u fajl DATA.DET snimaju novi podaci o vremenu i broju pokusaja, odnosno samo o broju startovanja programa jer se poslednje vreme startovanja programa cita iz DATA.DET file-timea. Kada snimimo sve ove promene mozemo da startujemo crackme i videcemo sledece:

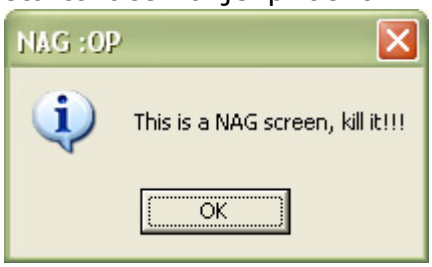

Mozemo restartovati ovaj crackme koliko puta zelimo, mozemo menjati datum a rezultat ce uvek biti isti: “Crackme zarobljen u vremenu :)”. Uspesno smo modifikovali program, a ako ste negde pogresili mozete iskoristiti backup fajl DATA.DET.bak da biste vezbali na ovom primeru.

# Patching a .dll

Cesto sretani problem prilikom reversinga je problem ubijanja NAGova ili pecanja serijskih brojeva na osnovu .dll fajla. Naime cesto se desava da se razne procedure za proveru ili obavestenje smestaju u .dll fajlove iz razloga sto ovi podaci mogu pozivati iz veceg broja .exe fajlova pa je ekonomicnije ovaj deo programa pisati samo jednom u nekom .dll fajlu. Ovaj deo poglavlja cemo podeliti na dva dela.

**Deo I - Patching a NAG in .exe file**

Prvi deo ovog dela poglavlja ce vas nauciti kako da pronadjete NAG. Kazem pronadjete NAG jer ce ovo trazenje biti malo drugacije od onoga na sta ste do sada navikli. Otvoricemo metu ...\Casovi\Cas08\Keygen.exe pomocu Ollya i startovacemo je pritiskom na F9. Ono sto cemo videti je sledeci NAG na




ekranu. Nas sledeci korak je logican, potrazicemo sve string reference u fajlu klikom na *desno dugme -> Search for -> All referenced text strings...* ali kao sto vidimo sam NAG tekst se ne nalazi u ovom fajlu. To nas dovodi do zakljucka da ili je sam tekst NAGa enkriptovan u samom fajlu, ili se cita iz nekog drugog fajla ili se u fajlu nalazi kao neka enkriptovana hex tabela. Bilo kako bilo postavicemo jedan break-point na MessageBoxA API jer se on sigurno koristi za prikaz samog NAGa. Dakle, otici cemo u Executable modules prozor (*ALT+E*), onda cemo izabrati Keygen.exe i pritisnucemo *CTRL + N* a onda cemo u sledecem prozoru pronaci MessageBoxA API i postavicemo break-point on every reference na taj API. Posto smo ovo uradili ostaje nam da restartujemo program sa *CTRL+F2* i da ga ponovo pokrenemo klikom na *F9*. Kao sto vidimo NAG se i dalje pojavljuje. Posto program nije zastao ni na jednom nasem break-pointu, uklonicemo ih sve klikom na *ALT+B* i njihovom pojedinacnom selekcijom i klikom na *DELETE*. Sada nam preostaje ili da pretrazimo sve .dll fajlove koje program poziva (*u nekim slucajevima ovo je komplikovanije zbog velikog broja samih .dll fajlova*) ili da traceujemo kroz .exe fajl sve dok sam program ne pozove NAG. Mi cemo iskoristiti ovu drugu opciju.

Posto se sam NAG pojavljuje pre pojavljivanja samog CrackMea zakljucujemo da se on nalazi u DLGProcu DialogBoxParamA APIja koji je zasluzan za prikazivanje .res dialoga. Dakle ako analiziramo samo OEP nase mete videcemo:

```
00407F72  .  6A 00            PUSH 0                          ; /lParam = NULL
00407F74  .  68 F47D4000      PUSH Keygen.00407DF4            ; |DlgProc = Keygen.00407DF4
00407F79  .  6A 00            PUSH 0                          ; |hOwner = NULL
00407F7B  .  6A 64            PUSH 64                         ; |pTemplate = 64
00407F7D  .  FF35 4C984000    PUSH DWORD PTR DS:[40984C]
00407F83  .  E8 84C5FFFF      CALL <JMP.&user32.DialogBoxParamA>
```

A posto znamo da se DLGProc koristi za obradu poruka koje dobija od mete postavicemo jedan break-point na DLGProc adresu, postavicemo break-point na 00407DF4, restartovacemo Olly i pritiskom na *F9* naci cemo se na nasem break-pointu. Sada nam samo ostaje da pritiskamo *F8* sve dok se ne pojavi nas trazeni NAG. Ali posto se sam DLGProc ponavlja veci broj puta lakse ce

nam biti da analiziramo ceo CALL nego da traceujemo. Analizom smo zakljucili da nam je interesantan samo deo od 00407E77 do 00407EC4 jer se on koristi za prikazivanje glavnog dijaloga. Ako pogledamo taj kod videcemo:

```
00407E77  |> \68 DC7E4000    PUSH Keygen.00407EDC
00407E7C  |.  A1 4C984000    MOV EAX,DWORD PTR DS:[40984C]
00407E81  |.  50             PUSH EAX
00407E82  |.  E8 A5C6FFFF    CALL <JMP.&user32.LoadIconA>
00407E87  |.  A3 54984000    MOV DWORD PTR DS:[409854],EAX
00407E8C  |.  A1 54984000    MOV EAX,DWORD PTR DS:[409854]
00407E91  |.  50             PUSH EAX
00407E92  |.  6A 00          PUSH 0
00407E94  |.  68 80000000    PUSH 80
00407E99  |.  56             PUSH ESI
00407E9A  |.  E8 ADC6FFFF    CALL <JMP.&user32.SendMessageA>
00407E9F  |.  A1 54984000    MOV EAX,DWORD PTR DS:[409854]
00407EA4  |.  50             PUSH EAX
00407EA5  |.  6A 01          PUSH 1
00407EA7  |.  68 80000000    PUSH 80
00407EAC  |.  56             PUSH ESI
00407EAD  |.  E8 9AC6FFFF    CALL <JMP.&user32.SendMessageA>
00407EB2  |.  A1 88824000    MOV EAX,DWORD PTR DS:[408288]
00407EB7  |.  50             PUSH EAX
00407EB8  |.  56             PUSH ESI
00407EB9  |.  E8 9EC6FFFF    CALL <JMP.&user32.SetWindowTextA>
00407EBE  |.  8935 50984000  MOV DWORD PTR DS:[409850],ESI
00407EC4  |.  E8 C3FDFFFF    CALL <JMP.&serial.ShowNag>
```

I kao sto se vidi iz ove analize prvo se ucitava ikona dijaloga, pa se ta ikona postavlja, pa se postavlja naziv samog dijaloga, dok se na samom kraju poziva dll fajl serial.dll i u njemu funkcija ShowNag... Mislim da smo pronasli nas NAG! Sada mozemo da uradimo patchovanje ovde pretvarajuci ovaj CALL u NOP ili mozemo da odemo u .dll fajl i da NAG patchujemo tamo.

## Deo II - Patching a NAG in .dll file

Drugi deo ovog dela poglavlja cemo posvetiti patchovanju samog .dll fajla. Ovo se takodje moze uraditi pomocu referentnih stringova ali mi cemo to uraditi na sasvim drugaciji nacin. Naime iskoristicemo cinjenicu da vec znamo naziv export funkcije u .dll fajlu koja se koristi za prikazivanje NAGa. Stoga cemo otvoriti serial.dll fajl pomocu prozora View names (*CTRL+N*) i tamo cemo naci nasu export funkciju:

```
Names in serial, item 64
 Address=1000100A
 Section=.text
 Type=Export
 Name=ShowNag
```

Duplim klikom na ovaj red doci cemo do mesta na kojem se nalazi kod vezan za samu eksport funkciju:

```
1000100A > /E9 51010000     JMP serial.10001160
```

Posto je u pitanju JMP skok praticemo ga do pocetka koda koji se koristi za prikazivanje NAGa:

```
1000117A   6A 40            PUSH 40
1000117C   68 44900210      PUSH serial.10029044   ; ASCII "NAG :OP"
10001181   68 1C900210      PUSH serial.1002901C   ; ASCII "This is a NAG screen, kill it!!!"
10001186   6A 00            PUSH 0
10001188   FF15 74030310    CALL <USER32.Message>
```

I konacno ovde mozemo da uradimo standardan postupak za uklanjanje MessageBoxA NAGova. Kada zavrsimo sa patchovanjem program cemo snimiti klikom na *desno dugme -> Copy to executable -> Save all -> Save...*

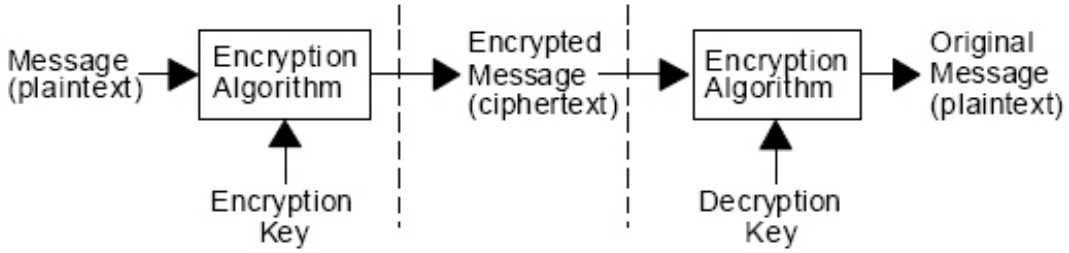
# 09 Decrypt me...

Neke lake dekrpicije na izgled kompleksnih problema. Ovde cete nauciti kako da brzo, lako i jednostavno razbijete lake enkripcije. Ovo poglavlje takodje sadrzi i deo posvecen bruteforcerima. Taj deo poglavlja nije ni malo lak i moracete da se potrudite kako biste stekli kompletnu sliku vezanu za ovu vrstu programa, ali se nadam da cete sve razumeti.

# Cryptography basics

Kriptografija je deo informatike koja se bavi osiguravanjem cuvanja podataka i komunikacije. Najpoznatije polje kriptografije je enkripcija koja se zasniva na tehnologiji izmene razumljivih podataka na specifican nacin tako da podaci postaju necitljivi, odnosno nerazumljivi prilikom obicnog posmatranja. Naravno postoji i suprotna operacija u odnosu na enkripciju i ona se zove dekripcija.

U kriptografiji podaci koji se enkriptuju, odnosno dekriptuju imaju specijalna imena i ona su: za neenkriptovane podatke *plaintext* a za vec enkriptovane podatke *ciphertext*. Uopsteni princip koriscenja neke vrste enkripcije je prikazan na sledecem grafiku:




Dakle enkripcija se zasniva na tri krucijalna podatka: plaintextu, algoritmu koji vrsi enkripciju i kljucu na osnovu koga se vrsi enkripcija. Analogno ovome dekripcija se vrsi pomocu algoritma za dekripciju, koji moze biti isti kao i algoritam za enkripciju, i kljuca pomocu kojeg se vrsi dekripcija koji takodje moze biti isti kao i kljuc za enkripciju. Kao rezultat dekripcije se dobija dekriptovana poruka sa pocetka ove sheme, odnosno plaintext.

Ovde treba naglasiti da snaga algoritma ne lezi u njegovoj tajnosti nego naprotiv, ona se ogleda u broju istrazivanja vezanih za njega i takozvanih "napada" na sam algoritam u cilju pronalazenja njegovih slabosti. Kriptografi su tokom godina razvili veliki broj tehnika za razbijanje slabih algoritama i slabih kljuceva, tako da ce veoma slabi algoritmi biti veoma brzo slomljeni, cak i bez odgovarajuceg kljuca. Ali ne dajte se zavarati, ni jedan algoritam za enkripciju nije nesalomiv, niti tako nesto tvrde njegovi autori, ali postoji velika verovatnoca da se ovo nece dogoditi ili da je za njegovo "razbijanje" potreban izuzetno velik vremenski period. Bez obzira na razlicite vrste napada na odredjeni algoritam, svaki algoritam je siguran samo dok su kljucevi za enkripciju odnosno dekripciju cuvani u tajnosti.

# Symetric and asymetric algorithms

Postoje dve klasicne vrste algoritama za enkripciju odnosno dekripciju. Ove dve vrste se nazivaju simetricni i asimetricni algoritmi. Glavna razlika izmedju ove dve vrste algoritama je u vrsti kljuceva koji se koriste za enkripciju, odnosno dekripciju. Ova razlika se odnosi na primenu jednog ili vise kljuceva za procese enkripcije/dekripcije. Tacnije simetricni algoritmi prilikom enkripcije i dekripcije koriste isti set kljuceva koji su isti za obe operacije, dok se kod asimetricnih algoritama koriste dva seta kljuceva,

jedan za enkripciju i jedan za dekripciju. Primer za koriscenje simetricnog algoritma bi bio koriscenje algoritma koji transformise nasu plaintext poruku u ciphertext pomocu jednog specificnog kljuca, odnosno sledeci Delphi kod:

```
function EncryptDecrypt(inputhash:string;key:integer):string;
var
 i:integer;
begin
        for i := 1 to Length(inputhash) do begin
                inputhash[i] := Chr(Ord(inputhash[i]) XOR key);
        end;
                EncryptDecrypt := inputhash;
end;
```

Ovaj kod jednostavnim XORovanjem svakog slova sa KEY vrednoscu plaintext pretvara u ciphertext. Posto znamo da je operacija XOR reverzibilna, zakljucujemo da ce se algoritam ponasati tako da isti kljuc enkriptuje plaintext u ciphertext i obrnuto. Ovo je samo jedan primer kada se kod simetricnog algoritma koristi samo jedna funkcija za enkripciju, odnosno dekripciju. Nasuprot tome postoje i algoritmi koji koriste posebne procedure za enkripciju i dekripciju. Posto se i za ovakav tip algoritama koristi isti kljuc za enkripciju i dekripciju, i ovi algoritmi pripadaju grupi simetricnih algoritama. Primer za ovo bi bio:

```
function Encrypt(inputhash:string;key:integer):string;
var
 i:integer;
begin
        for i := 1 to Length(inputhash) do begin
                inputhash[i] := Chr(Ord(inputhash[i]) + key);
        end;
                Encrypt := inputhash;
end;

function Decrypt(inputhash:string;key:integer):string;
var
 i:integer;
begin
        for i := 1 to Length(inputhash) do begin
                inputhash[i] := Chr(Ord(inputhash[i]) - key);
        end;
                Decrypt := inputhash;
end;
```

Ovde se za enkripciju koristi tehnika pomeranja slova plaintexta za vrednost kljuca u pozitivnu stranu, dok se prilikom dekripcije ciphertext pomera u negativnu stranu. Posto su matematicke operacije, plus i minus suprotne logicno je da ce se za enkripciju i dekripciju koristiti isti kljuc.

Sa druge strane, u odnosu na simetricne algoritme, nalaze se asimetricni algoritmi koji za enkripciju i dekripciju koriste razlicite kljuceve. Ova dva kljuca moraju biti povezana nekom, izuzetno slozenom matematickom relacijom na nacin pomocu koga se iz jednog moze dobiti drugi. Ovo znaci da se na osnovu istih podataka racunaju i jedan i drugi kljuc ali se sami zasebno tesko mogu povezati. Posto se jedan kljuc koristi samo za enkripciju a jedan samo za dekripciju mozemo slobodno reci da se ova dva kljuca takodje mogu podeliti na privatni i javni kljuc. Ovo znaci da u

zavisnosti od tipa sigurnosti koju implementiramo mozemo jedan kljuc proglasiti javnim i omoguciti njegovo preuzimanje svima, dok drugi cuvamo kao tajni. Kako je ovo moguce?

Posto se za operacije enkripcije i dekripcije koriste razliciti kljucevi nemoguce je izvrsiti operaciju za koju nemamo odgovarajuci kljuc. Ovo znaci da ako neko ima kljuc za dekripciju on moze samo da cita poruke zasticene tim kljucem ali ne i da ih proizvodi i salje. Naravno i suprotno vazi. Ali sta ako vec znamo jedan kljuc i ako znamo matematicku relaciju izmedju poznatog i nepoznatog kljuca. Da li je tada moguce pronalazenje drugog, nepoznatog, kljuca na lak i brz nacin? Ne, ovo nije moguce jer je matematicki algoritam dizajniran tako da cak i ako poznajete javni kljuc nikako ne mozete doci u posed privatnog kljuca, osim naravno ako ne pokusate da brute-force metodom dodjete do ovog kljuca, ali o tome ce biti vise reci kasnije.

Dakle asimetricni algoritmi su jako sigurni i stoga mogu biti implementirani na mestima gde je sigurnost podataka od izuzetne vaznosti, bez bojazni od njihovog brzog "razbijanja". Primer upotrebe asimetricnih algoritama bi bila komunikacija putem interneta gde se izmedju dva korisnika vrsi tajni saobracaj. Poruke koje korisnici salju jedni drugima su enkriptovane nekim asimetricnim algoritmom, dok se na njihovom odredistu vrsi dekripcija pomocu, u ovom slucaju, tajnog kljuca. Da bi ovakav tip komunikacije radio, oba korisnika moraju imati oba seta kljuceva jer je potrebno i vrsenje enkripcije i vrsenje dekripcije poruka.

## Bruteforceing encryption keys

Vec je receno da se tajni kljuc bilo koje simetricno/asimetricne enkripcije moze saznati pomocu brute-force tehnike. Ali da li je to stvarno isplativo?

Brute-force napad je nista drugo do testiranja svih mogucih kljuceva u potrazi za onim pravim. To znaci da cemo napisati algoritam koji ce pokusati da dekriptuje tajnu poruku svim mogucim tajnim kljucevima. Uspeh ovakvog nacina trazenja tajnog kljuca je zagarantovan, ali koliko je stvarno vremena potrebno za pronalazak jednog ovakvog kljuca?

Enkripcioni algoritmi koriste veoma dugacke kljuceve kojim se enkriptuje neka poruka zbog cega ce brute-force trajati izuzetno dugo. Primera radi ako koristimo algoritam koji vrsi enkripciju pomocu 128bitnog kljuca broj mogucih kombinacija validnih kljuceva iznosi $2^{128}$, odnosno 2 na 128 stepen. Ovo znaci da bi cak i sa masinom koja moze da proveri 1000 biliona kljuceva (priblizno $2^{40}$) kljuceva u sekundi bilo potrebno $2^{88}$ sekundi kako bi se proverile sve mogucnosti. Posto jedna godina ima priblizno $2^{25}$ sekundi bilo bi potrebno $2^{63}$ godina da bi se proverile sve mogucnosti, ova cifra ima preko 20 cifara. Ovaj primer se odnosio na Rijndael simetricni nacin enkripcije sa kljucem od 128bita.

Bez obzira na izuzetno veliko vreme razbijanja, simetricni kljucevi su ograniceni velicinom zbog cega se njihova sigurnost nikako ne moze meriti sa asimetricnim algoritmima. Primera radi simetricni algoritam DES koji koristi

56to bitne kljuceve moze biti razbijen za samo 9 sati na izuzetno mocnim racunarima ili sto je u praksi i dokazano za par meseci kolektivnog mreznog rada vise stotina racunara. Ovaj mrezni pristup je 1997 pokazao da se DES moze izuzetno brzo razbiti.

Nasuprot veoma male velicine kljuceva kojima raspolazu simetricni algoritmi stoje asimetricni algoritmi ciji su najmanji kljucevi duzine 128bita. Duzine ovih kljuceva mogu ici i do 1024bita sto komunikaciju i cuvanje podataka cini izuzetno sigurnim. Jedina prednost koju simetricni algoritmi imaju nad asimetricnim je brzina dekripcije odnosno enkripcije zbog cega za enkripciju vece kolicine podataka ne bi trebalo koristiti asimetricne algoritme.

## Hashing and digital signing

Iako je priblizno svima jasan znacaj kriptografije, njena upotreba nije ogranicena samo na enkripciju i dekripciju podataka.

*Hashing* je izuzetno bitan derivat standardnih enkripcionih algoritama. Ova tehnika se zasniva na pravljenju specificnog broja, koji se naziva hash, a koji biva dodeljen specificnom sadrzaju. Ovaj sadrzaj moze biti ili string ili sadrzaj nekog fajla manifestovan kako memory stream. Sta radi hashing algoritam?

On jednostavno uzima svaki bajt iz posmatranog objekta i na osnovu njega racuna unikatan broj koji sluzi za proveru verodostojnosti datog podatka. Ovo znaci da svaki objekat ima svoj jedinstveni hash i da samo potpuno isti objekti mogu imati isti hash. Naravno ovde se pod potputno isti objekti misli na objekte cija je bajtovna struktura identicna. Kao primeri ovakvih algoritama se srecu CRC, MD5 i SH1 algoritmi.

CRC ili Cyclic Redudancy Check je sistem generisanja hasha najcesce na osnovu sadrzaja nekog fajla. Ovakav hash se koristi za verodostojnost transfera nekog podatka putem interneta ili kao verifikator tacnosti neke arhive. Ovaj algoritam se moze koristiti ali ne predstavlja siguran nacin verifikacije posto je CRC algoritam u potpunosti reversovan tako da se svaki fajl moze falsifikovati na takav nacin da dobije zeljenu vrednost CRC hasha.

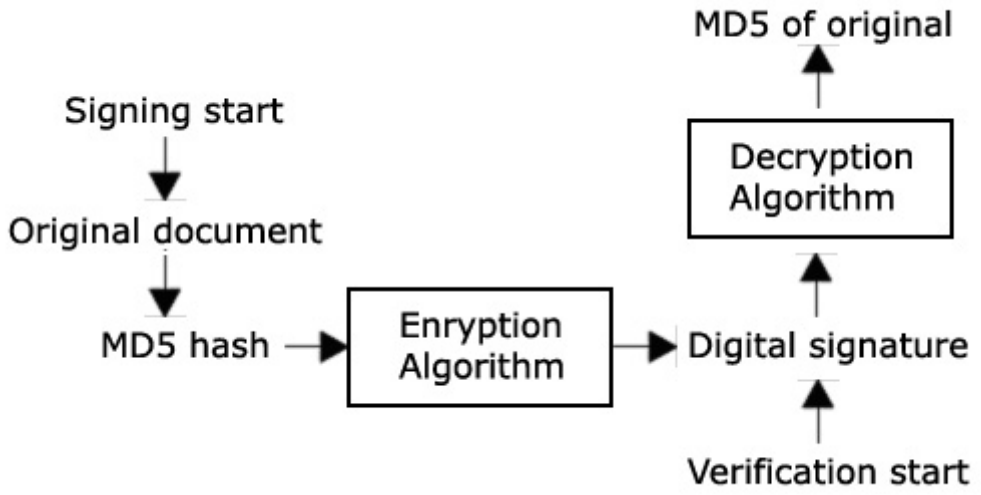
MD5 ili Message Digest 5 proizvodi 128bitne hashove na isti nacin kao i CRC algoritam samo koristeci se drugacijim algoritmom. Ovaj algoritam je za razliku od CRCa kompleksniji i napisan u kriptografske svrhe. Ovo znaci da je mogucnost sudaranja dve vrednosti, tako da dva stringa imaju isti MD5 hash, veoma mala i da je bez dodatne intervencije prakticno nemoguce postojanje dva ista MD5 hasha za dva razlicita fajla. Bez obzira na ove tvrdnje pre nekog vremena je pronadjen nacin za falsifikovanje MD5 algoritma tako da je sada moguce falsifikovanje MD5 potpisa na ‘isti’ nacin na koji je to moguce sa CRCom.

SHA-1 ili Secure Hashing Algorithm je algoritam koji je slican MD5 algoritmu sa razlikom sto SHA1 proizvodi 160bitne hashove.

Naravno samo lepljenje hashova za dokument nije dovoljno da bi se osigurala njegova verodostojnost, to jest da bi se osigurao njegov integritet. Ovo nije moguce iz razloga sto bi svako mogao da izmeni dokument, da

izracuna novi hash i da zameni postojeci hash sa novim. Na ovaj nacin bi modifikovani fajl izgledao kao originalni. Da bi se ovo prevazislo i da bi integritet fajla bio osiguran, koriste se sistemi digitalnog potpisivanja. Ovaj sistem predstavlja kombinaciju asimetricnih algoritama za enkripciju podataka i algoritama za kreiranje hashova. Kako radi digitalni podpis?

Ako pogledamo sledeci grafik:




videcemo da se potpisivanje dokumenata vrsi pomocu dva kratka koraka, pomocu racunanja MD5 hasha dokumenta i pomocu njegove transformacije u digitalni potpis enkriptovanjem. Sama enkripcija se uvek vrsi pomocu nekog asimetricnog algoritma posto je potrebno da se uz dokument dostavi i kljuc za dekripciju digitalnog potpisa, kako bi isti bio verifikovan. Da li je sigurno izdavanje javnog kljuca za dekripciju? U ovom slucaju to je sigurno jer se koristi neki od asimetricnih algoritama pa se pomocu istog kljuca (*kljuca za dekripciju*) ne moze falsifikovati MD5 hash originalnog fajla. Da bismo falsifikovali ovakav digitalni potpis potrebno je da znamo kljuc za enkripciju koji se u ovakvim slucajevima cuva u tajnosti.

Suprotno od digitalnog potpisivanja je digitalna verifikacija koja pomocu dva dostupna podatka verifikuje verodostojnost dokumenta. Ova dva podatka su javni kljuc za dekripciju i digitalni potpis. Pomocu javnog kljuca digitalni potpis biva dekriptovan, a ovako dobijena vrednost se poredi sa MD5 hashom posmatranog fajla. Ako su ove dve vrednosti iste, dokument je autentican. Najcesce korisceni algoritam za enkripciju za RSA (Rivest Aldman Shamir) o kome ce vise reci biti kasnije.

Imajte na umu da se pojmovi javnog i tajnog kljuca razlikuju kod enkripcije i digitalnog potpisivanja. Ovo znaci da je kod enkripcije javni kljuc, kljuc za enkripciju a tajni onaj za dekripciju, dok se kod digitalnog potpisivanja kao javni kljuc koristi onaj za dekripciju a kao tajni onaj za enkripciju.

# Simple Encryption

Ako ste citali deo poglavlja sedam o CRC proverama primeticete da sam se dotakao jednostavne enkripcije koja se moze sresti u nekim programima. Kao primer sam tamo naveo enkripciju koju koristi Trillian pa cemo mi upravo na ovom programu nauciti osnove kriptografije. Naravno ovo je obicno XORovanje i jedva se moze nazvati kriptografijom ali od neceg mora da se pocne. Pre nego sto se damo u resavanje problema osvrnucemo se malo ka istoriji, ka prvim primenama kriptografije. Dakle, prva (*meni poznata*) upotreba kriptografije se moze sresti za vreme vladavine Cezara. Kako bi zastitio poruke koje su nosili kuriri Cezar je primenjivao osnovnu kriptografiju. Kljuc za njeno resavanje je bio jednostavno pomeranje slova za tri. Dakle ako je slovo A onda bi ovako enkriptovano slovo bilo C, i tako dalje. Naravno pri dekripciji Rimljani bi radili suprotan postupak, to jest pomerali slova po tri ali unazad. Ova mala prica o istoriji je tu da vam samo predoci osnovu kako to kriptografija radi. Bez obzira koji se algoritam koristi uvek morate imati tri osnovne jedinice: tekst koji se enkriptuje, kljuc za enkripciju i algoritam koji vrsi enkriptovanje uz pomoc kljuca. Algoritmi se mogu razlikovati i tako kao danas najpopularnije komplikovane algoritme poznajemo: RC4, RC5, AES, DES, BlowFish, Gost, TEA,.... Nemojte se zanositi da cete nauciti kako se razbijaju ove slozene enkripcije. Ove enkripcije se ne mogu razbiti bez odgovarajuceg kjuca. Da biste pronasli pravi kljuc preostaje vam samo da radite bruteforce ali to moze da potraje dosta, dosta dugo i  na najjacim kompjuterima. Enkripcija koja se nalazi unutar jednog Trillian password fajla je veoma prosta i nju cemo razbiti u ovom poglavlju. Ako otvorimo msn.ini fajl koji se nalazi u Cas9 folderu videcemo ovo:

```
[msn]
auto reconnect=1
save passwords=1
idle time=15
show buddy status=1
port=1863
server=messenger.hotmail.com
last msn=ap0x@hotmail.com
connect num=10
connect sec=60
save status=1
ft port=6891
[profile 0]
name=ap0x@hotmail.com
password=C214B2F00CB0ECAA48
display name=Ap0x
auto connect=0
status=1
```

Ono sto nas interesuje je enkriptovan password. Ono sto ja znam a vi ne je koji sam ja to password uneo. Uneo sam 123456789 kao password sto ce nam omoguciti da provalimo nacin na koji se password 123456789 enkriptuje u C214B2F00CB0ECAA48. Ono sto moramo da primetimo je da je duzina neenkriptovanog passworda 9 a enkriptovanog 18. Ovo moze da znaci da svakom broju iz neenkriptovanog passworda odgovaraju dva iz enkriptovanog passworda. Ako rastavimo enkriptovani password na cinioce imacemo ovo:

**C2 14 B2 F0 0C B0 EC AA 48**

Dakle sada mozemo da uporedjujemo enkriptovane brojeve sa neenkriptovanim. Zakljucicemo da je ASCII od 1 XOR neki broj jednako C2 u hex formatu ili 194 u decimalnom (*za pretvaranje koristite windows calculator*). Dakle da bismo otkrili koji je to broj sa kojim se ASCII xoruje moramo da znamo: 1) ASCII vrednost svakog karaktera iz neenkriptovanog passworda i 2) decimalne vrednosti hex parova iz enkriptovanog passworda. Za prvi uslov pogledajte ASCII tablicu koja se nalazi na www.asciitable.com a za drugi koristite Windows Calculator. Jednacine koje mi moramo da resimo su:
49 XOR x = 194, x = ?
50 XOR y = 020, y = ?
51 XOR z = 178, z = ?
....
Ova jednacina se resava veoma lako. x = 49 XOR 194; y = 50 XOR 20; z = 51 XOR 178,.... Ovo je jednako posto je XOR funkcija reverzibilna. Tablica sa Hex i x,y,z,... vrednostima:

| ASCII | XOR | REZULTAT | HEX |
|---|---|---|---|
| 49 | 243 | 194 | C2 |
| 50 | 38 | 20 | 14 |
| 51 | 129 | 178 | B2 |
| 52 | 196 | 240 | F0 |
| 53 | 57 | 12 | 0C |
| 54 | 134 | 176 | B0 |
| 55 | 219 | 236 | EC |
| 56 | 146 | 170 | AA |
| 57 | 113 | 72 | 48 |

Kao sto se vidi iz tablice dobili smo vrednosti sa kojima se XORuju ASCIIji svih slova iz neenkriptovanog passworda. Koristeci ovu semu sada mozemo dekriptovati bilo koji password koji je Trillian enkriptovao ili mozemo cak napisati i program koji ce raditi dekripciju enkriptovanog passworda. Program koji dekriptuje Trillian password bi izgledao ovako:

```
Dim table(1 To 9) As Integer
 table(1) = 243
 table(2) = 38
 table(3) = 129
 table(4) = 196
 table(5) = 57
 table(6) = 134
 table(7) = 219
 table(8) = 146
 table(9) = 113
 x = 0
 dec = ""
 enc = Text1.Text                 'Ovo je prvi TextBox koji sadrzi enkriptovani password
                                  'U njega zalepite enkriptovani password bez razmaka
For i = 1 To Len(enc) Step 2
 x = x + 1
 dec = dec & Chr(Val("&H" + Mid$(enc, i, 2)) Xor table(x))
Next i
 Text2.Text = dec                 'Ovo je drugi TextBox koji sluzi za ispisivanje dekriptovanog
                                  'passworda
```

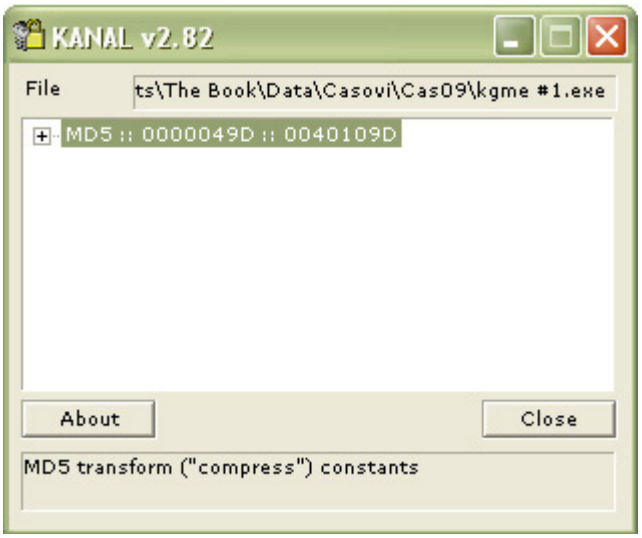
Ovaj program mozete probati na drugom yahoo.ini fajlu koji se nalazi u istom direktorijumu.

# Reversing MD5 encryption

Vec je bilo reci o standardnim nacinima enkripcije i vecina najpopularnijih je nabrojana. Sada cemo objasniti kako se to reversuje meta koja koristi jednu ovakvu enkripciju. Kazem enkripciju iako to MD5 sigurno nije. MD5 je metoda generisanja niza brojeva koju su jedinstveni za ulazne podatke. Odnosno ako MD5 hashing algoritmu prosledite neku rec kao parametar dobicete heksadecimalni broj duzine 32. Ovaj broj je kao sto je vec receno jedinstven i ne postoji neki drugi string koji bi imao isti MD5 hash kao bilo koja druga rec. Dakle ovakav nacin hashovanja stringova je prilicno siguran posto je skoro nemoguce "provaliti" koji je string koriscen da bismo dobili neki MD5 hash. Naravno jedina opcija koja nam preostaje je da uradimo  bruteforce da bismo dosli do originalnog stringa koji je hashovan.

Srecom po nas za razbijanje hashovanih serijskih brojeva nije potreban nikakav bruteforce. Ovo je moguce samo iz razloga sto svi programi koji koriste MD5 hashing prvo racunaju serijski broj na osnovu unetog podatka a onda dobijeni serijski pretvaraju u MD5 hash koji se poredi sa unetim serijskim brojem ili se na njemu izvrsava jos neka operacija.

Primer ovakvog koriscenja MD5 hashinga se nalazi u meti kgme #1.exe koja se nalazi u folderu Cas09. Pre nego sto pocnemo da reversujemo ovu metu uvek bi trebali da uradimo skeniranje mete u potrazi za kripto potpisima. Ovo mozemo uraditi pomocu PeID plugina koji se zove KANAL ili pomocu x3chung Crypto Searchera. Na slici pored se vidi rezultat koji se dobija kada metu skeniramo sa PeIDovim pluginom.




Kao sto vidimo u nasoj meti se nalazi samo jedan jedini kripto potpis a on se odnosi na MD5. Naravno postoje mete koje koriste veci broj kripto algoritama da bi generisale ili proverile serijske brojeve. Ovi algoritmi mogu biti MD5,SHA1,RSA,DSA,AES,...

Prvo sto treba da uradimo da bismo pronasli pravi serijski broj za nase ime je da pronadjemo gde se porede uneti serijski broj i tacan serijski broj. Ovo je izuzetno lako ako ste citali pazljivo knjigu do sada. Dakle znamo da ce program posle unosa serijskog broja sigurno proci ovaj deo koda:

```
00401AD8   .  E8 33010000     CALL <JMP.&USER32.GetDlgItemTextA>    ; \GetDlgItemTextA
00401ADD   .  0BC0            OR EAX,EAX
00401ADF   .  75 17           JNZ SHORT kgme_#1.00401AF8
```

i proveriti da li je serijski broj unet ili ne. Naravno ako je serijski broj unet onda ce se JNE skok izvrsiti i program ce doci do sledece adrese,

```
00401AF8   > \68 34334000     PUSH kgme_#1.00403334         ; /StringToAdd "BytePtr [e!]"
00401AFD   .  68 80334000     PUSH kgme_#1.00403380         ; |ConcatString = "ap0x"
00401B02   .  E8 EB000000     CALL <JMP.&KERNEL32.lstrcatA> ; \lstrcatA
```

na kojoj se, kao sto vidimo na nase uneto ime dodaje string `BytePtr [e!]` posle cega ce nase ime izgledati ovako `ap0xBytePtr [e!]'. Posle ove operacije se izvrsava sledeca MD5 kripto operacija:

```
00401B08  . 68 A8564000     PUSH kgme_#1.004056A8      addr 004056A8
00401B0D  . 50              PUSH EAX                   addr 00000004
00401B0E  . 68 80334000     PUSH kgme_#1.00403380      ASCII  "ap0xBytePtr [e!]"
00401B13  . E8 E8F4FFFF     CALL kgme_#1.00401000      addr 00401000
```

Ovaj veoma jednostavan CALL ka MD5 funkciji ima samo tri ulazna parametra: 1) adresa na kojoj ce se naci MD5 hash posle izvrsavanja MD5 hashing funkcije, 2) duzina stringa koji se pretvara u MD5 hash i 3) string koji se pretvara u MD5 hash.

Kao sto vidimo ovde se nalazi jedna zamka koja nas moze drzati u zabludi veoma dugo! Obratite paznju da se kao string koji se pretvara u MD5 hash koristi `ap0xBytePtr [e!]' ali da se kao duzina stringa koji se pretvara pojavljuje broj cetiri. Ovo znaci da se nece sva slova pretvoriti u MD5 hash nego ce se za pretvaranje koristiti samo prva cetiri slova od ovog stringa, odnosno koristice se bas nase uneto ime `ap0x'. I konacno posle izvrsenja CALLa za pretvaranje stringa u MD5 hash na adresi 004056A8 ce se nalaziti sledece:

```
004056A8  AF 36 BC 6E BB AA 52 0E  .6.n..R.
004056B0  8F 20 CF 74 9A 07 E5 AE  . .t...®
```

Ako upotrebimo HashCalc (www.slavasoft.com) videcemo da je MD5 hash za nase ime (ap0x) jednak 6ebc36af0e52aabb74cf208faee5079a. Ocigledno je da je ovaj MD5 hash podeljen na cetiri dela i da se sva cetiri dela posmatraju kao DWORD. To jest posmatracemo sadrzaj adrese 004056A8 kao MD5_1, MD5_2, MD5_3 i MD5_4. Gde je MD5_x jednak jednakim delovima MD5 hasha ali su u memoriji DWORDovani, to jest okrenuti.

Posle pretvaranja naseg imena u MD5 hash poziva se sledeci CALL:

```
00401B18  . E8 5C000000     CALL kgme_#1.00401B79
```

koji je zaduzen za dodatne operacije koje se izvrsavaju na sad kreiranom hashu. Te operacije se izvrsavaju na sledecim adresama i na sledeci nacin:

```
00401B84  |. 8B06           MOV EAX,DWORD PTR DS:[ESI]          ; MD5_1
00401B86  |. 8B5E 04        MOV EBX,DWORD PTR DS:[ESI+4]        ; MD5_2
00401B89  |. 33C3           XOR EAX,EBX
...
00401B9D  |. 81F3 9900BD0F  XOR EBX,0FBD0099                    ; XOR MD5_2
...
00401BB2  |. 8B4E 08        MOV ECX,DWORD PTR DS:[ESI+8]        ; MD5_3
00401BB5  |. 33CB           XOR ECX,EBX                         ; XOR EBX
...
00401BC9  |. 81F3 090A0C0B  XOR EBX,0B0C0A09
00401BCF  |. 8B56 0C        MOV EDX,DWORD PTR DS:[ESI+C]        ; MD5_4
```

a posle vracanja iz ovog CALLa vidimo na koji nacin se promenio MD5 hash koji se sada poredi sa nasim unetim serijskim brojem:

```
00401B1D  . 68 D0794000  PUSH 004079D0    "60EE9C1401EFAA2275208AADAEE5079A"
00401B22  . 68 F89C4000  PUSH 00409CF8    "11111111111111111111"
00401B27  . E8 CC000000  CALL <JMP.&KERNEL32.lstrcmpA>
```

Dakle nacin na koji se generise seriski broj je sledeci:

[1] Uzimamo ime i hashujemo ga sa MD5 deleci ga na cetiri dela

[2] Konacna formula za dobijanje serijskog broja je sledeca

Serial_1 = MD5_1 xor MD5_2

Serial_2 = MD5_2 xor 0FBD0099

Serial_3 = MD5_3 xor EBX [Serial_2]

Serial_4 = MD5_4 ; *Serial = Serial_1 + Serial_2 + Serial_3 + Serial_4*

# RSA Basics

RSA (Rivest Shamir Adleman) je algoritam za enkripciju podataka koji se oslanja na velike proste brojeve kako bi enkriptovao podatke u veoma sigurne hashove. Ovako enkriptovani stringovi se uz pomoc odgovarajuceg kljuca mogu dekriptovati u pocetne stringove. Ono sto karakterise RSA algoritam je postojanje posebnih kljuceva za enkripciju i dekripciju, sto se popularno zove asimetricna enkripcija, odnosno dekripcija. Ovo ujedno i znaci da ce za razbijanje ovakvog sistema biti potrebna izuzetna kolicina vremena posto je potrebno pronaci pocetni kljuc kako bi se falsifikovala poruka ili se enkriptovala nova uz pomoc istog kljuca. Osnova RSA algoritma, kao i svih drugih, lezi u specificnoj matematickoj formuli koja se koristi za enkripciju odnosno dekripciju. Osnovni ulazni parametri za RSA algoritam su:

**P** – Prvi veliki prosti broj; ulazni parametar
**Q** – Drugi veliki prosti broj; ulazni parametar
**E** – Javni eksponent; ulazni parametar
**N** – Javni modulus; racuna se na osnovu P i Q
**D** – Privatni eksponent; racuna se na osnovu P, Q i E.

Da pojasnim navedene pojmove. P i Q su osnovne enkripcione vrednosti i biraju se tako da su P i Q izuzetno veliki prosti brojevi, odnosno moraju da zadovolje uslov da su deljivi samo sa 1 i sa samim sobom. P i Q se koriste samo za enkripciju posle cega se oni dalje cuvaju kao tajni jer se mogu koristiti za falsifikovanje poruka.

E je javni eksponent koji se takodje koristi prilikom enkripcije ali i u dekripciji jer se na osnovu njega izracunava privatni eksponent D. Ovaj broj je najcesce jednak 10001h.

N je najvazniji deo sistema za dekripciju jer je on javno dostupan za razliku od argumenata P i Q. Naravno i N je veliki prost broj koji se dobija racunanjem na osnovu P i Q. Iz ovog razloga se bas N koristi za razbijanje RSA enkriptovanih poruka. Posto znamo gde se koji ulazni parametar koristi mozemo da predjemo na algoritme koji se koriste za enkripciju i dekripciju.

**Enkripcija** se izvodi po formuli:
C = M^E mod N
Gde je C rezultat enkripcije a M poruka koja se enkriptuje a koja mora da zadovolji uslov da je manja od javnog modulusa N. Ovo znaci da se za dugacke poruke moraju koristiti ili veliki kljucevi (256, 512bita) ili se poruka mora rastavljati na manje delove koji su manji od modulusa N. Imajte na umu da ^ nije XOR operacija nego funkcija power of!

**Dekripcija** se izvodi po formuli:
M=C^D mod N
Gde je M dekriptovana poruka a C enkriptovani tekst.

Sada mozemo da predjemo na razmatranje standardne primene RSA algoritma u aplikacijama. Naime RSA algoritam se u programima

(*crackmeima*) primenjuje tako da se u njima vec nalaze parametri za enkripciju E i N. Naravno ovo bi bila pravilna uporeba RSA, tako da se ne otkriva ni jedan tajni podatak koji bi nam omogucio da dodjemo do parametara P i Q lakse nego uobicajeno. Posto se u 99% aplikacija RSA primenjuje na pravilan nacin razmotricemo sledeci slucaj.

Reversujemo crackme koji u sebi sadrzi podatke N i E koji iznose:
N = AB185AE9243F57C7428B7CE76B62A5E1321CA3A3F966659A8368BD3879241F35407599BDC49EA31A9
E = 10001
Gde su ovi brojevi heksadecimalni. Recimo da nas crackme koristi nase ime da izracuna MD5 hash koji se poredi sa rezultatom dekripcije unetog serijskog broja. Odnosno:

serijski ^ E mod N = MD5hash

Ovo znaci da bismo pronasli pravi serijski broj za nase ime potrebno je da ga prvo hashujemo sa MD5 a tek onda da ga enkriptujemo pomocu RSA algoritma. Sada shvatate da su nam potrebni P i Q parametar to jest da bez njih ne mozemo nikako enkriptovati nas MD5 hash kako bismo dobili tacan serijski broj. Iz ovog razloga se radi takozvana faktorizacija (*razbijanje parametra N na tacno dva parametra P i Q koje je moguce samo iz razloga sto su i P i Q prosti brojevi pa je i njihov proizvod prost*) parametra N koja je moguca jer se parametar N racuna po formuli N = P * Q. Postoji vise vrsta napada na "slabe" RSA kljuceve ali nije potrebno da ih znamo, potrebno je samo da unesemo brojeve N i E u RSATool2v17 koji je napravio tE! i izaberemo Factor N funkciju, koja je ekvivalent bruteforceingu i stoga moze potrajati vise sati. Naravno kao rezultat faktorisanja parametra N dobijamo pocetne parametre P, Q i D koji se racuna na osnovu P,Q i E:

P = 2B53D5BFA34006A82D413E55EDF5867A8743A10B3
Q = 3F2E9F9F226FF94134CE8927DC45C2E407CB6AA33
D = A5389CB9DD5609F7130CCE6E4FE5F057FA636BA68B5B14672C7BD4DC26666F9A6BE23E30E2F12A01D

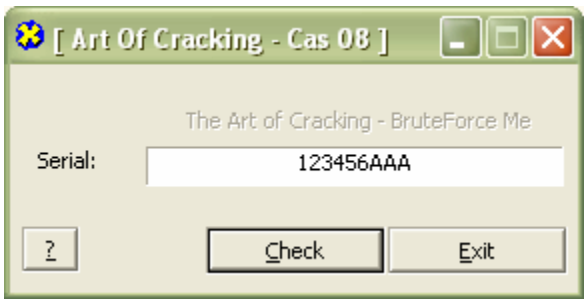
I sada konacno mozemo da napravimo keygenerator za nasu metu. Naravno ovo se mora odraditi reversno postupku provere tacnosti unetog serijskog broja. Dakle moramo prvo da hashujemo uneto ime sa MD5 i da onda pomocu parametara P,Q i E generisemo RSA enkriptovani string koji predstavlja tacan serijski broj za nase uneto ime.

Sada shvatate vaznost matematike prilikom reversovanja RSA algoritma. Ona je narocito izrazena u problemima kada RSA hashovanju prethodi niz matematickih operacija na osnovu kojih se generise broj koji se enkriptuje pomocu RSA. Bez obzira na to i dalje je potrebno dosta dugo vremena za faktorisanje parametra N, narocito ako je duzina ovog kljuca veca od 256 bita, sto cini RSA i dalje jednim od izuzetno pouzdanih algoritama za implementaciju, ali ga i dalje ne cini savrsenim.

# BruteForce #1

Pre nego sto sam napisao ovo poglavlje morao sam da napravim prvo algoritam koji bih morao da objasnim u ovom poglavlju. Od velikog broja mogucnosti odlucio sam se za onu najtezu za crackovanje, odlucio sam se za slucaj kada je jedini parametar od kojeg zavisi da li ce program biti uspesno crackovan sam serijski broj. Kao takav, serijski broj se u ovom slucaju ne racuna uz pomoc nekog hardware IDa ili nekog unetog parametra. Jedini potreban i dovoljan parametar samom algoritmu je serijski broj. Pogodnost ovakve vrste algoritma je to sto postoji beskonacan broj tacnih resenja, a samo resenje necete naci u programu. Zbog ove cinjenice potrebno je napisati program koji ce pronaci sva ili samo neka moguca resenja.
Otvorite program BruteForceMe.exe koji se nalazi u folderu Cas9 pomocu Ollyja. Namerno sam ubacio u program da on pokaze string Cracked OK kada ste pronasli tacan serijski broj. Ovo sam uradio da bih vama olaksao posao trazenja mesta gde se to racuna serijski broj. Videcete da CALL koji racuna serijski broj pocinje na adresi 00407D40. Analizirajte malo kod sami kako biste stekli sto vise informacija o algoritmu.

Sigurno ste primetili petlju koja pocinje na adresi 00407DE3. Postavicemo break-point na tu adresu i startovacemo program sa F9. U sam crackme unesite sledece. Zasto smo uneli bas ovo? Veoma jednostavno ako pritisnete dugme? videcete da sam vam ostavio malu pomoc kako bi trebalo da izgleda serijski broj. Ovo nije bas 100% isti format zapisa ali redosled brojeva i slova nema veze. Tako sam napravio ovaj crackme. Ako pritisnemo dugme Check program ce zastati na adresi 00407DE3 i mi cemo se kretati kroz ovaj loop red po red sa F8 sve dok ne saznamo sta to ovaj loop radi. Evo tog loopa izdvojenog i analiziranog:




```
MOV EAX,DWORD PTR SS:[EBP-C]          Prebaci u EAX uneti seriski broj
MOV DL,BYTE PTR DS:[EAX+EBX-1]        U DL stavi 1,2,3,... slovo seriskog
MOV EAX,EDX                           Prebaci ga u EAX, to jes u HEX
CMP AL,40                             Uporedi EAX sa 40 Hex
JBE SHORT BruteFor.00407E01           Ako je EAX <= 40 Hex onda preskoci...
MOV ECX,DWORD PTR SS:[EBP-C]          #
CMP AL,5B                             Uporedi HEX vrednost slova sa 5B
JNB SHORT BruteFor.00407E01           Ako nije manje ili jednako 5B preskoci..
MOV EAX,DWORD PTR SS:[EBP-C]          #
XOR EAX,EAX                           # Obrisi EAX
MOV AL,DL                             Prebaci slovo iz AL u DL
ADD DWORD PTR SS:[EBP-14],EAX         Saberi sadrzaj EBP-14 (pocetno 1) sa EAXom
MOV EAX,DWORD PTR SS:[EBP-C]          Prebaci u EAX ceo seriski broj...
MOV AL,BYTE PTR DS:[EAX+EBX-1]        Prebaci u AL slovo 1,2,3...
CMP AL,2F                             Uporedi AL sa 2F HEX...
JBE SHORT BruteFor.00407E2D           Ako je AL <= 2F Hex onda preskoci...
MOV EDX,DWORD PTR SS:[EBP-C]          #
CMP AL,3B                             Prebaci u AL slovo 1,2,3,4...
JNB SHORT BruteFor.00407E2D           Ako nije manje ili jednoko onda skoci na...
LEA EAX,DWORD PTR SS:[EBP-18]         #
MOV EDX,DWORD PTR SS:[EBP-C]          #
MOV DL,BYTE PTR DS:[EDX+EBX-1]        U DL prebaci slovo 1,2,3,...
CALL BruteFor.004037B8                #
MOV EDX,DWORD PTR SS:[EBP-18]         #
LEA EAX,DWORD PTR SS:[EBP-4]          Posle CALLa se brojevi dodaju jedan na drugi
CALL BruteFor.00403874                dole desno u HEX dump prozoru
INC EBX                               Povecaj brojac ciklusa
DEC ESI                               #
JNZ SHORT BruteFor.00407DE3           Loopuj
```

Sa leve strane se nalazi kod iz loopa a sa desne se nalaze objasnjenja ASM komandi iz loopa. Mislim da je svima jasno sta se desava u ovom loopu. Izdvajaju se brojevi i slova. Brojevi se dodaju jedan na drugi kao string a slova se pretvaraju u ASCII vrednosti i sabiraju (*add komanda*). Pocetna vrednost na koju se dodaju ASCII vrednosti je jedan.

Odmah posle ovog loopa na adresi 00407E31 pocinje novi segment koda. Mozete slobodno proci sa F8 kroz ovaj deo koda dok ne dodjete do adrese 00407F18. Izmedju 00407E31 i adrese 00407F18 se brojevi grupisu u parove. Znaci ako je uneti serijski broj 123456 onda ce postojati tri para brojeva i to su 12, 34, 56. Ako pogledamo ovaj deo koda odmah posle loopa:

```
00407E39  |.  83F8 06          CMP EAX,6
00407E3C  |.  0F85 1D010000    JNZ BruteFor.00407F5F
```

videcemo da se brojevna duzina (*123456*) poredi sa brojem 6. Ovo znaci da u samom serijskom broju mora postojati 6 cifara. Mozemo i da analiziramo posebno sta se sve to desava izmedju 00407E31 i 00407F18 ali nema potrebe jer je mnogo lakse shvatiti ovaj algoritam na ovom delu koda:

```
00407F20  |.  8BC8             MOV ECX,EAX
00407F22  |.  8BC6             MOV EAX,ESI
00407F24  |.  F7EB             IMUL EBX
00407F26  |.  F7E9             IMUL ECX
00407F28  |.  99               CDQ
00407F29  |.  F77D EC          IDIV DWORD PTR SS:[EBP-14]
00407F2C  |.  83FA 11          CMP EDX,11
00407F2F  |.  75 17            JNZ SHORT BruteFor.00407F48
```

Posle izvrsavanja komandi MOV ECX,EAX i MOV EAX,ESI videcemo sta se to nalazi u registrima EAX,ECX,ESI jer su nam ovi registri potrebni za IMUL komande ispod. Dakle posle izvrsavanja adrese 00407F22 imacemo u registrima sledece vrednosti:

```
EAX 0000000C                    <- Decimalno 12
ECX 00000038                    <- Decimalno 56
EDX 0012FC5C
EBX 00000022                    <- Decimalno 34
ESP 0012FC74
EBP 0012FCBC
ESI 0000000C
EDI 00860888 ASCII "123456AAA"
```

U tim registrima se nalaze oni parovi brojeva samo u hex obliku. Posle mnozenja, to jest posle izvrsavanja adrese 00407F28 imacemo u EAXu vrednost 5940h ili 22848 decimalno. A ako pogledamo proizvod 12 * 34 * 56 = 22848 i tako se dobija vrednost u EAXu. Na sledecoj adresi ovaj broj iz EAXa se deli sa vrednoscu adrese EBP-14 a ona je jednaka zbiru ASCII vrednosti svih slova iz imena plus jedan. Posle deljenja EAX ce biti jednak 74h ili 116 decimalno. Ovo nam govori da nije u pitanju obicno deljenje nego je u pitanju ostatak pri celobrojnom deljenju. Na sledecoj adresi se ovaj ostatak poredi sa 11h ili 17 decimalno. Ako je ostatak jednak 17 onda je serijski broj ispravan a ako je ostatak veci ili manji onda je serijski broj pogresan.

Posto postoji neogranicen broj tacnih serijskih brojeva mi moramo da napravimo program koji ce pronaci tacne vrednosti za uneta slova. Ova tehnika se moze poistovetiti sa buteforceingom. Algoritam cemo napraviti u VBu. Pre samog pravljenja ovog bruteforcera moramo da znamo sledece:

1) Da se serijski deli na tri dvocifrena broja i na slova
2) Da se tri dvocifrena broja medjusobno mnoze

3) Da se proizvod mnozenja deli sa zbirom ASCII vrednosti slova
4) Da se proverava da li je ostatak deljenja jednak 17

Na osnovu ove cetiri tacke mozemo da konstruisemo sledeci algoritam:

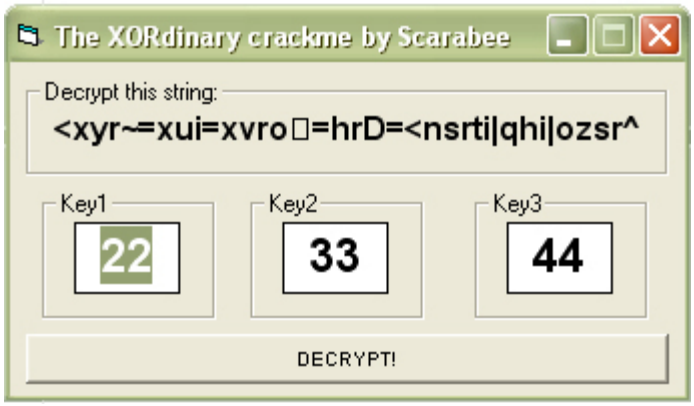
```
List1.Clear
 nic = 1                                  'Osnova za dodavanje je jedan
For i = 1 To Len(Text1.Text)
 nic = nic + Asc(Mid$(Text1.Text, i, 1))  'Dodavanje ASCII vrednosti slova
Next i
For y = 10 To 99                          'Posto se mnoze tri dvocifrena broja
 For x = 10 To 99                         'Moramo imati tri petlje od 10 do 99
  For i = 10 To 99
   If (i * x * y) Mod nic = 17 Then       'Proverimo da li je ostatak deljenja 17
    List1.AddItem x & i & y & Text1.Text  'Ako jeste serijski broj je ispravan
   End If
  Next i
 Next x
Next y
```

Imajte na umu da su ovo samo moguci serijski brojevi, sto znaci da nece svi generisani serijski brojevi pomocu ovog algoritma raditi. Ja sam vec napravio bruteforcer po ovom algoritmu i on se moze naci u istom folderu kao i sam crackme. Ako u bruteforcer unesete neki veliki string sa razmacima izmedju slova postoji velika verovatnoca da serijski broj nece raditi. Stoga se drzite unosenja do 4-5 slova, gde slova mogu biti samo velika.

Ovaj deo poglavlja je samo uvod u tematiku bruteforceovanja pa biste trebali da shvatite ovo kako biste ostatak ovog poglavlja lakse savladali. Iako je algoritam koji se crackuje lak za reversing on je sasvim dovoljan da shvatite kada se, zasto se i kako se to prave bruteforceri.

# Bruteforce #2

Kao drugi primer za bruteforceing sam odabrao jedan jako zanimljiv primer koji je napisao Scarabee. Ovaj primer dekriptuje jedan string pomocu XORovanja sa tri razlicita broja. Metu koja se nalazi ovde ..\Cas9\bforceME.exe cemo otvoriti pomocu Ollyja. Kada startujemo program sa F9 videcemo kako izgleda taj string. Ono sto je bitno kod dekriptovanja ovog i drugih XOR enkripcija je da se enkriptovani string mora u potpunosti tacno prepisati sa ekrana! Ali posto jedan od karaktera ocigledno ne pripada standardnom ASCII prikazu, sam enkriptovani string cemo morati da uzmemo iz samog .exe fajla. Stoga cemo potraziti sve referentne stringove u samom fajlu i naci cemo sledeci string:




```
00401416  ASCII "=hrD=<nsrti|qhi|"
00401426  ASCII "ozsr^",0
```

Posto ovo predstavlja samo deo naseg stringa 2x cemo kliknuti na gornji deo stringa i naci cemo se ovde:

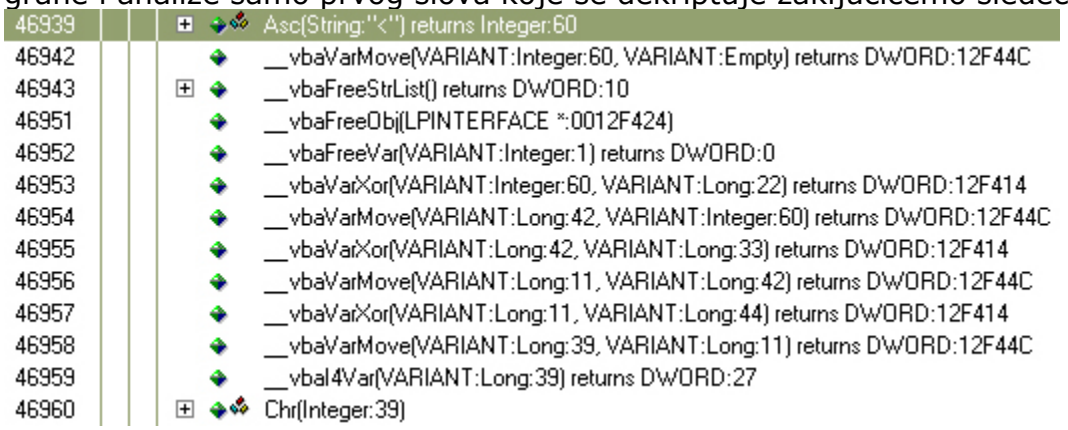
```
004013FC  . 4C 61 62 65 6>ASCII "Label1",0
00401403    01          DB 01
00401404    01          DB 01
00401405    24          DB 24                    ; CHAR '$'
00401406    00          DB 00
00401407    3C          DB 3C                    ; CHAR '<'
00401408    78          DB 78                    ; CHAR 'x'
00401409    79          DB 79                    ; CHAR 'y'
0040140A    72          DB 72                    ; CHAR 'r'
0040140B    7E          DB 7E                    ; CHAR '~'
0040140C    3D          DB 3D                    ; CHAR '='
0040140D    78          DB 78                    ; CHAR 'x'
0040140E    75          DB 75                    ; CHAR 'u'
0040140F    69          DB 69                    ; CHAR 'i'
00401410    3D          DB 3D                    ; CHAR '='
00401411    78          DB 78                    ; CHAR 'x'
00401412    76          DB 76                    ; CHAR 'v'
00401413    72          DB 72                    ; CHAR 'r'
00401414    6F          DB 6F                    ; CHAR 'o'
00401415    7F          DB 7F
00401416  . 3D 68 72 44 3D 3C 6E 73 72 74 69 7C 71 68 69 7C   ASCII "=hrD=<nsrti|qhi|"
00401426  . 6F 7A 73 72 5E 00                    ASCII "ozsr^",0
```

I videcemo kako izgleda i taj karakter koji nam fali, on je 7Fh. Posto smo pronasli sve karaktere sada mozemo da formiramo jedan niz od heksadecimalnih brojeva koji ce nam pomoci da dekriptujemo string:

```
dec[]={0x3C,0x78,0x79,0x72,0x7E,0x3D,0x78,0x75,0x69,0x3D,0x78,0x76,0x72,0x6F,0x7F,0x
3D,0x68,0x72,0x44,0x3D,0x3C,0x6E,0x73,0x72,0x74,0x69,0x7C,0x71,0x68,0x69,0x7C,0x6F,0x
7A,0x73,0x72,0x5E,0x00};
```

Sada nam preostaje da ukljucimo SmartCheck i da pronadjemo nacin na koji se dekriptuje ovaj enkriptovani string. Naravno ovo je moguce uraditi i pomocu Ollyja ali mislim da ce svima biti mnogo jasnije ako analizu ovog algoritma uradimo pomocu SmartChecka.

Posto pokrenete SmartCheck i pomocu njega startujete metu bez unosenja bilo koje vrednosti u metu pritisnite Decrypt dugme. Posle otvaranja [+]Click grane i analize samo prvog slova koje se dekriptuje zakljucicemo sledece:




Da se uzima slovo po slovo iz enkriptovanog stringa i da se dekriptuje pomocu sva tri XOR kljuca po formuli:

**Slovo = "<"**
**ASCII = ASCII(Slovo) = 60**
**ASCII xor KEY1 = 42**
**ASCII = 42**
**ASCII xor KEY2 = 11**
**ASCII = 11**
**ASCII xor KEY3 = 39**

I kao sto se vidi sva tri kljuca se koriste za dekripciju samog stringa. Ali posto XOR funkcija ima divnu osobinu da je (X1 xor X2 xor X3) xor ASCII jednako X4 xor ASCII, to jest da se sva tri kljuca mogu svesti na jedan broj jer ako je X1 = X2, onda je X1 xor X2 = 0, a posto je 0 xor X3 = X3 ovaj slucaj za dekripciju se svodi samo na X3 u opsegu od 0 do 99.

Pre nego sto pocnemo da pisemo algoritam na samom kraju [+]Click grane u SCu cemo primetiti funkciju StrReverse koja se koristi za okretanje dekriptovanog stringa (*ABC* = *CBA*) zbog cega dekripciju treba raditi u suprotnom smeru, od poslednjeg enkriptovanog karaktera do prvog. Bruteforce algoritam bi izgledao ovako:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int
dec[]={0x3C,0x78,0x79,0x72,0x7E,0x3D,0x78,0x75,0x69,0x3D,0x78,0x76,0x72,0x6F,0x7F,0x
3D,0x68,0x72,0x44,0x3D,0x3C,0x6E,0x73,0x72,0x74,0x69,0x7C,0x71,0x68,0x69,0x7C,0x6F,0x
7A,0x73,0x72,0x5E,0x00};
        int i,j;
        for (i=1;i<100;i++)     {
    printf("Combination %i  ",i);
         for(j=35;j>-1;j--){
    printf("%c",dec[j] ^ i);
         }
    printf("\n");
        }
   getchar();
   return 0;
}
```

A posle pokretanja ..\Cas9\Source4-cpp\main.exe videcemo da je resenje ovog XORdinary problema Key1 = Key2 = bilo koji broj od 0 do 99, Key3 = 29.

# Bruteforce the encryption

Ovaj deo poglavlja spaja dve jako bitne celine u jednu. Ovde ce biti detaljno objasnjene tehnike koriscenja bruteforceinga u razbijanju enkripcija. Morate da shvatite da kada je u pitanju reversing slozenih ili relativno slozenih enkripcija nije moguce naci pravi nacin za njihovo dekriptovanje kao sto je to bio slucaj u prethodnom delu poglavlja. Jedini nacin da se pronadje pravi password za enkriptovani fajl je da proverimo sve oblike koje serijski broj moze da ima i da dekriptujemo fajl pomocu njih. Ovo moze biti komplikovano ako pokusavamo da dekriptujemo ceo fajl, da bismo ubrzali ovaj proces mi cemo dekriptovati samo prvih sest bajtova fajla. Ako su svih sest bajtova ispravni onda je velika verovatnoca da ce ceo fajl biti dekriptovan kako treba. Ja sam delimicno olaksao posao dekriptovanja tako sto sam koristio password duzine 6 karaktera, gde su svi karakteri velika slova. Iako se koriste samo velika slova broj kombinacija slova koje mogu predstavljati tacan serijski broj iznosi 244,140,625 serijskih brojeva duzine sest karaktera. Ovaj neverovatan broj postaje dosta veci ako se koriste mala slova i brojevi zajedno sa velikim slovima. Naravno da nalazenje passworda rucno moze da potraje dosta dugo vremena pa se zbog toga pisu specijalizovane aplikacije koje nam olaksavaju posao proveravajuci sve moguce kombinacije slova i brojeva.

Za potrebe ovog poglavlja je napisan poseban algoritam koji enkriptuje odredjeni fajl pomocu nekog passworda i nekog algoritma za enkriptovanje. Primer enkripcije se vidi na slici ispod. Naravno password kojim je enkriptovan fajl encrypted.exe nije AAAAAB nego neki drugi koji mi treba da otkrijemo. Pre samog pisanja decryptera-bruteforcera moracemo prvo da vidimo kako to algoritam za enkripciju radi da bismo mogli na osnovu toga da napisemo odgovarajuci bruteforcer. Ne brinite algoritam nije komplikovan ali je dovoljno tezak da se fajl ne moze dekriptovati bez bruteforcera. Otvorimo program encryptFile.exe pomocu Ollyja. Posto program izbaci string encrypting done!!! kada se zavrsi enkriptovanje mozemo lako da nadjemo CALL koji je zaduzen za enkripciju. Taj CALL pocinje na adresi 00453A64. Kao parametre u program unesite original.exe, new.exe i PSWORD. Postavite break-point na pocetak CALLa za racunanje i pritisnite Encrypt File dugme. Olly je zastao na break-pointu i sada treba da se krecemo kroz kod sa F8 kako bismo videli sta se to desava u ovom CALLu. Sve ovo nam je totalno nebitno dok ne stignemo dovde:


```
00453BD5  |.  BB 01000000     MOV EBX,1
00453BDA  |.  8B07            MOV EAX,DWORD PTR DS:[EDI]
00453BDC  |.  3347 04         XOR EAX,DWORD PTR DS:[EDI+4]
00453BDF  |.  3347 08         XOR EAX,DWORD PTR DS:[EDI+8]
00453BE2  |.  3347 0C         XOR EAX,DWORD PTR DS:[EDI+C]
00453BE5  |.  3347 10         XOR EAX,DWORD PTR DS:[EDI+10]
00453BE8  |.  33C6            XOR EAX,ESI
00453BEA  |.  8BF0            MOV ESI,EAX
```

Posto znamo da se prilikom enkripcije koriste XOR komande ovaj deo koda mora biti vazan za ceo proces enkripcije. Primeticemo da su sadrzaji adresa EDI, EDI+4, EDI+8, EDI+C, EDI + 10, ESI Hex vrednosti ASCII kodova slova iz passworda kojim enkriptujemo fajl. Iz ovoga zakljucujemo da je:

**EAX = Pass[1] xor Pass[2] xor Pass[3] xor Pass[4] xor Pass[5] xor Pass[6]**

gde su Pass[1..6] ASCII vrednosti 1..6 slova iz passworda. Posle ovog xorovanja vrednost iz EAX se prebacuje u ESI. Ispod ovoga sledi petlja iz koje zakljucujemo sledece:

```
00453C01  |.  33C0            |XOR EAX,EAX
00453C03  |.  8A45 FF         |MOV AL,BYTE PTR SS:[EBP-1]
00453C06  |.  33C6            |XOR EAX,ESI
00453C08  |.  33449F FC       |XOR EAX,DWORD PTR DS:[EDI+EBX*4-4]
00453C0C  |.  43              |INC EBX
00453C0D  |.  83FB 06         |CMP EBX,6
00453C10  |. /7E 05           |JLE SHORT encryptF.00453C17
00453C12  |. |BB 01000000     |MOV EBX,1
```

- Na adresi 00453C03 se u AL smesta jedan po jedan karakter iz fajla original.exe (MZ....)
- Na sledecoj adresi se EAX xoruje sa vrednoscu iz ESIja
- Ispod ovoga se EAX xoruje sa ASCII vrednoscu slova prolaza. Primera radi ako je ovo prvi prolaz od sest mogucih (jer toliko ima slova u passwordu) EAX ce biti xorovan sa ASCII vrednoscu prvog slova iz passworda.
- EBX se povecava za jedan i on predstavlja brojac prolaza
- Ako je prolaz veci od 6 prolaz (EBX) ce biti resetovan na 1
- Posle se EAX pretvara u ASCII slovo koje se zapisuje u novi fajl

Ovaj loop ce se ponavljati 4096 puta jer toliko bajtova ima originalni fajl. Kao sto se vidi enkripcija se vrsi na svakih 6 bajtova a onda se postupak ponavlja. To jest prvi bajt se xoruje sa ESIjem i sa ASCII vrednoscu prvog slova, drugi bajt sa drugim, a kada brojac slova predje sest, to jest dodje do sedam onda se on resetuje i krece ponovo od jedan. Tako ce sedmi bajt biti xorovan sa ESIjem i sa ASCII vrednosu prvog karaktera. Na osnovu detalja koje smo saznali mozemo napraviti deo algoritma za bruteforceing. Pre pisanja algoritma moramo da znamo unapred kako cemo ga napisati. Najjednostavniji nacin je da dekriptujemo samo 6 prvih bajtova iz originalnog fajla i da ih uporedimo sa bajtovima iz neenkriptovanog fajla. Zasto uzimamo prvih sest bajtova? Prosto posto se enkripcija ponavlja svakih sest bajtova, prvi i sedmi bajt ce biti enkriptovani na isti nacin.

Kada konstruisemo algoritam prvo moramo da definisemo tablicu slova koja ce biti koriscena da se od njih napravi password. Ova tablica je velika 26 karaktera i sadrzi sva velika slova od A-Z. Posto imamo sest karaktera koji cine password moramo imati i sest petlji od 1 do 26 koje se ponavljaju jedna unutar druge. Zasto ovo radimo? Jer se kombinacije koje prave password krecu od AAAAAA pa do ZZZZZZ menjajuci pri tome zadnje slovo 26 puta, a kada zadnje slovo dodje do 27tog onda se krece iz pocetka i umesto 27og slova koristi se prvo slovo, slovo A, dok se drugo A od kraja povecava za jedan i postaje B. Tada ce kombinacija izgledati ovako AAAABA.

Ono sto mi moramo da proverimo je da li je xorovana vrednost svih slova iz passworda (ESI) xorovana slovom passworda (Pass[i]) koje se

odredjuje pomocu brojaca (I) i xorovana ASCII vrednoscu prvog bajta iz enkriptovanog fajla jednaka 4D hex ili M u ASCII obliku. Ako je ovo ispunjeno provericemo da li je drugi bajt jednak 5A hex ili Z u ASCIIju. Sta sve ovo znaci ??? Ovo znaci da cemo prepisati prvih 6 bajtova iz originalnog fajla (*4D5A90000300*) i da cemo jednacinu postaviti ovako:

**Da li je 10h xor ESI xor ASCII(A) = 4D**
**Da li je 03h xor ESI xor ASCII(A) = 5A**
**Da li je C0h xor ESI xor ASCII(A) = 90**
**Da li je 4Dh xor ESI xor ASCII(A) = 00**
**Da li je 5Fh xor ESI xor ASCII(A) = 03**
**Da li je 5Ah xor ESI xor ASCII(A) = 00**

gde je 10h prvi bajt iz enkriptovanog fajla (prvih 6 bajtova iz enkriptovanog fajla encrypted.exe *1003C04D5F5A*), ESI sadrzaj ESI registra, a ASCII(A) je ASCII vrednost prvog slova iz pretpostavljenog passworda. Pretpostavljeni password za prvi prolaz je AAAAAA, za drugi prolaz je AAAAAB, za treci je AAAAAC, itd... sve dok se ne dodje do ZZZZZZ. Ovo radimo ovako zato sto pokusavamo da dekriptujemo bajtove iz enkriptovanog fajla pomocu xorovanja enkriptovanog bajta sa vrednoscu ESIja i vrednoscu slova prolaza, ako je dobijena vrednost jednaka neenkriptovanoj onda moguce da je pretpostavljeni password tacan. Da bi sve ovo bilo jasnije pogledajte sledecu tabelu:

| Password | ESI | Prolaz | Slovo / ASCII | Enkriptovano | Rez. | Original |
|---|---|---|---|---|---|---|
| ABCDEF | 7 | 1 | A / 65 / 41h | 10h / 16 | 86 | 77 |
| ABCDEF | 7 | 2 | B / 66 / 42h | 03h / 03 | 70 | 90 |
| ABCDEF | 7 | 3 | C / 67 / 43h | C0h / 192 | 132 | 144 |
| ABCDEF | 7 | 4 | D / 68 / 44h | 4Dh / 77 | 14 | 0 |
| ABCDEF | 7 | 5 | E / 69 / 45h | 5Fh / 95 | 29 | 3 |
| ABCDEF | 7 | 6 | F / 70 / 46h | 5Ah / 90 | 27 | 0 |

- Kolona ESI se dobija ASCII(A) xor ASCII(B) xor ASCII(C) xor.... ASCII(F)
- Prolaz je brojac koji se koristi za odabiranje slova iz passworda
- Kolona enkriptovano sadrzi prvih 6 bajtova iz encrypted.exe fajla
- Rezultat je: kolona ESI xor ASCII iz istog reda xor Enkriptovano
  Primer: 7 xor 65 xor 16 = 86
- Posto 86 nije jednako originalnom bajtu 77 (4Dh) ovaj pretpostavljeni password nije tacan pa se ostali prolazi ne moraju ni proveravati. Preci cemo samo na sledeci password ABCDEG i pokusacemo sa njim.

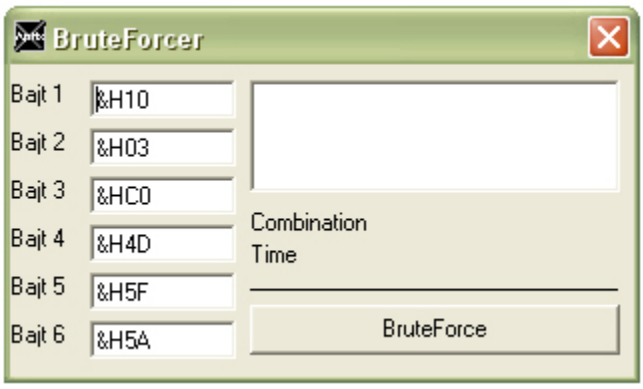
Sada znamo sve sto nam treba kako bismo napisali algoritam u nekom programskom jeziku. Ono sto moramo da napravimo je da napisemo algoritam koji ce generisati sve moguce passworde (od AAAAAA-ZZZZZZ) i proveravati da li je pretpostavljen serijski broj tacan ili ne. Provera se vrsi bas onako kako je opisano u tablici iznad. Ja sam napravio jedan jako dobar primer u VB kako se to moze napisati jedan bruteforcer. Ovaj primer naravno nije optimizovan i posle par sekundi bruteforceovanja program ce prikazivati da je "zakucao" ali nije, on u pozadini radi svoj posao sto brze moze. Dajte mu vremena i on ce zavrsiti svoj posao, kad tad. Ovo moze mnogo da

potraje ali bruteforceovanje je bas takvo, ipak postoji “samo” 244,140,625 kombinacija koje treba proveriti.

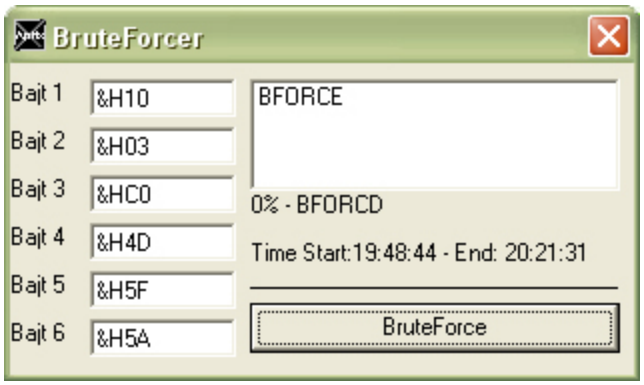
**Algoritam:**

```
Dim table(1 To 26) As String
For i = 65 To 90
 table(i - 64) = Chr(i)
Next i
 Label3.Caption = "Time Start: " & Time$
 Me.Refresh
For i1 = 1 To 26
 For i2 = 1 To 26
  For i3 = 1 To 26
   For i4 = 1 To 26
    For i5 = 1 To 26
     For i6 = 1 To 26
       sm = Asc(table(i1)) Xor Asc(table(i2)) Xor Asc(table(i3)) Xor Asc(table(i4)) Xor
Asc(table(i5)) Xor Asc(table(i6))
      my = (Val(Text1.Text) Xor sm Xor Asc(table(i1)))
     If my = 77 Then
      my = (Val(Text2.Text) Xor sm Xor Asc(table(i2)))
     If my = 90 Then
      my = (Val(Text3.Text) Xor sm Xor Asc(table(i3)))
     If my = 144 Then
      my = (Val(Text4.Text) Xor sm Xor Asc(table(i4)))
     If my = 0 Then
      my = (Val(Text5.Text) Xor sm Xor Asc(table(i5)))
     If my = 3 Then
      my = (Val(Text6.Text) Xor sm Xor Asc(table(i6)))
     If my = 0 Then
       List1.AddItem table(i1) & table(i2) & table(i3) & table(i4) & table(i5) & table(i6)
       GoTo 2
     End If
    End If
   End If
  End If
 End If
End If
  Label2.Caption = table(i1) & table(i2) & table(i3) & table(i4) & table(i5) & table(i6)
  Form1.Caption = "BruteForcer - " & Label2.Caption
 Next i6
Next i5
 Next i4
 Next i3
Next i2
Next i1
2
Form1.Caption = "BruteForcer"
Label3.Caption = Label3.Caption & " - End: " & Time$
```

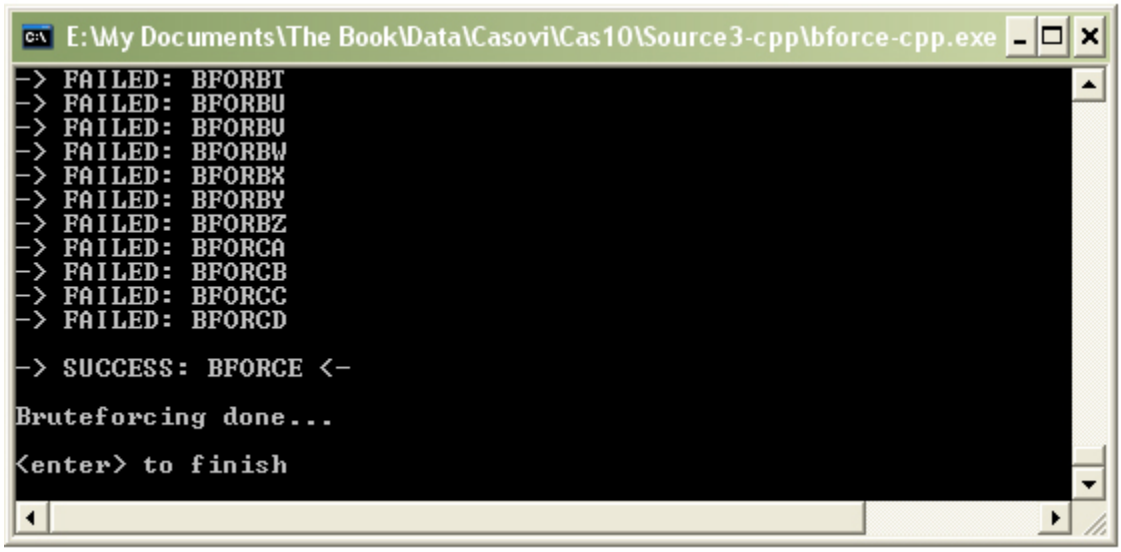
**Objasnjenje:**




Tekstualna polja od 1 do 6 se koriste za unosenje prvih sest bajtova iz enkriptovanog fajla. Ako koristite HEX vrednosti ne zaboravite da ih unosite kao &H pa tek onda HEX vrednost bajta. Pritiskom na dugme BruteForce algoritam odozgo se izvrsava i krece se sa razbijanjem passworda. Source se nalazi u folderu ...\Cas9\Source2\bforce



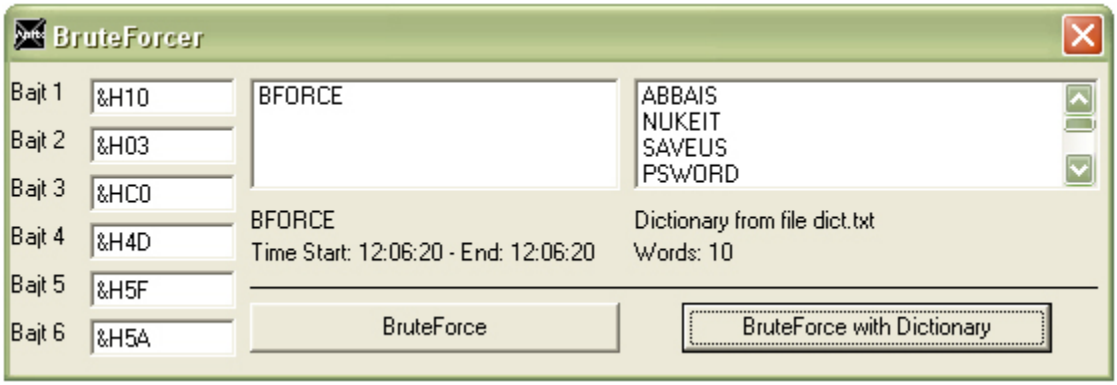

Slobodno startujete algoritam (*bforce-vb.exe*) da biste pronasli pravi password za enkriptovani fajl encrypted.exe... I tako posle 40 minuta na ekranu ce se pojaviti konacan bruteforce rezultat. Password sa kojim je enkriptovan fajl encrypted.exe je BFORCE. Naravno ako fajl original.exe zelite da enkriptujete nekim drugim passwordom mozete to uciniti pomocu programa encryptFile.exe. Da biste bruteforceovali taj novi password morate da ispunite prvih sest polja sa vrednostima prvih sest bajtova iz novog enkriptovanog fajla. Kako biste se resili ovog VBu tipicnog zakucavanja napisacemo isti algoritam u C++, source tog algoritma se nalazi u folderu ...\Casovi\Cas9\Source3-cpp a kompajlovani bruteforcer se zove bforce-cpp.exe. Posle malo kraceg vremena i ovaj C++ bruteforcer pokazuje tacan rezultat:




Konacno kad smo nasli tacan password startujmo encryptFile.exe i u njega redom unesimo encrypted.exe, new.exe, BFORCE ispunivsi sva tri polja sa po jednim parametrom. Kada se zavrsi dekriptovanje mozemo startovati new.exe koji predstavlja dekriptovani encrypted.exe fajl!!!
Ova tehnika se mozda na prvi pogled cini komplikovanom ali je ona u nekim slucajevima jedina koja se moze upotrebiti kada je u pitanju crackovanje nekih passwordom zasticenih fajlova. Naime ova tehnika se mora koristiti kada je zasticeni fajl enkriptovan nekim passwordom. Ove slucajeve srecemo kod arhivera kao sto su WinZIP (od verzije 9.0 koristi AES enkripciju), WinRAR (AES-128 bit), WinACE i tako dalje... Postoje mnogi programi koji vec mogu da vrse bruteforce na sve standardne algoritme kao sto su AES, 3DES, RC6,... ali ako je u pitanju neki specifican algoritam sami cete morati da napisete bruteforcer za njega.

# Bruteforce with dictionary

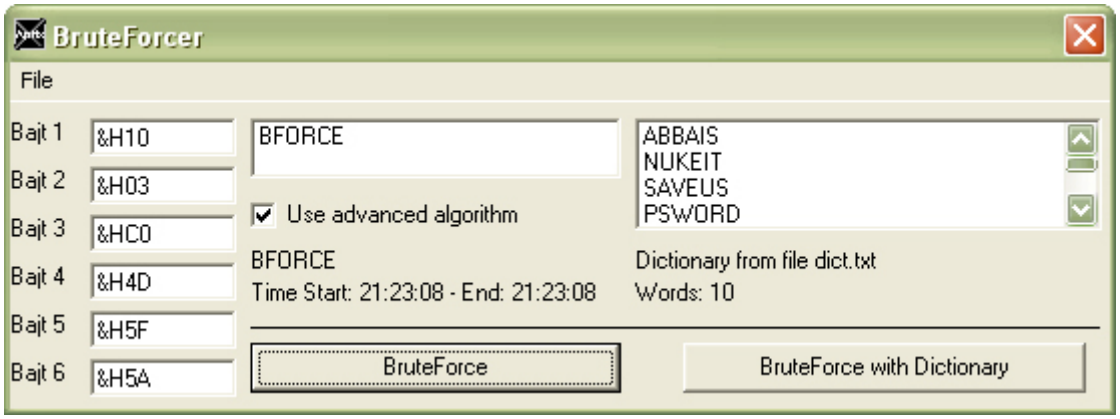
Ovo nije posebna vrsta bruteforceovanja nego je samo jedna njegova grana. Cilj ove tehnike je da se smanji vreme koje je potrebno da se pronadje tacan password. Naravno ova tehnika ima svoje prednosti i mane. Najbitnija prednost je smanjenje vremena koje je potrebno da se pronadje pravi password. Nazalost ova tehnika ima mnogo vise mana nego prednosti. Najveca mana je to sto moramo sami da napravimo recnik koji bi nas program koristio, a takodje je problem to sto mi ne mozemo znati sve moguce kombinacije koje korisnik moze upotrebiti kao password. Mozemo samo da pretpostavimo neke najcesce koriscene passworde i da se nadamo da je korisnik koristio neki od njih. Primer upotrebe recnika u bruteforceovanju mozete naci ovde ...\Cas9\Source2\bforce with dict\




Kao sto vidite iskoristio sam obe opcije za bruteforceovanje i napravio jednu lepu i funkcionalnu celinu. Program ima dve mogucnosti: da vrsi klasicno bruteforceovanje ili da vrsi bruteforceovanje pomocu recnika. Primeticete da je vreme koje je potrebno da se pronadje pravi password svedeno na samo jednu sekundu, ali da je velicina samog recnika smanjena na samo deset mogucih kombinacija. Ovo znaci da brze ne znaci uvek i bolje, klasican nacin bruteforceovanja bi nasao password za veoma dug vremenski period ali bih ga sigurno nasao ako on ispunjava uslove algoritma koji se koristi za dekriptovanje, dok kod recnika to nije slucaj. Brze ili bolje odlucite sami...

# Advanced bruteforceing

Prica o ovom bruteforce algoritmu nije gotova. Naime postoji jos nesto sto mozemo da uradimo kako bismo ubrzali trazenje tacnog passworda. Za ovo nam nece trebati ni velike baze podataka kao recnici, ni selektivno trazenje u raznim opsezima. Jednostavno cemo iskoristiti matematiku samog algoritma tako da napisemo bruteforcer koji ce pronaci tacan password za samo jednu sekundu bez obzira na prvo slovo samog passworda. Ne dajte se plasiti matematikom kao nekim velikim zlom, shvatite to tako da cracking bez osnova matematike ne ide. Naravno za cracking vam nece trebati integrali, ili determinante ali ce vam trebati osnove kombinatorike i sistema verovatnoce. Ovaj deo poglavlja ce detaljno objasniti upotrebu matematike u pravljenu bruteforcera.




Kao sto znamo vec smo napravili alogritam koji ce sigurno naci pravi password za enkriptovani fajl samo sto ce broj mogucih kombinacija biti ogroman, pa ce samim tim i vreme za dekriptovanje biti dugacko. Recimo da zelimo da izbrojimo sve moguce kombinacije koje password moze imati. Za ovo ce nam biti potrebno malo poznavanje kombinatorike.
Dakle uslovi zadatka su da imamo sest slova od A do Z, i da se slova u passwordu mogu ponavljati. Posto ce prva kombinacija izgledati AAAAAA a zadnja ZZZZZZ, zakljucujemo da se na prvom mestu moze nalaziti bilo koje od 25 slova, isto tako i na drugom,... sestom. Znaci broj mogucih kombinacija koje moze imati password je:
25 * 25 * 25 * 25 * 25 * 25 = 25^6 = 244,140,625
Dakle broj mogucih kombinacija iznosi nesto preko 244 miliona. Naravno ovo smo mogli i da izbrojimo samo bi to potrajalo. Naravno mi smo napravili algoritam koji ce probati svako od ovih resenja i ispitati da li je tacno, ali ovo ce potrajati a mi zelimo da smanjimo vreme ispitivanja na sto je manje mogucu meru. Zbog ovoga cemo pokusati nasu srecu sa matematikom i sa algoritmom za enkriptovanje.
Algoritam za bruteforceing smo pisali jer tacan password zavisi od vise parametara. Tacan password zavisi od svakog slova passworda ali i od redosleda slova. Mozda se tokom citanja ovog poglavlja pomislili da iskoristimo istu tehniku kao kod dekriptovanja Trillian passworda, ali onda

ste sigurno zaboravili na registar ESI. Da vas podsetim registar ESI se dobija tako sto se xoruju sve ASCII vrednosti iz pretpostavljenog passworda. Zato nismo mogli da iskoristimo tehniku koju smo upotrebili za Trillian, jer se svaki bajt iz fajla enkriptuje ne samo slovom iz passworda nego i ESI vrednoscu. Naravno sama ESI vrednost se razlikuje za svaki password, ali ne i za svaki redosled slova :) Ovo znaci da ce ESI biti isti za AAABBB i za BBBAAA jer jer xor funkcija reverzibilna to jest A xor A xor A xor B xor B xor B = 3 ali je i B xor B xor B xor A xor A xor A = 3 tako da cemo ovo iskoristiti. Pogledajmo malo sta ta xor funkcija radi.

```
65 xor 65 = 0
0 xor 65 = 65
65 xor 66 = 3
3 xor 66 = 65
65 xor 66 = 3
```

Dakle iz ovoga zakljucujemo da ako imamo sest razlicitih slova bez obzira na to koje je slovo na kom mestu rezultat xorovanja to jest ESI se krece u vrednostima od 0 pa do 91. Zasto? Zato sto ako se na kraju xoruju dva ista slova onda ce rezultat biti minimalan to jest 0, a ako su slova najudaljenija jedna od drugog onda je maksimum 90 xor 1 to jest 91. Ovo je jako zanimljivo jer sada ne moramo racunati ESI jer znamo u kojim se on granicama krece. Naravno ovo moze da se sredi i preko integrala posto znamo granice kojima je odredjen ESI ali necemo to tako. Imamo kompjuter ispred sebe pa cemo ga maksimalno iskoristiti.
Moracemo samo malo da promenimo princip razmisljanja kako bismo ubrzali onaj nas algoritam za bruteforceovanje. Dakle vec smo dosli do formule:
ESI xor ASCII(SLOVO1) xor BAJT1 = 4D
i tako dalje za svako sledece slovo i sledeci bajt. Ono sto cemo mi iskoristiti je cinjenica da ESI moze da ima vrednosti od 0 do 91. Zbog ovoga vise ne moramo da racunamo sam ESI nego cemo ga pretpostaviti u intervalu od 0 do 100. Dakle dekriptovacemo samo prvo slovo po formuli:
ESI xor ASCII(SLOVO1) xor BAJT1 = 4D
gde je ESI od 0 do 100 a SLOVO1 je takodje pretpostavljena vrednost od A do Z. To bi u Visual Basicu izgledalo ovako:

```
For j = 0 To 100
 For i = 65 To 90
   chk = i Xor j Xor tab2(1)
  If chk = tab1(1) Then
     ???????????
  End If
 Next i
Next j
```

Ovim smo definisali kod sve do onih upitnika. Sada treba da napravimo algoritam koji ce dekriptovati i sva ostala slova iz passworda. Imajte na umu da smo vec odredili ESI kao promenljivu **j** i da se ona u predelu upitnika nece promeniti. Ostatak algoritma je jednostavan samo treba da napravite jos jednu petlju koja ide od A do Z i da proverite da li je j xor SLOVOx (x je broj od 2 do 6) xor BAJTx jednako originalnom BAJTUx. Ovaj primer sam ja vec iskodirao i nalazi se u folderu ...\Cas9\Source2\bforce with dict - optimized\ Mozete slobodno analizirati ovaj kod da biste stekli kompletnu sliku o ovom naprednom pisanju algoritma.

# 10 Unpacking...

Ovo veoma bitno poglavlje reversnog inzenjeringa se bavi razbijanjem osnovnih zastita koje mozete naci na netu. Vecina ovih zastita sluzi za sprecavanje modifikacije koda dok manji deo ovih programa sluzi za smanjenje velicine kompajlovanih exe i dll fajlova.

# Unpacking anything...

Sta su to pakeri? Pakeri su specificni programi koji se koriste u cilju zastite ili smanjenja velicine originalnih exe i dll fajlova. Ovakvi fajlovi se ne mogu direktno debugovati i / ili modifikovati. Ono sto se mora prvo uraditi da bi se moglo pristupiti klasicnom reversingu je otpakivanje. Postoji veliki broj komercijalnih, ali i besplatnih zastita i pakera tako da se nas posao dodatno komplikuje jer je potrebno znati otpakovati sve ove vrste zastita. Naravno to nije moguce uciniti jer se otpakivanje programa zasticenih ovom vrstom packera razlikuje od verzije do verzije pakera pa ce u ovoj knjizi biti opisani samo nacini otpakivanja najpopularnijih zastita koje se mogu naci na internetu.

Otpakivanje svih vrsta zastita i pakera je u osnovi veoma lako i svodi se na samo tri bitna koraka:

1) **Ubijanje Anti-Debugging algoritama**
2) **Pronalazenje OEPa i dumpovanje**
3) **Popravka importa i optimizacija fajla**

Analiziracemo svaki od ova tri koraka:

1) Prvi korak je vec detaljno objasnjen u poglavlju "Getting caught" i nema nikakve potrebe za dodatnim objasnjenjima u vezi detektovanja aktivnog debbugera.
2) Sta je to OEP? OEP je cesto koriscena skracenica za Original Entry Point sto je u prevodu prva linija koda koja ce se izvrsiti nakon ucitavanja fajla u memoriju. Sada shvatate zasto nam je ovo bitno. Ako je program pakovan i / ili zasticen nekom tehnikom onda ce se sve sem koda packera nalaziti kompresovano i / ili enkriptovano u fajlu. Nas cilj je da pustimo sam algoritam da se otpakuje u memoriju tako da ne krene sa izvrsavanjem zapakovanog programa, to jest da se linija koda koja se nalazi na OEPu ne izvrsi. Ovaj podatak nam je bitan jer mi pokusavamo da vratimo fajl u ono stanje kakvo je bilo pre pakovanja nekim packerom.
3) Sta su to importi? Importi su tablice sa tacnim adresama do svih dll fajlova koje koristi neki program. Pri mapiranju ovih dll fajlova takodje se mapiraju i sve API funkcije i ostale export funkcije koje se nalaze u njima i koje program moze da poziva prilikom svog izvrsavanja.

Sada kada znamo sta su nam prioriteti prilikom svakog otpakivanja zasticenog fajla mozemo se posvetiti pojedinacnim packerima.
Za identifikovanje verzije i vrste pakera koristicemo program pod imenom PeID sa kojim ste se do sada, ako ste pazljivo citali ovu knjigu, sigurno morali sresti i upoznati.
Pre nego sto pocnemo sa otpakivanjem fajlova upoznacemo se sa strukturom i osnovim pojmovima PE fajlova.

# PE Basics

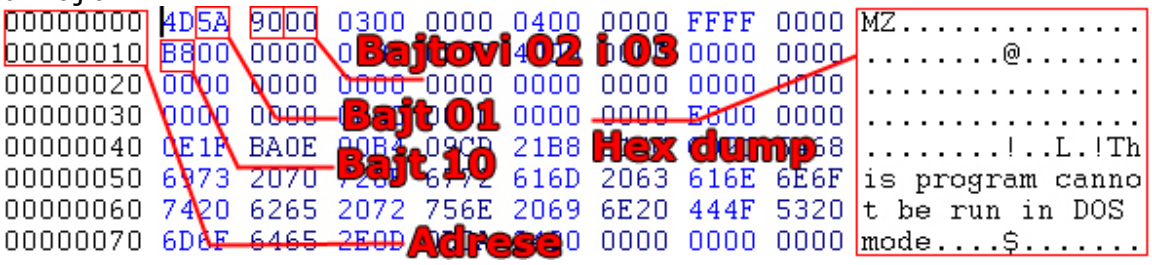
PE je skracenica za Portabile Executable i predstavlja standard po kojem kompajleri raznih programskih jezika kreiraju exe fajlove. Kao sto sigurno znate exe fajlovi predstavljaju samo srce Windows operativnog sistema i zasluzni su za rad svih programa koje koristite u radu. Ali PE format se ne primenjuje samo na exe fajlovima, PE se primenjuje u specijalnom obliku i na dll fajlove koji predstavljaju dinamicke biblioteke, o cemu ce reci biti kasnije. Posto je naveci deo reverserskih problema vezan direktno za same exe fajlove,analizu PE formata cemo zapoceti bas sa ovim standardom.

# PE ExE files - Intro

Svaki klasican PE fajl pocinje uvek sa dva ista bajta, sa MZ ili sa 4D5A heksadecimalno. Ovako bi izgledao neki PE exe fajl otvoren pomocu nekog

```
00000000 4D5A 9000 0300 0000 0400 0000 FFFF 0000 MZ..............
00000010 B800 0000 0000 0000 4000 0000 0000 0000 ........@.......
00000020 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000030 0000 0000 0000 0000 0000 0000 E800 0000 ................
00000040 0E1F BA0E 00B4 09CD 21B8 014C CD21 5468 ........!..L.!Th
00000050 6973 2070 726F 6772 616D 2063 616E 6E6F is program canno
00000060 7420 6265 2072 756E 2069 6E20 444F 5320 t be run in DOS
00000070 6D6F 6465 2E0D 0D0A 2400 0000 0000 0000 mode....$.......
```

hex editora. Kao sto se vidi sa slike plavo su obelezeni parovi bajtova, dok se crno sa strane nalazi tumacenje tih istih parova bajtova samo u ASCII obliku. Svi Hex editori rade automatsko prevodjenje svih heksadecimalnih brojeva u ASCII, naravno pod uslovom da postoji ASCII slovo sa kodom koji koresponduje heksadecimalnoj vrednosti. Osnovna ASCII tabela se sastoji od 127 karaktera ili 7F ako broj karaktera predstavljamo heksadecimalno. Posto je svakom crackeru preko potrebna ASCII tabela predlazem da i vi preuzmete vasu sa sajta http://www.asciitable.com. Analizu PE formata pocecemo analizom bilo kojeg exe fajla iz Hex editora po vasem izboru. Primeticete sledece karakteristicne stvari na samom pocetku svakog exe ili dll fajla.




Ova analiza se ne razlikuje od analize bilo kog klasicnog fajla otvorenog pomocu Hex editora. Ovaj prvi deo poglavlja o PEu ce vas samo uvesti u nacin razmisljanja o fajlovima kao nizovima bajtova.
Adrese, kao sto je oznaceno na slici iznad, predstavljaju lokacije na kojima se nalaze bajtovi. Posmatrajte ovo kao da su u pitanju redovi u bilo kom tekst fajlu, gde svaki red sadrzi samo po dva slova koja predstavljaju jedan bajt. Adrese predstavljaju samo broj reda u kojem se nalazi bajt koji posmatramo.

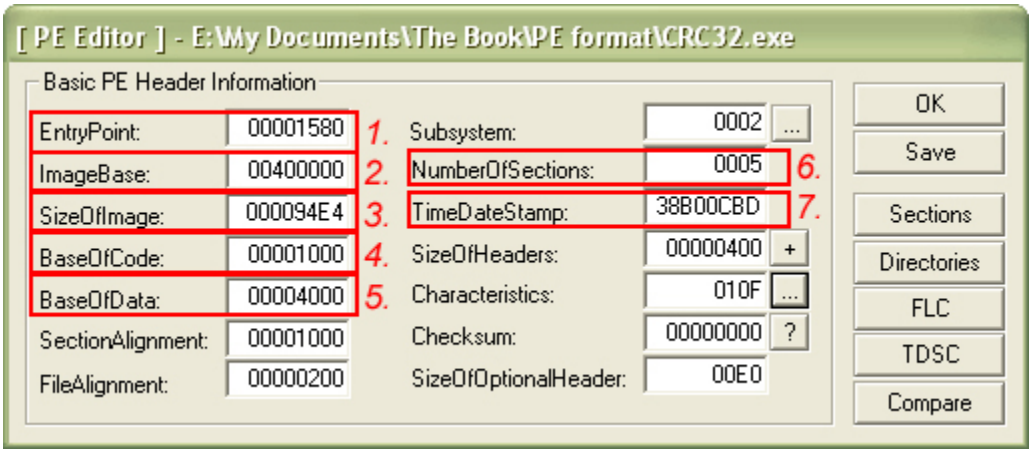
Ovo je laicki pristup ali je tako najlakse da shvatite zasto se adresa na kojoj se nalazi prvi 90h bajt oznacava sa 3h. Primeticete da sam posle brojeva pisao slovo h. Ovo slovo oznacava da se ovi brojevi racunaju kao heksadecimalni a ne decimalni brojevi. Posto se heksadecimalni brojevi znatno razlikuju od decimalnih potrebno ih je pretvarati, a najbolji program za to je obican Windowsov kalkulator.


Kao sto se vidi na slici potrebno je samo selektovati dugme Hex umesto dugmeta Dec i uneti zeljeni heksadecimalni broj. Probajte sa brojem 90h, posle cega samo treba ponovo izabrati dugme Dec posle cega broj 90h postaje 144. Razumevanje heksadecimalnih brojeva je kljucno za reversere jer se svi problemi zasnivaju na heksadecimalnim brojevima. Vec sam objasnio kako se oznacavaju adrese, ali ono sto vam sigurno nije jasno je sledece: Ako svaki par brojeva (4D,5A,90) predstavlja po jedan bajt i posto se taj jedan bajt nalazi na samo jednoj hex adresi, zasto se onda prva adresa u drugom redu oznacava sa 10h? Odgovor je jednostavan: Posto se adrese sa porastom broja bajtova povecavaju za jedan a u jedan red stane samo 16 bajtova, logicno je da ce sledeci red pocinjati sa 10h ili 16 decimalno. Naravno fajl se na disku ne snima tako da postoje redovi, odnosno svi bajtovi se nalaze u istom jednom redu i predstavljaju sukcesivni niz brojeva, ali hex editor radi lakseg prikazivanja fajla prikazuje samo po 16 bajtova u jednom redu. Ovako nam je lakse da znamo na kojoj se to adresi nalazimo i koji tacno bajt citamo ili modifikujemo. Imajte na umu da su ovo stvarne lokacije bajtova u fajlu, a ove adrese korespondiraju fizickim lokacijama. Ovo je jako bitno da znate jer se u fajlu prilikom modifikacije menjaju stvarne, fizicke hex adrese a ne virtualne sa kojima cemo se kasnije upoznati. Na kraju na gornjoj slici je crveno uokviren deo koji je oznacen kao Hex dump. On samo predstavlja ASCII tumacenje bajtova sa leve strane.

# PE ExE files - Basics

Posto smo objasnili osnovne stvari vezane za analizu bilo kojeg fajla pomocu hex editora vreme je da se posvetimo ozbiljnijoj tematici. Da biste pratili ono sto ce biti objasnjeno ovde potreban vam je program LordPE i meta koja se nalazi zajedno sa ovom knjigom, crc32.exe.

Pokrenite LordPE i u njemu izaberite opciju Pe Editor posle cega cete u program ucitati gore pomenutu metu. Ono sto vidite prikazano je na sledecoj slici.




Kao sto se vidi na slici ja sam vec obelezio najbitnije podatke za reversere koji se mogu videti posle otvaranja nase mete. Pocecemo sa detaljnom analizom obelezenih delova programa.

*EntryPoint* je prva a ujedno i jako vazna opcija za reversere. Da biste razumeli ono sto ova opcija predstavlja objasnicu to na laksi nacin. Svaki program bez obzira u kom programskom jeziku je napisan mora da ima svoj pocetak. Taj pocetak predstavlja prvu liniju koda koja ce se izvrsiti odmah nakon startovanja programa. Ova prva linija se naziva EntryPoint ili kako se to cesce srece u reverserskoj literaturi OEP. Broj koji se nalazi upisan u ovom polju predstavlja virtualnu adresu na kojoj se nalazi prva linija koda. Ova virtualna adresa ne predstavlja stvarnu fizicku lokaciju na kojoj se nalazi prvi bajt koji ce se izvrsiti, ali razliku izmedju stvarnih i virtualnih adresa ostavljamo za posle.

*ImageBase* je druga opcija u nizu i takodje je izuzetno bitna. Sigurno ste navikli da OEP adresa izgleda npr. ovako 00401000 ili kako bi to bilo u ovom slucaju 00401580. Vec smo zakljucili da je virtualna adresa OEPa 00001580, ali zasto se ova adresa i adresa koju dobijamo za OEP iz debuggera ili disassemblera razlikuju? Odgovor je sledeci: Pravi OEP se nalazi na adresi 00001580 ali posto se na ovaj broj dodaje ImageBase broj on sada iznosi 00401580. Svrha koriscenja ImageBasea se sastoji u tome sto je potrebno da znamo da li se nalazimo u samom kodu exe fajla ili u kodu nekog dll-a. Ovo je jako bitna informacija i imajte je na umu prilikom menjanja OEPa pomocu PE editora.

*SizeOfImage* je treca opcija u nizu i nije od izuzetne vaznosti za reversing jer predstavlja samo zbir Virtualnog offseta poslednje sekcije PE fajla i njegove Virtualne velicine. O sekcijama ce biti reci posle.

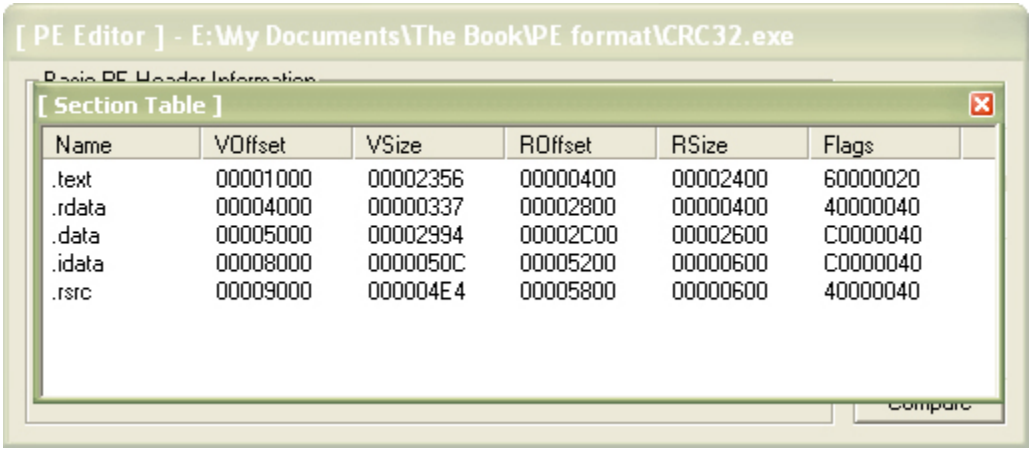
*BaseOfCode* je cetvrta opcija u nizu i predstavlja Virtualni offset glavne code sekcije koja sadrzi OEP i kod koji se izvrsava. Ovo je glavna sekcija na kojoj se u 99% slucajeva izvrsava reversing.

*BaseOfData* je peta opcija u nizu i predstavlja Virtualni offset sekcije koja sadrzi podatke o memoriji programa. Ovo je izuzetno vazno.

*NumberOfSections* je sesta opcija u nizu i predstavlja broj sekcija koje se nalaze u PE fajlu.

*TimeDateStamp* je sedma opcija u nizu i nije bitna iz razloga sto vecina programa proverava datum svoje kreacije i modifikacije na osnovu file atributa a ne preko time stampa pa je ova opcija zanemarljiva.

Siguran sam da vam vecina stvari izgleda jako komplikovano i cudno ali to nije razlog za brigu. Vecinu tih stvari cu objasniti odmah sad ali cete za neke morati par puta da se vratite na ovaj deo i da povezete stvari. Objasnjavanje PE strukture nije jednostavno tako da je izuzetno tesko odabrati pravi pocetak, ali ja sam se odlucio da prvo pomenem skoro sve pojmove vezane za PE header a tek onda da ih objasnim. Da biste me pratili pritisnite dugme Sections u LordPEu.




*Sekcije* - Ono sto ce se pojaviti na ekranu vidi se na slici iznad. Ovaj spisak mozda izgleda konfuzno, ali to nije. Sekcije predstavljaju delove samog PE exe/dll fajla, organizovane tako da bi operativni sistem lakse baratao podacima / komandama koje sam exe fajl zadaje operativnom sistemu. Organizacija se vrsi prema tipu podataka i tako postoje posebne sekcije za kod programa, za resurse (dijalozi,slike,zvuk,...), za reference ka funkcijama koje se nalaze u externim dll fajlovima i slicno.
Obratite paznju na prvu kolonu koja sadrzi imena sekcija. Primeticete da imena svih sekcija pocinju tackom. Ovo je definisano standardom kompajlera koji je napravio exe fajl ali ime sekcije ne mora pocinjati tackom. Ono sto je

bitnije je samo ime sekcije. Iako je ime sekcije proizvoljno, ako se radi o programima koje su napravili standardni kompajleri (c++,vb,delphi,...) imena nam mogu pruziti dosta podataka o samoj sekciji. Na primer prva sekcija se zove .text ali ona ne sadrzi samo text nego je glavna sekcija u fajlu i sadrzi izvrsni kod exe fajla. Ovo znaci da se OEP uvek nalazi u izvrsnoj sekciji sto je u ovom slucaju .text. Najcesca dva imena izvrsne sekcije su .text i .code u zavisnosti od kompajlera do kompajlera. Od ostalih sekcija bitna nam je sekcija .data koja sadrzi dodatni izvrsni kod, sekcija .idata koja sadrzi reference ka funkcijama koje se nalaze u .dll fajlovima i .rsrc koja sadrzi resurse. Imena ovih preostalih sekcija su vise-manje standardna.
Primeticete da je broj sekcija jednak pet sto se slaze sa podatkom koji se nalazio u proslom prozoru i koji je definisan poljem NumberOfSections. Prilikom modifikovanja PE sekcija treba biti ekstremno oprezan jer ako pogresimo u modifikaciji programa, on ce odbiti da se startuje. Prilikom reversinga imacete priliku da veliki broj puta morate da modifikujete PE headere tako da cete brzo nauciti kako se sta modifikuje.

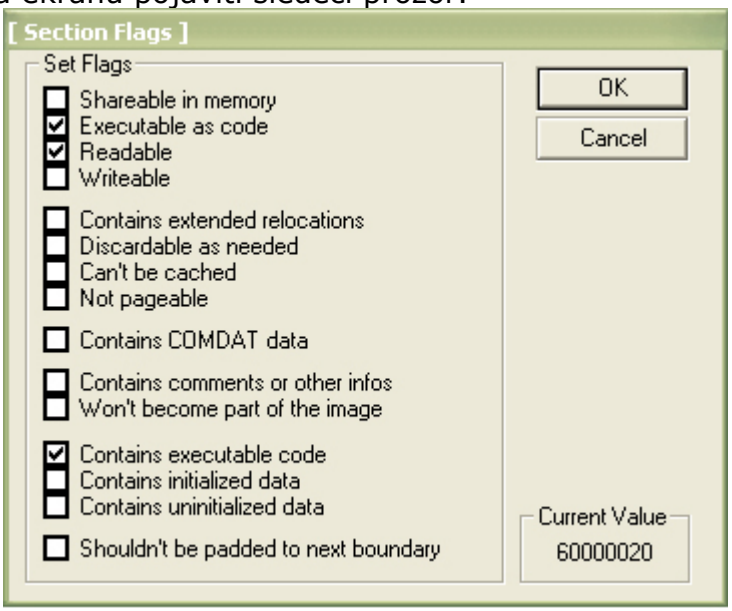
*Virtual Offset* - Ovo je druga kolona i izuzetno je bitna za razumevanje virtualnih adresa i njihovo pretvaranje u prave fizicke. Ova kolona sadrzi heksadecimalne brojeve koji predstavljaju prvu adresu koja se nalazi u sekciji. Ovo znaci da ako je u pitanju prva sekcija, izvrsna, adrese u njoj pocinju od 00001000. Mozda ce vas ovo malo zbuniti jer kako znamo OEP je 00001580 a prva adresa u sekciji je 00001000. Ovo samo znaci da OEP moze biti na bilo kojem mestu u sekciji, to jest da ne mora biti na samom pocetku sekcije nego recimo moze biti na kraju. Zasto su nam potrebni virtualni offseti? Prilikom reversinga je potrebno znati u kojoj sekciji se nalazimo kako bismo znali sta i kako da modifikujemo. Bitno je da ovi brojevi moraju sukcesivno rasti a da ovaj porast mora biti jednak ili veci od Virtual Offset + Virtual size vrednosti. Najcesce je ovaj broj zaokruzen kako bi se ostavila mogucnost za ubacivanje koda u prazan prostor izmedju sekcija.

*Virtual Size* - Ovo je treca kolona i predstavlja velicinu sekcije. Ova velicina mora biti priblizna ili ista kada se radi o virtualnom i raw (stvarnom) offsetu. Ono na sta treba obratiti paznju prilikom modifikovanja velicine sekcije je to da velicina izvrsne sekcije mora uvek biti manja, bar za jedan bajt, od stvarne fizicke velicine sekcije. Ako je velicina sekcije veca od unetog broja u polje Virtual size onda se program nece startovati i izbacice poruku da PE fajl nije validan. Ako je u pitanju neki drugi tip sekcije, velicina nije toliko bitna.

*Raw Offset* - Ovo je cetvrta kolona i predstavlja stvarnu fizicku lokaciju sekcije. Ova fizicka lokacija predstavlja mesto prvog bajta selektovane sekcije u fajlu pocevsi od prvog bajta.

*Raw Size* - Ovo je peta kolona i predstavlja stvarnu fizicku velicinu sekcije. Velicina izvrsne sekcije mora biti veca, bar za jedan bajt, od virtualne inace se program nece startovati. Ako je u pitanju neki drugi tip sekcije, velicina najcesce nije bitna.

*Flags* - Ovo je poslednja kolona i predstavlja atribute same sekcije. Kao sto vidite njen sadrzaj je takodje heksadecimalni broj ali u zavisnosti od kombinacije ovih brojeva sekcija ima razlicite osobine. Da biste editovali sekciju selektujte sekciju i kliknite desnim dugmetom na nju. Posle ovoga selektujte Edit Section Header, posle cega pritisnite dugme tri tackice. Posle ovoga ce se na ekranu pojaviti sledeci prozor.



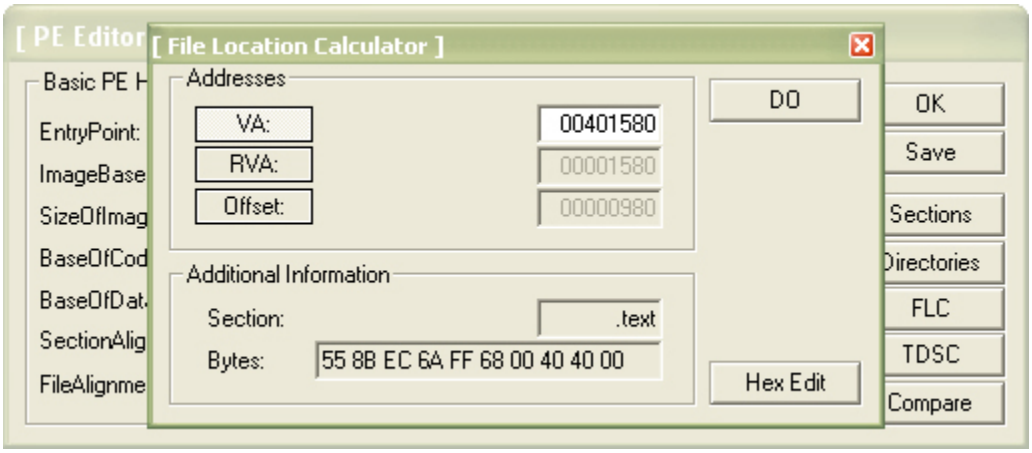

Kao sto se vidi ja sam selektovao glavnu .text sekciju i ovo su njeni atributi. Posto je ovo glavna sekcija i sadrzi kod koji se izvrsava. Ovo se i vidi iz atributa same sekcije. Ono sto je bitno kod postavljanja flagova je da ako sekcija koju zelite da modifikujete direktno pomocu ubacenog ASM koda mora imati atribute readable i writeable. Ako zaboravite na ovo onda ce se program srusiti prilikom pokusaja modifikovanja memorije sekcije iz samog programa *(videti ASM deo 4)*.Kada su u pitanju sekcije koje sadrze kod najbolje da karakteristike odmah postavite na E0000020 da biste tako resili problem pisanja u memoriju sekcije. Postavljanje pravih flagova je vazno samo ako planirate da dodajete sekcije, dok su kod vec postojecih flagovi namesteni kako treba. Ovo je opcija za napredne crackere koji ce iz samog exe fajla modifikovati kod glavne sekcije pa stoga moraju da izmene njene atribute.

*Virtualne adrese* - U daljoj reverserskoj praksi cesto cete se sretati sa potrebom da pretvarate virtualne adrese u fizicke kako biste uspesno patchovali vase mete. Da biste ovo razumeli otvorite nasu metu pomocu Ollyja. Ono sto cete videti u glavnom CPU prozoru je klasican VC++ OEP:

```
00401580 > $  55              PUSH EBP
00401581   .  8BEC            MOV EBP,ESP
00401583   .  6A FF           PUSH -1
00401585   .  68 00404000     PUSH CRC32.00404000
```

Kao sto vidimo OEP se nalazi na adresi 00401580, ali ovo smo vec znali jer se u polju EntryPoint u PE editoru nalazi 00001580 a u polju ImageBase se nalazi 00400000. Posto se nalazimo u prvoj sekciji prva adresa same sekcije je 00001000 dok se sve ostale sukcesivno nastavljaju. Naravno da bismo

dobili tacnu virtualnu adresu na broj 00001580 moramo dodati ImageBase posle cega cemo dobiti tacnu adresu naseg OEPa, to jest 00401580.
Dalje je jako bitno da razumete zasto, i kako se adrese povecavaju. Primeticete da se OEP nalazi na adresi 00401580 a da se sledeca komanda nalazi na adresi 00401581. Iz ovoga zakljucujemo da se adrese povecavaju za jedan po redu. Ali da li je ovo bas tako? Ako pogledamo sledecu komandu videcemo da sledeca adresa nije 00401582 kao sto smo ocekivali nego je 00401583. Zasto je ovo bas ovako? Posto prva kolona oznacava adrese na kojima se nalaze komande, a druga kolona same komande u hex obliku odgovor je jednostavan.Posto dvocifreni heksadecimalni brojevi predstavljaju po jednu komandu ovo znaci da se na jednoj adresi nalazi samo jedna komanda. Primera radi, na adresi 00401580 se nalazi samo komanda 55, na sledecoj 00401581 se nalazi 8B, na adresi 00401582 se nalazi EC a na 00401583 komanda 6A. Kao sto se vidi svaka komanda ima jedinstvenu adresu. Prva kolona u gornjem prikazu samo oznacava prvu adresu iz niza bajtova koji cine jednu komandu. Ovo znaci da jednu komandu moze ciniti vise dvocifrenih heksadecimalnih brojeva. Na primer: 68 00404000 oznacava jednu ASM komandu, a ona je PUSH 00404000.
Sada shvatate da se javljaju problemi prilikom modifikovanja PE fajlova sa hex editorima. Naime da bismo patchovali stvarnu fizicku lokaciju u fajlu potrebno je pretvoriti virtualnu adresu koju zelimo da izmenimo u stvarnu fizicku adresu i tek onda treba izvrsiti modifikaciju. Za ovo sluzi FLC opcija u LordPEu.

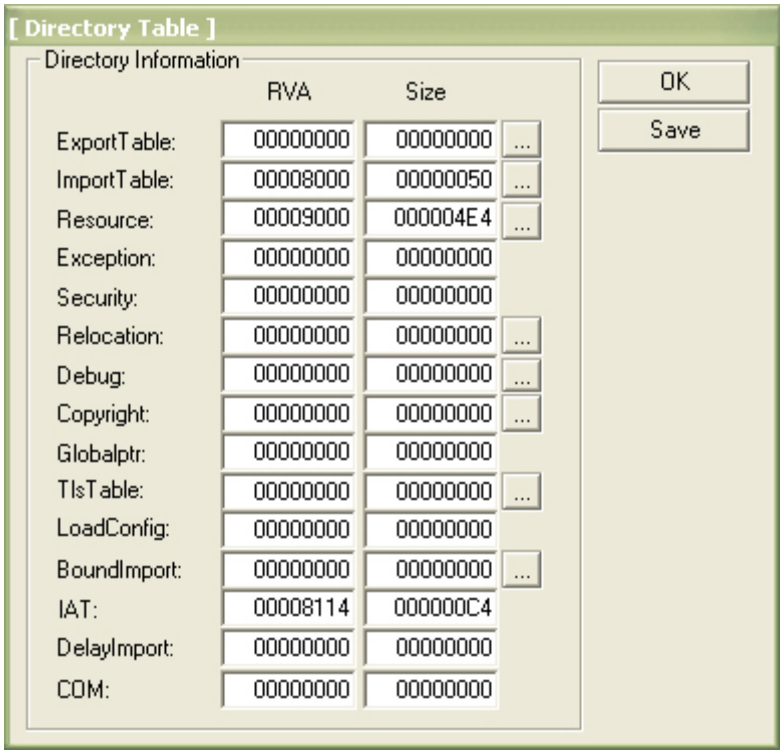


Kao sto se vidi sa slike moguca je konverzija u sva tri pravca. Moguce je uneti virtualnu adresu, relativnu virtualnu adresu (adresa bez image base) i konacno file offset koji predstavlja stvarnu lokaciju bajta u fajlu. Da bismo izvrsili pretvaranje ovih velicina potrebno je da izaberemo metod pretvaranja, unesemo adresu i pritisnemo dugme DO. Posle ovoga imacemo sve potrebne podatke o lokaciji bajta u fajlu.

# PE ExE files - Tables

Posto smo objasnili osnovne stvari vezane za PE strukturu vreme je da analiziramo import/export tabele. Pre nego sto pocnemo moramo da razjasnimo sta su to import tabele.

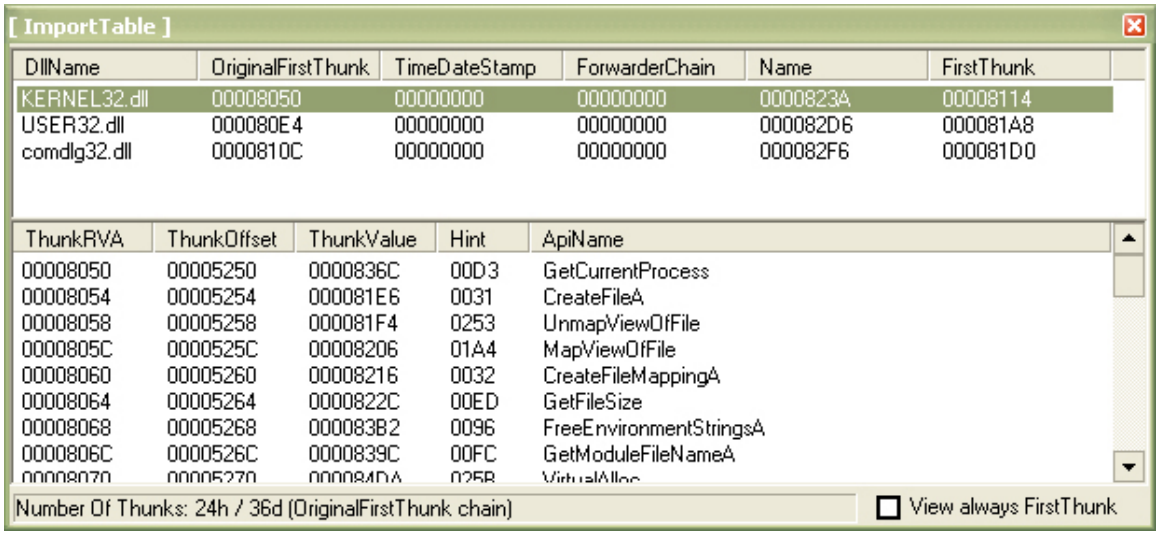
Ako ste se do sada bavili programiranjem u bilo kom programskom jeziku sigurno ste culi za API. Ako niste, API predstavlja spisak funkcija koje se nalaze u standardnim Windows dll fajlovima kao sto su kernel32.dll, user32.dll i slicni. Svaki exe fajl sadrzi import tabelu koja predstavlja referencu ka funkcijama koje se nalaze u ovim windows dll fajlovima ali i drugim specificnim dll fajlovima koji se mogu isporucivati uz program. Da biste videli koje tabele postoje u jednom exe fajlu u LordPEu izaberite opciju Directories sto ce vam prikazati sledece:




Kao sto se vidi sa slike svaki PE fajl moze imati vise tablica. Od svih ovih opcija za reversing su nam interesantna samo cetiri polja, polje ExportTable, ImportTable, IAT i Resource.

*ExportTable* - predstavlja tablicu funkcija koju jedan dll fajl "izvozi". Ovo znaci da standardni exe fajlovi mogu pozivati .dll fajl, to jest specificne funkcije u njemu koje ce odraditi neki posao umesto samog exe fajla. Ovo znaci da ako analiziramo neki windowsov dll fajl funkcije koje se nalaze u njemu ce biti smestene u ExportTable, dok ce se u .exe fajlu koji ih poziva biti smestene u ImportTable.

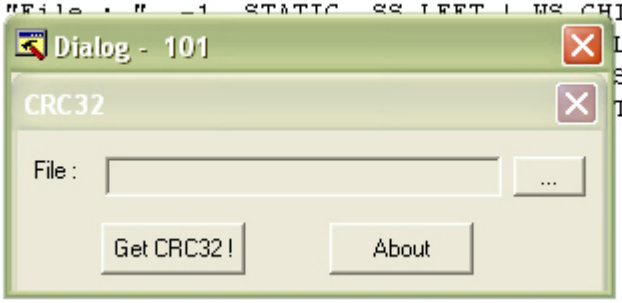
*ImportTable* - predstavlja tablicu funkcija koju jedan .exe fajl poziva iz jednog ili vise .dll fajlova. Da bi se .exe fajl startovao potrebno je da postoje svi .dll fajlovi koji se nalaze u import listi .exe fajla. Importi se mogu citati iz prakticno neogranicenog broja .dll fajlova.
*IAT* - predstavlja tablicu funkcija koja sluzi za mapiranje svih API funkcija koje se nalaze u jednom PE fajlu.
Import/export tablice se mogu pregledati direktno iz LordPEa, samo treba da pritisnemo dugme tri tackice i videcemo sledeci prozor.




Kao sto se sa slike vidi API funkcije su podeljene po dll fajlovima. Ovaj pregled nam je jako koristan jer u slucaju poziva funkcija koje smo rucno dodali u PE fajl mozemo lako procitati RVA adresu. Takodje se iz ovog prozora mogu videti sve API funkcije, sto nam je jako korisno jer mozemo saznati mnogo podataka o samoj meti na ovaj nacin. Ista vrsta prozora se koristi i za Export tabele tako da nema potrebe ovo ponovo objasnjavati. Ako zelite da vidite kako izgleda jedna Export tablica otvorite bilo koji .dll fajl.

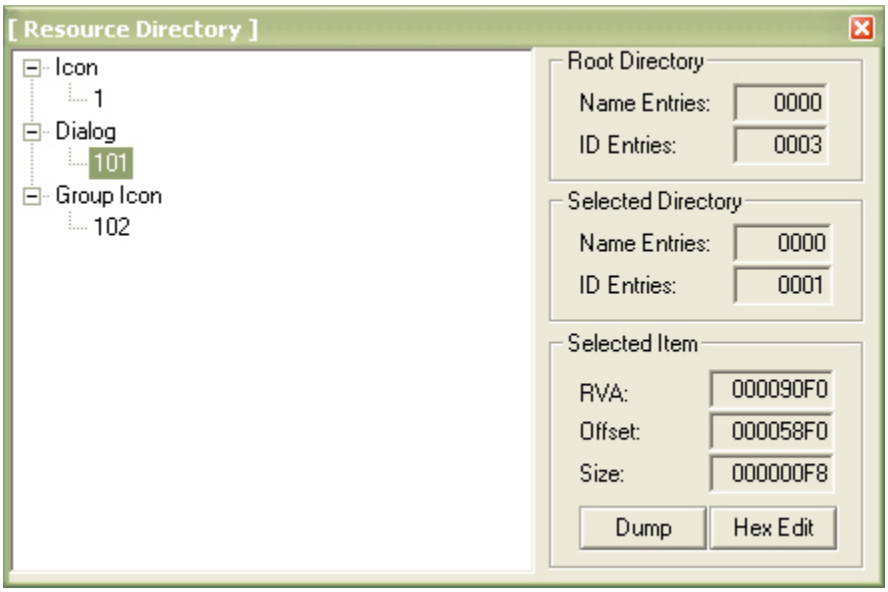
Cetvrta tablica koju smo pomenuli je Resource tablica. Ona se koristi za cuvanje specificnih podataka u jednom PE fajlu. Ovi podaci mogu biti multimedijalnog karaktera (slika,tekst,zvuk...) ili programski podaci kao sto su ikone, dijalozi i slicni. Primer jednog klasicnog dijaloga se nalazi na slici 01.09. Ovi dijalozi predstavljaju standardan Windows interface, ili GUI.

```
{
    CONTROL "File : ",  -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIB
    CONTROL                                    LL | WS_CHILD
    CONTROL                                   S_CHILD | WS_V
    CONTROL                                   TTON | WS_CHIL
    CONTROL                                    WS_CHILD | WS
}
```




Ako vam je potreban ovako detaljan prikaz upotrebite program ResHacker. Za osnovne potrebe pregleda IDa svih resursa iz PE fajla dovoljan je LordPE.

Kao kada su u pitanju import/export tablice potrebno je pritisnuti samo dugme tri tacke pored polja Resources u Directories prozoru. Posle ovoga ce se pojaviti sledeci prozor.

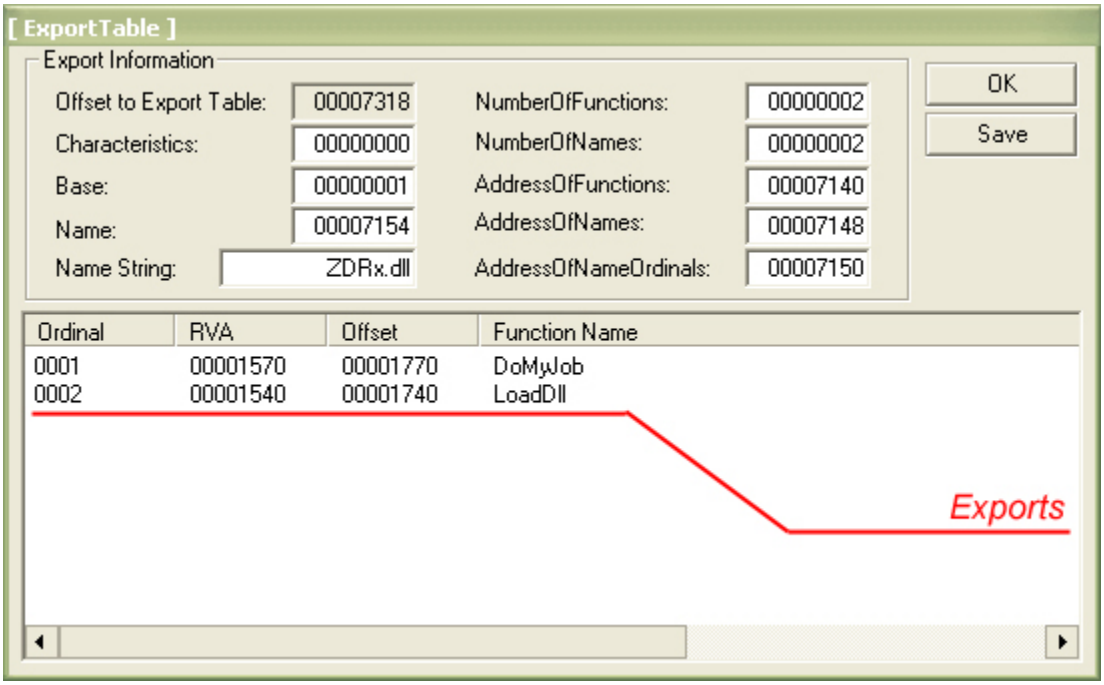



Kao sto se vidi na slici resursi su poslagani po tipu podataka. Svaki od resursa se ima svoj jedinstveni ID iz razloga sto kada se preko ASM koda pristupa pojedinacnim resursima potrebno je znati tacno na koji resurs se data ASM komanda odnosi, zbog ovoga svi resursi u PE fajlu dobijaju jedinstvene IDove.
Editovanje samih resursa moze se vrsiti na razne nacine. U zavisnosti od onoga sto zelite da promenite mozete koristiti ili Hex editor ili neki od specijalnih resurs editora kao sto je Resource Hacker.

Ovo su najbitnije osobine svakog PE fajla. Reversing se zasniva na detaljnom poznavanju svih osnivnih osobina PE fajlova i detaljnom poznavanju sekcija i import/export/resource tablica.

# PE DLL files - Exports

Kao sto smo vec rekli postoji znacajna razlika izmedju PE exe i PE dll fajlova. Iako se razlika ogleda u mnogo osnovnih PE detalja kao sto su ImageBase i karakteristikama samog PE fajla, najbitnija razlika izmedju PE exe i dll fajlova je u postojanju Export tabele u PE dll fajlu. Kao sto smo vec rekli iz LordPEove opcije Directory moze se videti kako izgleda Export tabela jednog PE fajla:

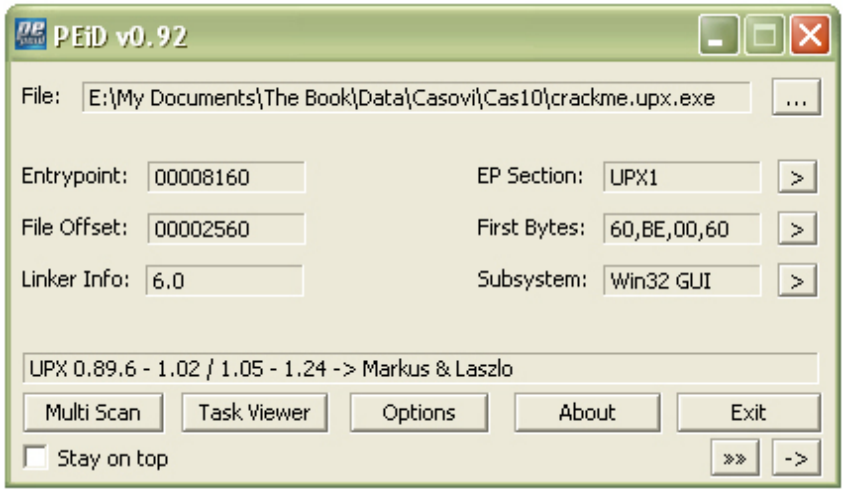



Na slici se vidi da nas posmatrani dummy.dll fajl ima samo dve export funkcije i da su njihova imena DoMyJob i LoadDll. Pored ovih bitnih podataka ExportTable deo LordPEa nam pruza jos dosta bitnih informacija o samoj export tabeli. Takodje su nam bitni podaci koji se nalaze u poljima: NumberOfFunctions, AddressOfFunctions, AddressOfNames i Offset to Export Table. Ovi podaci nam govore o virtualnim lokacijama same tabele, stringova koji cine imena .dll funkcija i lokaciji ordinalnih brojeva funkcija. Prilikom reversinga .dll fajlova nisu nam bitni ovi podaci, bitan nam je samo spisak .dll funkcija i njihove lokacije u samom .dll fajlu kako bismo reversovali fajl na pravom mestu.

# UPX 0.89.6 - 1.02 / 1.05 - 1.24

Kao klasican primer packera, ali nikako i protektora,navodi se UPX. Ovaj odlican paker je najbolji i najlaksi primer kako se to radi unpacking zapakovanih aplikacija. Pre nego sto pocnemo sa objasnjenjem kako to packeri rade, najjednostavnije je da razmisljate o pakovanim exe fajlovima kao da su u pitanju standardni SFX exe fajlovi. To jest razmisljajte o tome da imate arhivu, rar ili zip stvarno je nebitno, i da ste od nje napravili SFX exe fajl. Kada startujete SFX prvo ce se startovati kod SFXa koji ce otpakovati pakovani sadrzaj negde na disk. Isto se desava i sa exe pakerima kao sto je UPX samo sa razlikom sto se pakovani kod otpakuje direktno u memoriju a ne na hard disk.

Krenucemo sa otpakivanjem programa koji se nalazi u folderu Cas10 a zove se crackme.upx.exe. Prvo sto moramo uraditi je identifikacija pakera kojim je nasa meta pakovana. Stoga cemo pritisnuti desno dugme na taj fajl i selektovati Scan with PeID. Pojavice se prozor sa slike. Kao sto vidimo PeID




nam daje dosta jako korisnih detalja u vezi programa koji pokusavamo da otpakujemo. Sada znamo koja je glavna izvrsna sekcija (UPX1) znamo OEP (8160) znamo i sa kojom je to verzijom i kojim je pakerom zapakovana meta (UPX 0.89 - 1.24). Ove verzije packera oznacavaju da se princip otpakivanja nije promenio od verzije 0.89 i da bez ikakvih problema mozemo otpakovati bilo koju veziju UPXa primenjujuci isti metod. Otvoricemo metu pomocu Ollyja i videcemo upozorenje da je u pitanju zapakovani PE fajl pa da se zbog toga mora paziti gde i kako postavljamo breakpointe. Ignorisite ovo upozorenje posto se ovo kada su upitanju zapakovani fajlovi podrazumeva. Prve linije koda ili sam OEP izgledaju bas ovako:

```
00408160 > $  60              PUSHAD
00408161   .  BE 00604000     MOV ESI,crackme_.00406000
00408166   .  8DBE 00B0FFFF   LEA EDI,DWORD PTR DS:[ESI+FFFFB000]
0040816C   .  57              PUSH EDI
0040816D   .  83CD FF         OR EBP,FFFFFFFF
00408170   >  EB 10           JMP SHORT crackme_.00408182
```

I ovaj sablon je isti za svaku verziju UPXa po cemu se UPX i raspoznaje od drugih pakera. Otpakivanje UPXa je veoma lako, postoji veoma mnogo nacina da se otpakuje program pakovan program UPXom ali ja cu vas nauciti najlaksi i najbrzi nacin kako da dodjete do samog OEPa i tu uradite memory dump. Otpakivanje UPXovanih programa se svodi na jednostavno skrolovanje

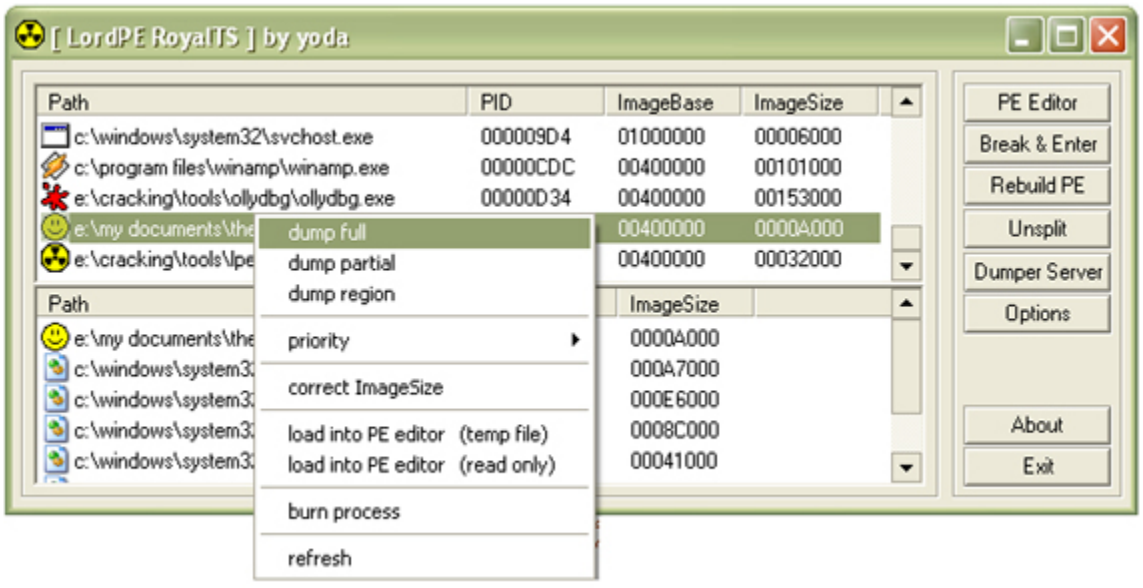
nanize dok ne dodjete do zadnje komande iz koje se nalaze samo neiskorisceni 0x00 bajtovi. Tih par poslednjih redova izgledaju ovako:

```
004082A8   > \FF96 54850000 CALL DWORD PTR DS:[ESI+8554]
004082AE   >  61            POPAD
004082AF   .- E9 0C90FFFF   JMP crackme_.004012C0
```

Ako ste u nekim drugim tutorijalima citali kako treba traziti sledecu POPAD komandu i to je tacno isto jer posle POPAD komande nalazi se samo jos jedna komanda koja kada se izvrsi vodi nas pravo na OEP. Posto je u pitanju bezuslovni skok (JMP) sama adresa ka kojoj on vodi je adresa samog OEPa. Ovo znaci da se OEP nalazi na adresi 004012C0. Ako odete na adresu 004012C0 videcete da se tamo ne nalazi nista, to jest nalazi se samo gomila praznih 0x00 bajtova. Ovo se desava zato sto algoritam za otpakivanje nije izvrsio otpakivanje zapakovanog dela fajla u memoriju i stoga originalni OEP jos ne postoji. Ono sto moramo da uradimo je da nateramo algoritam za otpakivanje da se izvrsi u potpunosti da bismo onda mogli da uradimo dump memorije. Da bismo ovo uradili postavicemo breakpoint na adresu na kojoj se nalazi jump koji vodi direktno do OEPa, to jest na adresu 004082AF. Sa F9 cemo startovati program i Olly ce zastati na adresi na kojoj smo postavili breakpoint. Potrebno je jos samo izvrsiti ovaj skok da bismo se nasli na OEPu. Pritisnucemo F8 i naci cemo se na OEPu.

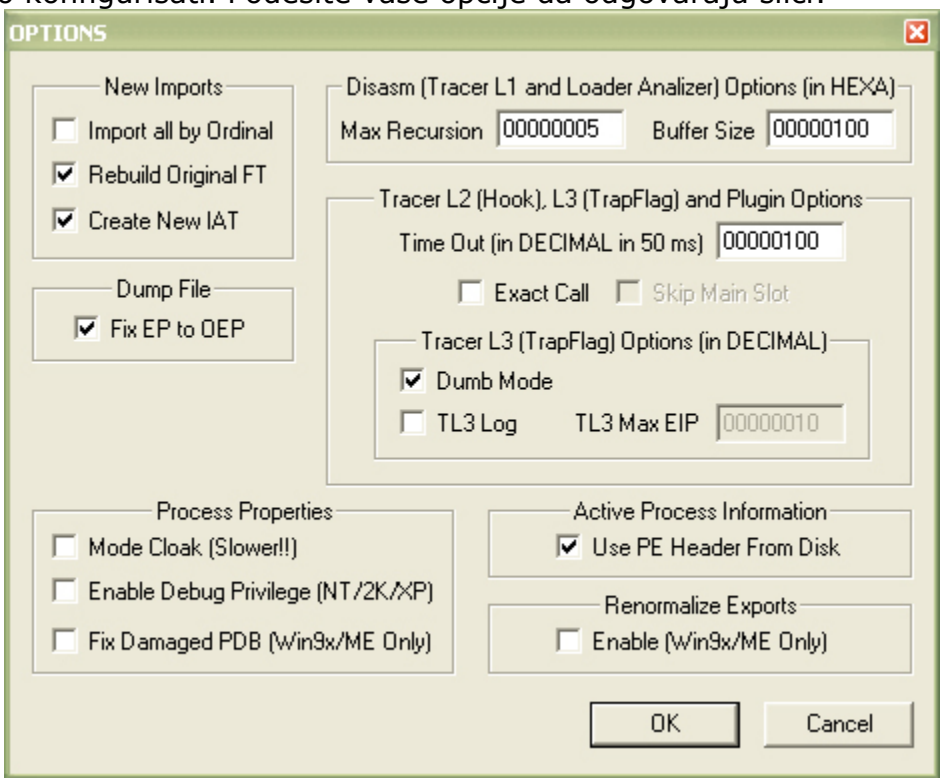
```
004012C0   55              PUSH EBP
004012C1   8BEC            MOV EBP,ESP
004012C3   6A FF           PUSH -1
004012C5   68 F8404000     PUSH crackme_.004040F8
004012CA   68 F41D4000     PUSH crackme_.00401DF4
004012CF   64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
004012D5   50              PUSH EAX
```

Sada kada se nalazimo na OEPu treba da uradimo memory dump kako bismo sacuvali memorijsku sliku otpakovanog fajla na hard disk. Za ovo ce nam trebati LordPE. Startujte ga i iz liste procesa selektujte crackme.upx.exe fajl. Klikom na desno dugme na selektovani fajl izabracemo opciju dump full kao na slici.



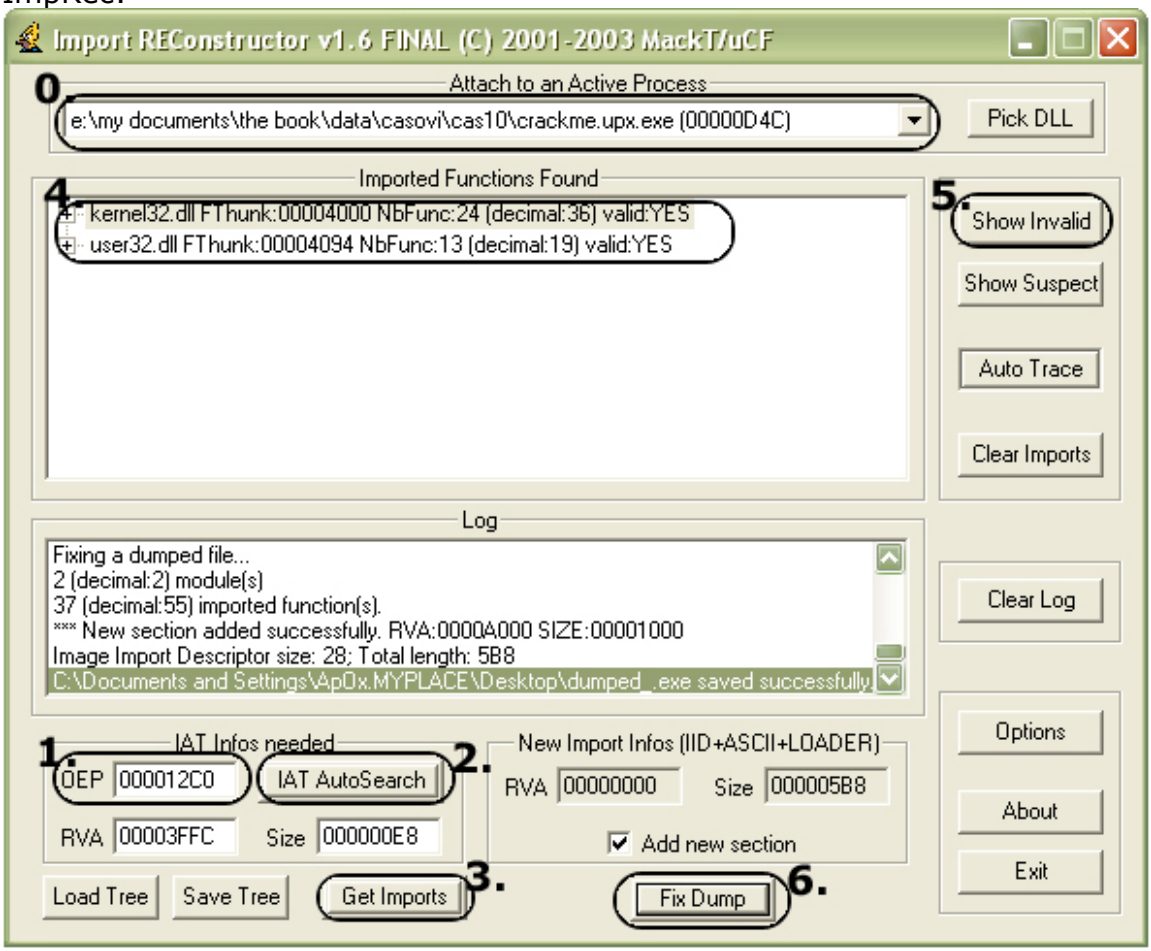

Kada smo snimili memorijsku sliku nase mete bilo gde na disk mozemo da zatvorimo LordPE posto nam on nece vise trebati. Pitate zasto nisam koristio

Ollyjev dump plugin? Odgovor je jednostavan: Sve vezije koje sam ikada testirao je imalo neku gresku i 50% fajlova koje sam dumpovao imali su neku gresku, zbog cega nisu mogli da se startuju, zato vise volim dobri stari full dump iz LordPEa pa preporucujem i vama da ga koristite.
Ako ste pomislili da je gotovo sa otpakivanjem UPXa, prevarili ste se. Ostaje nam jos dosta posla kako bismo naterali metu da se startuje. Zapamtili smo adresu na kojoj se nalazi pravi OEP: 004012C0, ona ce nam trebati kako bismo ispravili OEP na pravu vrednost. Ali pre nego sto ispravimo OEP moracemo da popravimo importe posto je UPX ostavio samo par importa koji su njemu potrebni kako bi otpakovao zapakovani kod u memoriju. Da bi smo popravili importe treba nam jedan drugi alat, treba nam Import Reconstructer. Ne gaseci Olly i ne pomerajuci se sa linije OEPa startujete Import Reconstructer ili ImpRec. Pre upotrebe ImpReca moramo ga prvo pravilno konfigurisati. Podesite vase opcije da odgovaraju slici:




Kao sto primecujete podesio sam ImpRec tako da on sam prilikom popravljanja importa popravi i OEP sto ce nam samo sacuvati vreme odlaska u neki PE editor kako bismo popravili OEP. Kada smo konfigurisali ImpRec selektovacemo iz liste aktivnih procesa nas crackme.upx.exe fajl i sacekacemo da se proces ucita. Kada se ovo desi videcemo da je vrednost OEPa ona stara 00008160 pa cemo je promeniti na novu kako bi ImpRec pronasao sve importe. Umesto stare vrednosti OEPa unecemo novu, unecemo 004012C0 - ImageBase (00400000) = 000012C0. Obratite paznju na ovo, a to je da se image base mora oduzeti od VA vrednosti originalnog OEPa kako bi ImpRec trazio importe na pravom mestu. Kada ovo uradimo pritisnucemo dugme IATAutoSearch pa dugme GetImports. U gornjem delu

prozora ce se pojaviti svi .dll fajlovi koje ovaj crackme koristi i kao grane istih ce se pojaviti odgovarajuci API pozivi. Provericemo da li su svi pozivi tacni i da li je ImpRec pronasao sve API pozive. Kliknucemo na dugme Show Invalid. Posto se nista nije pojavilo zakljucujemo da su svi importi ispravni. Sada nam ostaje samo pa popravimo importe u fajlu koji smo dumpovali prethodno. Kliknucemo na dugme Fix Dump i selektovacemo fajl koji smo dumpovali sa LordPEom. ImpRec ce napraviti novi fajl koji ce sadrzati popravljene importe. Na sledecoj slici je detaljno objasnjeno kako se koristi ImpRec:




Ono sto je bitno a nije naglaseno na ovoj slici je da checkbox Add new section mora biti selektovan kako bi se importi dodali u novu sekciju PE fajla a ne prepisali preko starih. Ako pogledamo sekcije koje sada ima ovaj novi popravljeni fajl videcemo sledece:

**UPX0 | 00005000 | 00001000 | 00005000 | 00001000 | E0000080**
**UPX1 | 00003000 | 00006000 | 00003000 | 00006000 | E0000040**
**.rsrc | 00001000 | 00009000 | 00001000 | 00009000 | C0000040**
**.mackt | 00001000 | 0000A000 | 00001000 | 0000A000 | E0000060**

Videcemo da je dodata sekcija .mackt koja sadrzi sve ispravne importe. Pitate se zasto sam posvetio cetiri strane ovako jednostavnom pakeru? Razlog je jednostavan. Otpakivanje svih vrsta pakera se zasniva na istim postupcima pa ostale packere necu objasnjavati ovako detaljno, podrazumevacu da sve ove osnovne stvari vec znate.

# UPX-Scrambler RC1.x

Vec je objasnjeno kako se otpakuje UPX. UPX-Scrambler predstavlja samo malu modifikaciju pakerskog koda kako se on ne bi lako prepoznao i jos lakse otpakovao. Ako izuzmemo ovu cinjenicu videcemo da se i ovaj skremblovani UPX veoma lako otpakuje. Meta se nalazi u folderu Cas10 a zove se crc32.upx-scrambler.exe. Zeleo bih da napomenem da sam ovu aplikaciju stavio kao primer samo zato sto nisam imao UPX-Scrambler da zapakujem neki fajl, nadam se da se Ghosthunter nece ljutiti. Inace aplikacija je jako korisna ako zelite da proverite da li se CRC32 nekog fajla promenio.

Otvorite ovu metu pomocu Ollyja i videcete sledece na OEPu:

```
0041544F > $  90                  NOP
00415450   >  61                  POPAD
00415451   .  BE 00E04000         MOV ESI,crc32_up.0040E000
00415456   .  8DBE 0030FFFF      LEA EDI,DWORD PTR DS:[ESI+FFFF3000]
0041545C   .  57                  PUSH EDI
0041545D   .  83CD FF             OR EBP,FFFFFFFF
00415460   .  EB 10               JMP SHORT crc32_up.00415472
00415462   .  EB 00               JMP SHORT crc32_up.00415464
00415464   >^ EB EA               JMP SHORT crc32_up.00415450
00415466   .^ EB E8               JMP SHORT crc32_up.00415450
00415468   >  8A06                MOV AL,BYTE PTR DS:[ESI]
0041546A   .  46                  INC ESI
0041546B   .  8807                MOV BYTE PTR DS:[EDI],AL
```

Ovo ni malo ne podseca na UPX, ali slicnost postoji. Ako pogledate dole na sam kraj algoritma za otpakivanje videcete da se nalazi par komandi slicnih UPXu. Ti zadnji redovi izgledaju ovako:

```
0041559E   > \60                  PUSHAD
0041559F   .- E9 1C69FFFF         JMP crc32_up.0040BEC0
004155A4      BC554100            DD crc32_up.004155BC
004155A8      C4554100            DD crc32_up.004155C4
004155AC      CC                  INT3
004155AD      D4                  DB D4
004155AE      40                  DB 40                    ;  CHAR '@'
004155AF      00                  DB 00
```

Primeticemo slicnost sa UPXom samo sto se u UPX pre zadnjeg skoka nalazi POPAD a ovde se nalazi PUSHAD komanda. Postavite breakpoint na JMP to jest na adresu 0041559F, pritisnite F9 da startujete program i program ce zastati na adresi 0041559F. Sada ostaje samo da pritisnemo F8 i da izvrsimo i ovaj skok kako bi smo se nasli na OEPu. Posle izvrsenja ovog skoka nalazimo se ovde:

```
0040BEC0   55                     PUSH EBP                 ; USER32.77D40000
0040BEC1   8BEC                   MOV EBP,ESP
0040BEC3   83C4 F4                ADD ESP,-0C
0040BEC6   B8 38BE4000            MOV EAX,crc32_up.0040BE38
```

Ovo je pravi OEP i tu treba da uradimo memory dump. Posle dumpa treba da uradimo jos popravljanje importa sa ImpRecom i uspesno smo zavrsili sa otpakivanjem UPX-Scramblera.

# UPX Protector 1.0x

Jos jedna u seriji modifikacija UPX kompresora. Meta koju cemo otpakivati je NFO builder a nalazi se ovde ...\Cas10\NFO-Builder.2000.rar kao sto vidite ovo je izuzetno koristan NFO editor stoga spajamo lepo i korisno. Mislim da mi momci iz FMWa nece zameriti sto se sluzim njihovom aplikacijom da objasnim otpakivanje UPX Protectora.

Otvorite metu pomocu Ollyja i pogledajte sta se nalazi na OEPu. Kao sto primecujute OEP je malo neobican:

```
0044E91E   .  FF00              INC DWORD PTR DS:[EAX]
0044E920   >  60                PUSHAD
0044E921   .  BE 00204300       MOV ESI,NFO_Buil.00432000
0044E926   .  8DBE 00F0FCFF     LEA EDI,DWORD PTR DS:[ESI+FFFCF000]
0044E92C   .  57                PUSH EDI
0044E92D   .  83CD FF           OR EBP,FFFFFFFF
0044E930   .  EB 10             JMP SHORT NFO_Buil.0044E942
0044E932 > $^ EB EC             JMP SHORT NFO_Buil.0044E920
```

ali bez obzira na sve i ovaj UPX "Protector" se otpakuje na isti nacin kako i klasicna UPX i kao UPX Scrambler. Mislim da je ovaj protector napisan samo zato da se aplikacije pakovane UPXom ne bi otpakivale preko -d opcije u UPXu. Za one koji to ne znaju UPXovana aplikacija se moze otpakovati preko samog pakera koji se moze preuzeti sa http://upx.sourceforge.net. Sve sto treba da uradite je da startujete upx.exe sa parametrom upx -d imefajla.exe i on ce se otpakovati za nekoliko sekundi. Naravno UPX ne moze da otpakuje programe "zasticene" UPX Protectorom i UPX Scramblerom. Ovo moramo da uradimo rucno.

Jednostavno cemo odskrolovati na sam kraj algoritma za otpakivanje i videcemo sledece:

```
0044EA71   > \61                POPAD
0044EA72   .  C3                RET
0044EA73   >  61                POPAD
0044EA74   .  EB 06             JMP SHORT NFO_Buil.0044EA7C
0044EA76      AD                DB AD
0044EA77      FA                DB FA
0044EA78      EA                DB EA
0044EA79      00                DB 00
0044EA7A   .  00EA              ADD DL,CH
0044EA7C   >- E9 2B52FCFF       JMP NFO_Buil.00413CAC
```

Videcemo da se ispod prve POPAD komande nalazi RET, sto nam je totalno nebitno, a ispod druge skok koji vodi na drugi skok 0044EA7C, koji dalje vodi na sam OEP. Vidimo da cak iako je ovaj algoritam malo promenjen osnove otpakivanja ostaju iste. Dakle sve sto treba da uradimo je da postavimo breakpoint na zadnju jump komandu, da pritisnemo F9 da bismo dosli do samog breakpointa i da pritisnemo F8 1x kako bismo se nasli na OEPu. I evo kako to OEP izgleda:

```
00413CAC   55                   PUSH EBP
00413CAD   8BEC                 MOV EBP,ESP
00413CAF   6A FF                PUSH -1
00413CB1   68 D8694100          PUSH NFO_Buil.004169D8
```

Posle ovoga ostaje samo da uradimo memory dump i popravku importa na isti nacin kao da je u pitanju obican UPX.

# UPXShit 0.06

I poslednji paker u seriji UPX modifikacija za ovu knjigu (*obecavam :)*) je UPXShit autora snaker-a. Meta koja je pakovana ovom vrstom pakera je sam PeID pa cemo otpakivati njega. Verzija PeIDa koju cu ja otpakivati je verzija 0.92. Ucitacemo metu u Olly i prve linije koda ce izgledati ovako:

```
0045E3E2 > $  B8 CCE34500    MOV EAX,PEiD.0045E3CC          <- Packer OEP
0045E3E7   .  B9 15000000    MOV ECX,15
0045E3EC   >  803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E3F0   .^ E2 FA          LOOPD SHORT PEiD.0045E3EC
0045E3F2   .^ E9 D6FFFFFF    JMP PEiD.0045E3CD
```

Mozete probati da izvrsite sve ove linije koda sa F8, sto vam ja ne preporucujem jer LOOPD komanda oznacava loop koji ce se odigrati mnogo puta. Kao sto vidimo ovaj loop sluzi za xorovanje memorije bajtom 7F. Da ne bismo izvrsavali LOOPD komandu postavicemo breakpoint na adresu 0045E3F2 i pritisnucemo F9 da dodjemo do njega. Sa F8 cemo izvrsiti taj skok i zavrsicemo ovde:

```
0045E3CD   > /B8 B7E34500    MOV EAX,PEiD.0045E3B7
0045E3D2    |B9             DB B9
0045E3D3    |15             DB 15
0045E3D4    |00             DB 00
0045E3D5    |00             DB 00
0045E3D6    |00             DB 00
0045E3D7    |80             DB 80
0045E3D8   . |34 08 7F       ASCII "4.."
```

Kao sto vidimo Olly nije stigao da analizira ovaj deo koda pa cemo ga naterati da to uradi pritiskom na CTRL+A. Posle analize taj deo koda sada izgleda ovako:

```
0045E3CD   > /B8 B7E34500   MOV EAX,PEiD.0045E3B7
0045E3D2   . |B9 15000000   MOV ECX,15
0045E3D7   > |803408 7F     XOR BYTE PTR DS:[EAX+ECX],7F
0045E3DB   .^|E2 FA         LOOPD SHORT PEiD.0045E3D7
0045E3DD   .^|E9 D6FFFFFF   JMP PEiD.0045E3B8
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E3DD, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E3B8   > /B8 A2E34500   MOV EAX,PEiD.0045E3A2
0045E3BD   . |B9 15000000   MOV ECX,15
0045E3C2   > |803408 7F     XOR BYTE PTR DS:[EAX+ECX],7F
0045E3C6   .^|E2 FA         LOOPD SHORT PEiD.0045E3C2
0045E3C8   .^|E9 D6FFFFFF   JMP PEiD.0045E3A3
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E3C8, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E3A3   ?  B8 8DE34500   MOV EAX,PEiD.0045E38D
0045E3A8   .  B9 15000000   MOV ECX,15
0045E3AD   >  803408 7F     XOR BYTE PTR DS:[EAX+ECX],7F
0045E3B1   .^ E2 FA         LOOPD SHORT PEiD.0045E3AD
0045E3B3   .^ E9 D6FFFFFF   JMP PEiD.0045E38E
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E3B3, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet

zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E38E  > /B8 78E34500    MOV EAX,PEiD.0045E378
0045E393  ? |B9 15000000    MOV ECX,15
0045E398  . |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E39C  ?^|E2 FA          LOOPD SHORT PEiD.0045E398
0045E39E  ?^|E9 D6FFFFFF    JMP PEiD.0045E379
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E39E, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E379  > /B8 63E34500    MOV EAX,PEiD.0045E363
0045E37E  . |B9 15000000    MOV ECX,15
0045E383  > |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E387  .^|E2 FA          LOOPD SHORT PEiD.0045E383
0045E389  .^|E9 D6FFFFFF    JMP PEiD.0045E364
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E389, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E364  > /B8 4EE34500    MOV EAX,PEiD.0045E34E
0045E369  . |B9 15000000    MOV ECX,15
0045E36E  > |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E372  .^|E2 FA          LOOPD SHORT PEiD.0045E36E
0045E374  .^|E9 D6FFFFFF    JMP PEiD.0045E34F
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E374, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E34F  > /B8 39E34500    MOV EAX,PEiD.0045E339
0045E354  . |B9 15000000    MOV ECX,15
0045E359  > |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E35D  .^|E2 FA          LOOPD SHORT PEiD.0045E359
0045E35F  .^|E9 D6FFFFFF    JMP PEiD.0045E33A
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E35F, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E33A  > /B8 24E34500    MOV EAX,PEiD.0045E324
0045E33F  . |B9 15000000    MOV ECX,15
0045E344  > |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E348  .^|E2 FA          LOOPD SHORT PEiD.0045E344
0045E34A  .^|E9 D6FFFFFF    JMP PEiD.0045E325
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E34A, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E325  ?  B8 0FE34500    MOV EAX,PEiD.0045E30F
0045E32A  .  B9 15000000    MOV ECX,15
0045E32F  >  803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E333  .^ E2 FA          LOOPD SHORT PEiD.0045E32F
0045E335  .^ E9 D6FFFFFF    JMP PEiD.0045E310
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E335, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E310  > /B8 FAE24500    MOV EAX,PEiD.0045E2FA
0045E315  . |B9 15000000    MOV ECX,15
0045E31A  > |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E31E  .^|E2 FA          LOOPD SHORT PEiD.0045E31A
0045E320  .^|E9 D6FFFFFF    JMP PEiD.0045E2FB
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E320, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E2FB  > /B8 E5E24500    MOV EAX,PEiD.0045E2E5
0045E300  . |B9 15000000    MOV ECX,15
0045E305  > |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E309  .^|E2 FA          LOOPD SHORT PEiD.0045E305
0045E30B  .^|E9 D6FFFFFF    JMP PEiD.0045E2E6
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E30B, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E2E6  > /B8 D0E24500    MOV EAX,PEiD.0045E2D0
0045E2EB  . |B9 15000000    MOV ECX,15
0045E2F0  > |803408 7F      XOR BYTE PTR DS:[EAX+ECX],7F
0045E2F4  .^|E2 FA          LOOPD SHORT PEiD.0045E2F0
0045E2F6  .^|E9 D6FFFFFF     JMP PEiD.0045E2D1
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E2F6, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E2D1  ?  B8 5FE14500    MOV EAX,PEiD.0045E15F
0045E2D6  ?  B9 71010000    MOV ECX,171
0045E2DB  >  803408 7F       XOR BYTE PTR DS:[EAX+ECX],7F
0045E2DF  .^ E2 FA          LOOPD SHORT PEiD.0045E2DB
0045E2E1  .^ E9 7AFEFFFF    JMP PEiD.0045E160
```

Ponovicemo isti postupak od malopre i postavicemo breakpoint na 0045E2E1, pa cemo pritisnuti F9 da dodjemo do bp-a a onda F8 da izvrsimo skok. I opet zavrsavamo na delu koda koji nije analiziran od strane Ollyja. Posle pritiska na CTRL+A pojavljuje se ovo:

```
0045E160  > /60             PUSHAD
0045E161  . |BE 00D04300    MOV ESI,PEiD.0043D000
0045E166  . |8DBE 0040FCFF  LEA EDI,DWORD PTR DS:[ESI+FFFC4000]
0045E16C  . |57             PUSH EDI
0045E16D  . |83CD FF        OR EBP,FFFFFFFF
```

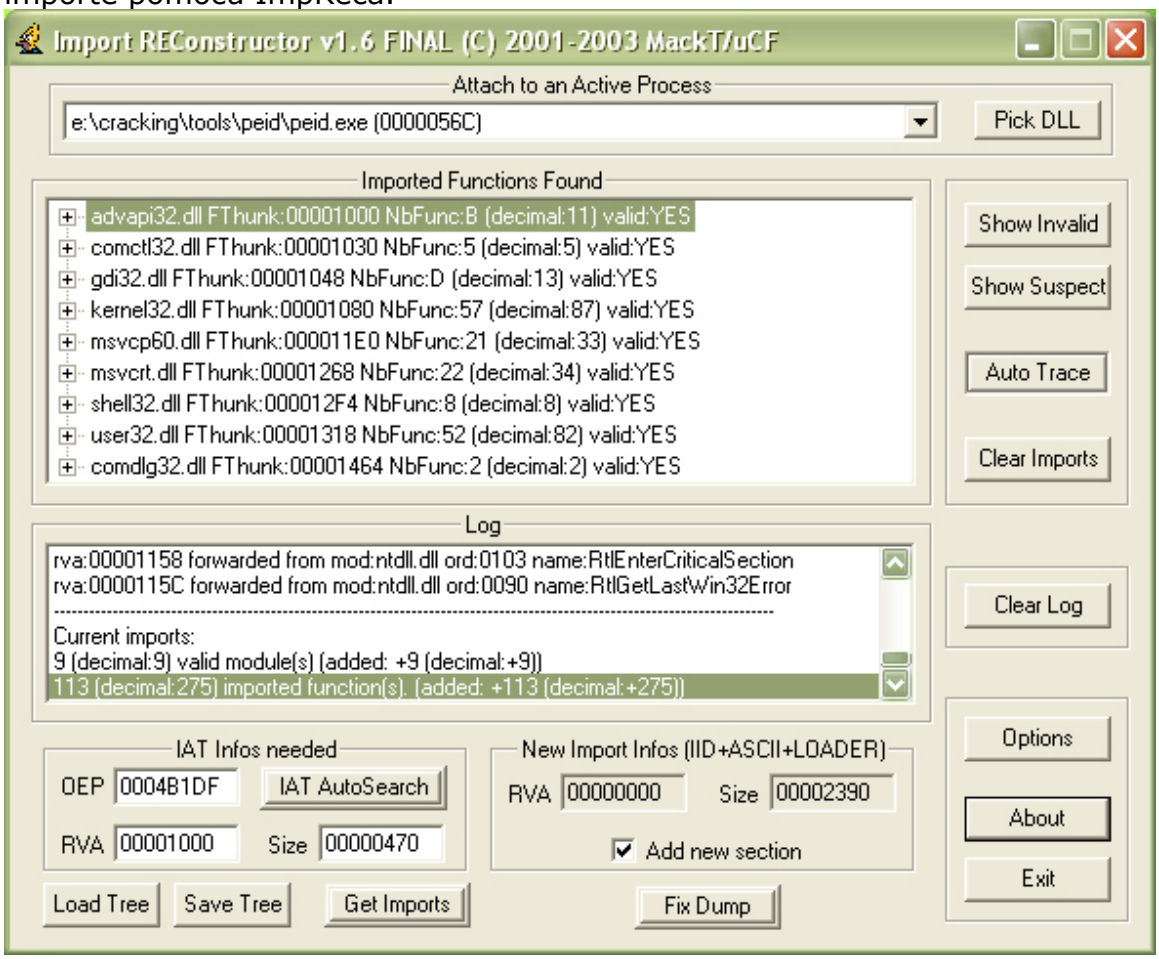
Izvinjavam se sto je malo vise od dve cele strane knjige copy-paste ali tako se otpakuje i zeleo sam da budem siguran da cete ispratiti sve ove jumpove kako treba. Ono sto sada vidimo nas podseca na originalni UPX kod, ali nije bas sasvim. Bez obzira na sve bas kao i obican UPX, UPXShit ima jedan obican JMP skok koji vodi pravo na OEP odmah ispod prve POPAD komande. Taj deo koda izgleda ovako:

```
0045E2BC  . /74 07          JE SHORT PEiD.0045E2C5
0045E2BE  . |8903            MOV DWORD PTR DS:[EBX],EAX
0045E2C0  . |83C3 04         ADD EBX,4
0045E2C3  .^|EB D8           JMP SHORT PEiD.0045E29D
0045E2C5  > \FF96 6CE90500 CALL DWORD PTR DS:[ESI+5E96C]
0045E2CB  >  61             POPAD
0045E2CC  .^ E9 0ECFFEFF    JMP PEiD.0044B1DF
```

pa cemo samo postaviti jedan breakpoint na 0045E2CC, pritisnucemo F9 da dodjemo do njega, i sa F8 dolazimo na OEP koji ce posle analize sa CTRL + A izgledati ovako:

```
0044B1DF  /> /55            PUSH EBP                <- Pravi OEP
0044B1E0  |. |8BEC           MOV EBP,ESP
0044B1E2  |. |6A FF         PUSH -1
0044B1E4  |. |68 702B4200   PUSH PEiD.00422B70
0044B1E9  |. |68 AAB14400   PUSH PEiD.0044B1AA
0044B1EE  |. |64:A1 0000000> MOV EAX,DWORD PTR FS:[0]
0044B1F4  |. |50            PUSH EAX
```

Sada nam ostaje samo da uradimo dump preko LordPEa i da popravimo importe pomocu ImpReca.




Ovo je najbolja modifikacija UPXa koju sam do sada video, stoga moram da priznam da je snaker odradio fenomenalan posao dodajuci enkripciju samog koda za otpakivanje UPX sekcije. Ideja je odlicna ali je otpakivanje bez obzira na sto je malo duze i dalje lako izvodljivo.

# FSG 1.30 - 1.33

FSG takodje sluzi samo za smanjenje velicine konacnog kompajlovanog exe ili dll fajla. On, kao ni UPX, u sebi ne sadrzi kod za zastitu od otpakivanja ili debuggovanja sa Ollyjem ili bilo kojim drugim debuggerom. Pre nego sto pocnemo sa otpakivanjem mete pakovane FSGom objasnicu par stvari vezanih za kod koji otpakuje program u memoriju. Bez obzira na vrstu i verziju pakera sledeca stvar mora uvek biti ispunjena:

***Posle izvrsavanja koda za otpakivanje program ce nekako skociti na virtualnu adresu otpakovanog koda. Ovo skakanje moze biti uradjeno samo na dva nacina, preko komande RET u slucaju da se kod za otpakivanje ponasa kao CALL i preko komande JMP (i varijanti) kada se posle otpakivanja direktno skace na OEP.***

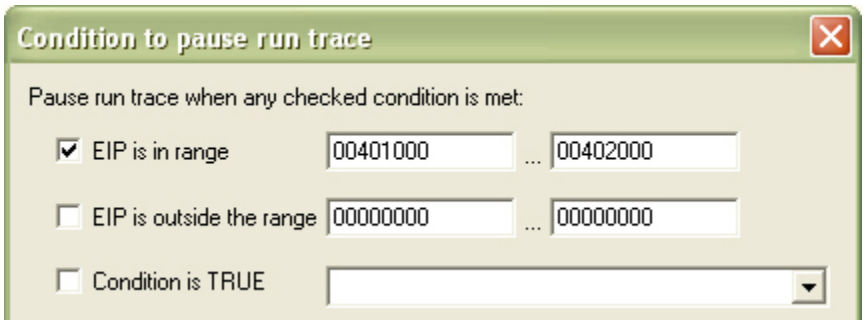
Ovo je najbitnije pravilo sto se tice otpakivanja zapakovanih programa i uvek mora biti ispunjeno sto cemo mi iskoristiti u nasu korist. Meta pakovana sa FSGom se nalazi u folderu Cas10 a zove se crackme.fsg.exe i nju cemo otpakovati uz pomoc Ollyja. Kada otvorite ovu metu pomocu Ollyja videcete sledece komande na mestu OEPa:

```
00405DC5 >  BE A4014000      MOV ESI,crackme_.004001A4
00405DCA    AD               LODS DWORD PTR DS:[ESI]
00405DCB    93               XCHG EAX,EBX
00405DCC    AD               LODS DWORD PTR DS:[ESI]
00405DCD    97               XCHG EAX,EDI
00405DCE    AD               LODS DWORD PTR DS:[ESI]
```

Ocigledno je da je ovo kod koji pripada pakeru i da to nije originalni nepakovani kod. Kao i kod UPX se na kraju koda za otpakivanje nalazi gomila 0x00 bajtova. Deo koda od packer OEPa pa do 00405E8C predstavlja deo koda koji sluzi direktno i samo za otpakivanje. U ovom delu koda cemo pokusati da pronadjemo skok koji bi vodio do adrese koja ne pripada ovom delu koda, od 00405DC5 do 00405E8C. Svi skokovi pripadaju ovom delu adresa osim jednog koji se nalazi na adresi 00405E66.

```
00405E66  - 0F84 26B5FFFF    JE crackme_.00401392
```

Ovaj skok moze voditi direktno do OEPa. Ovo mozemo proveriti tako sto cemo staviti breakpoint na tu adresu i pusticemo je da se izvrsi. Posto je ce ovo malo duze trajati jer se ovaj skok ne izvrsava vise puta ali se preko njega prelazi vise puta pa zbog toga nije pozeljno traziti OEP na ovaj nacin. Mi cemo iskoristiti cinjenicu da se u registru EIP uvek nalazi adresa koja ce sledeca izvrsiti. Dakle ako je EIP = 00401392 sledeca adresa koja ce se izvrsiti je 00401392 to jest pravi OEP. Za ovo cemo iskoristiti mogucnot koju nam pruza Olly: Tracing. Pritisnite CTRL+T, ispunite prozor kao na slici i pritisnite OK. Sada nam ostaje samo da pokrenemo Trace pritiskom na CTRL + 12 ili na Debug -> Trace over u gornjem meniju. Posle nekoliko sekundi Olly ce se zaustaviti ovde:

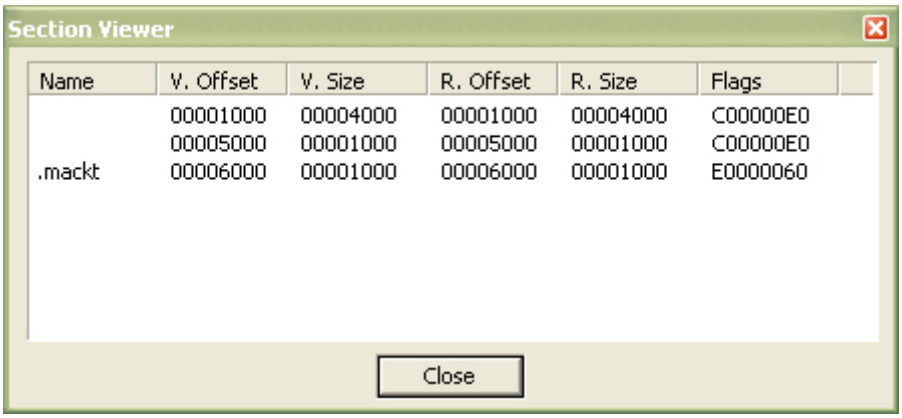


```
00401392    55          DB 55                    ; CHAR 'U'
00401393    8B          DB 8B
00401394    EC          DB EC
00401395    83          DB 83
00401396    EC          DB EC
00401397    44          DB 44                    ; CHAR 'D'
00401398    56          DB 56                    ; CHAR 'V'
00401399    FF          DB FF
0040139A    15          DB 15
0040139B    14          DB 14
```

Ovo sigurno ne izgleda kao OEP programa. Gde li smo to pogresili ??? Odgovor je: Nigde. Ovo je verovali ili ne pravi OEP bas na adresi 00401392. Ovaj deo koda Olly nije stigao da analizira pa izgleda ovako kako izgleda. Pritisnite CTRL + A da biste analizirali kod i umesto gornjeg junka pojavice se pravi OEP.

```
00401392 /. 55              PUSH EBP                 ; USER32.77D40000
00401393 |. 8BEC            MOV EBP,ESP
00401395 |. 83EC 44         SUB ESP,44
00401398 |. 56              PUSH ESI
```

Ostaje nam samo da uradimo dump na adresi 00401392 i da popravimo importe ImpRecom kao sto je objasnjeno za UPX. Ono sto sam primetio prilikom otpakivanja FSGa je da on ne deklarise imena za svoje sekcije. Ovo i nije bitno jer PE fajl moze da radi sasvim OK i bez imena sekcija. Ovo se vidi u svim section viewerima. Jedan takav imamo u PeIDu a do njega se dolazi klikom na dugme > pored prve EP Sekcije. U slucaju FSGa vidimo ovo:




## FSG 2.0

FSG 2.0 ne donosi mnogo novina u odnosu na FSG 1.33 koji smo vec uspesno otpakovali. Jedina stvarna razlika je u nacinu na koji FSG skace na pravi OEP otpakovane mete. Ovo se jasno vidi na sledecem isecku koda:

```
004001CD  ^\78 F3           JS SHORT unpackme.004001C2
004001CF   75 03            JNZ SHORT unpackme.004001D4
004001D1   FF63 0C          JMP NEAR DWORD PTR DS:[EBX+C]        <- JMP to OEP
```

Poslednji skok vodi direktno ka OEPu a na adresi EBX+C se uvek nalazi adresa pravog OEPa sto se i vidi ako odemo na adresu 00408414:

```
00408414  00 10 40 00 61 D9 E7 77  ..@.a..w                     <- OEP
```

# ASPack 1.x - 2.x

Ovaj paker se nije mnogo menjao kroz svoje verzije kao ni UPX, dakle ako mozete da otpakujete ovaj primer mozete da otpakujete bilo koju verziju ASpacka. Otpakujemo fajl damn_contest.aspack.exe koji se nalazi u folderu Cas10. Za razliku od proslih pakera u ovom cemo koristiti Ollydump plugin jer ovde skracuje posao otpakivanja i popravljanja importa. Otvorimo Olly i ucitajmo damn_contest.exe u njega  File -> Open. U CPU Windowu [ ALT + C ] vidimo prve redove fajla:

```
00411000 >  60              PUSHAD
00411001    E8 00000000     CALL damn_con.00411006
00411006    5D              POP EBP
00411007    81ED 0A4A4400   SUB EBP,444A0A              ; UNICODE "ymunds"
0041100D    BB 044A4400     MOV EBX,444A04
```

Ovo nije originalni kod DAMN tryme-a nego je kod ASpacka koji se prvo izvrsava i sluzi za otpakivanje ASpackovanog fajla u memoriju. Pritisnite F8 da biste dosli do linije 00411001 odnosno da biste presli preko PUSHAD komande.

```
EAX 00000000
ECX 0006FFB0
EDX 7FFE0304
EBX 7FFDF000
ESP 0006FFA4              <- Crven
EBP 0006FFF0
ESI 00000000
EDI 00000000
EIP 00411001 damn_con.00411001
```

Pogledajmo CPU registre. Sve je isto samo se ESP promenio. Desno dugme na njega i klik na Follow in Dump. Dole u Hex dumpu ce se pojaviti ovo:

```
0006FFA4  00 00 00 00 00 00 00 00  ........
0006FFAC  F0 FF 06 00 C4 FF 06 00  ðÿ□.Äÿ□.
0006FFB4  00 F0 FD 7F 04 03 FE 7F  .ðý□□□þ□
0006FFBC  B0 FF 06 00 00 00 00 00  °ÿ□.....
```

Sada selektujte prva cetiri para nula. Desno dugme na selektovano pa Breakpoint -> Hardware, on access -> Dword. Postavili smo breakpoint i program ce stati kad dodje do selektovane linije. Pritisnimo jedno F9 da startujemo program. Zavrsili smo ovde:

```
00411559   /75 08           JNZ SHORT damn_con.00411563      <- Ovde smo
0041155B   |B8 01000000     MOV EAX,1
00411560   |C2 0C00         RETN 0C
00411563   \50              PUSH EAX
00411564    C3              RETN
00411565    55              PUSH EBP
00411566    8BEC            MOV EBP,ESP
```

Pritisnite F8 2x i pustite RETN komandu da se izvrsi. Ona nas vodi direktno do prve linije programa pre pakovanja sa ASpackom tj. vodi nas do OEPa. Vidimo nesto neobicno:

```
00401000    6A        DB 6A                    ;  CHAR 'j'
00401001    00        DB 00
00401002    E8        DB E8
00401003    28        DB 28                    ;  CHAR '('
```

itd itd, ali verujte mi na rec to je OEP, tu smo. Pritisnite CTRL + A da analizirate fajl i umesto ovoga gore pojavice se:

```
00401000  .  6A 00          PUSH 0
00401002  .  E8 28040000    CALL damn_con.0040142F
00401007  .  BF 0B234000    MOV EDI,damn_con.0040230B
```

```
0040100C   .  85C0                TEST EAX,EAX
0040100E   .  74 1E               JE SHORT damn_con.0040102E
00401010   .  A3 95234000         MOV DWORD PTR DS:[402395],EAX
00401015   .  8307 01             ADD DWORD PTR DS:[EDI],1
00401018   .  33C0                XOR EAX,EAX
0040101A   .  50                  PUSH EAX                    ; /lParam => NULL
0040101B   .  68 45104000         PUSH damn_con.00401045
00401020   .  50                  PUSH EAX                    ; |hOwner => NULL
00401021   .  6A 73               PUSH 73                     ; |pTemplate = 73
00401023   .  FF35 95234000       PUSH DWORD PTR DS:[402395]
00401029   .  E8 B3030000         CALL damn_con.004013E1
0040102E   >  6A 00               PUSH 0                    ; /ExitCode = 0
00401030   .  E8 F4030000         CALL damn_con.00401429    ; \ExitProcess
```

Sta sam vam rekao na OEPu smo. Sada preostaje samo da izvrsimo dump programa i to je to. Idemo na Plugins -> OllyDump -> Dump debugged proccess. Default podesavanja su OK pritisnite Dump i snimite fajl. Zatvorite Olly i probajte da startujete novi fajl i on radi! Ok uspeli smo. Da bi smo proverili da li je sve OK otvorimo dumpovan fajl u Ollyju. Idemo na Desno dugme -> Search for -> All referenced strings. Vidimo ovo:

```
Text strings referenced in dmp:CODE
Address   Disassembly                    Text string
00401000  PUSH 0                         (Initial CPU selection)
0040108D  PUSH dmp.00402023              ASCII "-=[ ABOUT ]=-"
00401092  PUSH dmp.00402031              ASCII "You are just trying to solve DAMN's Official
joinig Contest. Made by tHE EGOiSTE/DAMN. At
first make a keygen for this simple keycheck
routine, then try to crack this program. The
LOCKED - Button should show an UNLOCKED-sign
and if "...
00401107  PUSH dmp.0040227D              ASCII "-=[ YEAH! ]=-"
0040110C  PUSH dmp.0040220E              ASCII "You got it! Thank you for registering!"
00401124  PUSH dmp.00402353              ASCII "                                      "
00401149  PUSH dmp.00402321              ASCII "                                      "
0040127E  PUSH dmp.00402317              ASCII "About"
004012CE  PUSH dmp.0040228B              ASCII "-=[ CHECK ]=-"
00401301  MOV EDI,dmp.00402353           ASCII "                                      "
```

Ovo izgleda OK. Sada idemo na ALT + E da proverimo da li su svi importi OK. Desno dugme na ime .exe fajla pa na View names. Tabela izgleda ovako:

```
Names in dmp
Address   Section   Type    (  Name                        Comment
004030B0  .idata    Import  (  user32.AppendMenuA
004030B4  .idata    Import  (  user32.DeleteMenu
004030F8  .idata    Import  (  GDI32.DeleteObject
004030B8  .idata    Import  (  user32.DialogBoxParamA
004030BC  .idata    Import  (  user32.DrawMenuBar
004030C0  .idata    Import  (  user32.EnableWindow
004030C4  .idata    Import  (  user32.EndDialog
004030EC  .idata    Import  (  kernel32.ExitProcess
004030C8  .idata    Import  (  user32.GetDlgItem
004030CC  .idata    Import  (  user32.GetDlgItemTextA
004030F0  .idata    Import  (  kernel32.GetModuleHandleA
004030D0  .idata    Import  (  user32.GetSystemMenu
004030D4  .idata    Import  (  user32.LoadBitmapA
004030D8  .idata    Import  (  user32.LoadIconA
004030DC  .idata    Import  (  user32.MessageBoxA
00401000  CODE      Export     <ModuleEntryPoint>
004030E0  .idata    Import  (  user32.SendDlgItemMessageA
004030E4  .idata    Import  (  user32.SendMessageA
004030AC  .idata    Import  (  user32.SetWindowTextA
004030A8  .idata    Import  (  user32.wsprintfA
```

Sve izgleda OK. Znaci sve je OK tj. raspakovali smo fajl kako treba! Po importima se zakljucuje da je ovaj .exe fajl pisan u ASMu.

# PETite 2.2

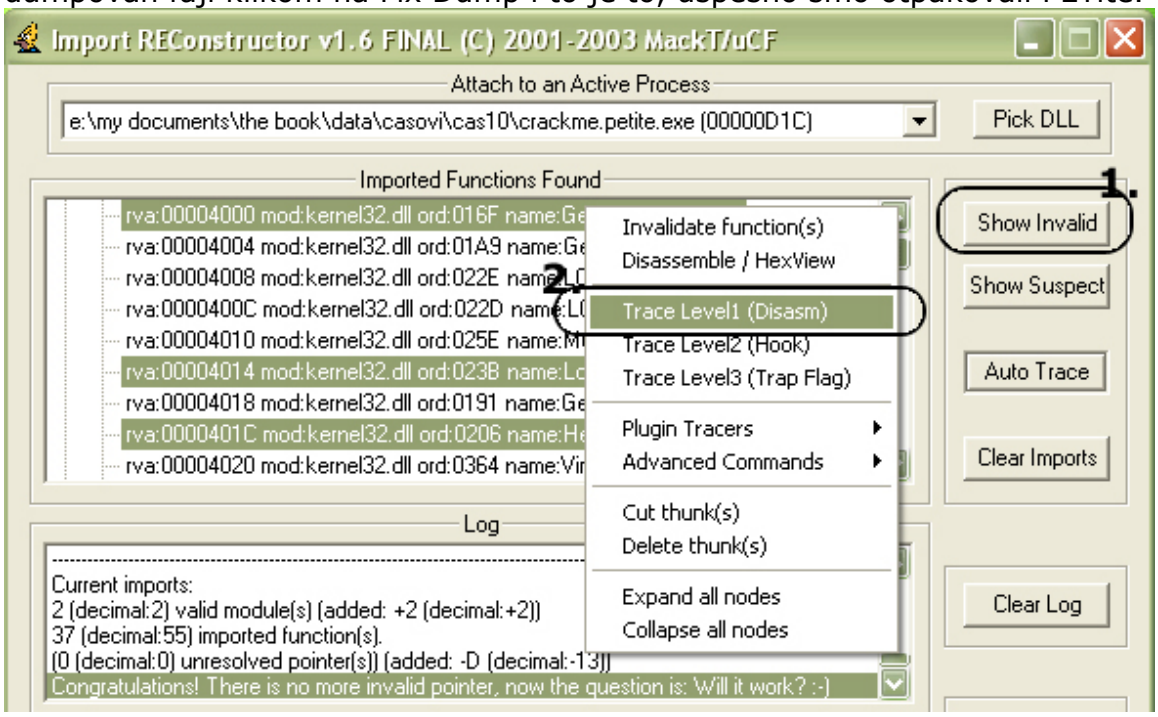
Do sada smo rucno trazili OEPe, ali cemo za otpakivanje PEtitea koristiti Ollyjev plugin OllyScript kako bismo brze i lakse nasli sam OEP. Meta koju cemo koristiti se nalazi u folderu Cas10 a zove se crackme.petite.exe a olly skripta koja ce nam trebati se nalazi u fajlu ...\Cas10\OllyDBG scripts.rar a zove se petite2.2.txt.
Kada imamo sve sto nam treba mozemo da pocnemo sa otpakivanjem mete. Ucitajte metu u Olly. Ako Olly bude izbacivao neke poruke, pritisnite Yes. Sada samo treba da izvrsimo petite2.2.txt pomocu Plugins -> OllyScript -> Run Script... Pritisnemo OK tri puta i nalazimo se na OEPu. Brzo zar ne? Ono sto vidimo je ovo:

```
004012C0    55          DB 55                          ; This is the entry point
004012C1    8B          DB 8B
```

Naravno ovo je sigurno OEP samo treba da ga analiziramo pomocu Ollyja. Pritiskom na CTRL + A vidimo ovo:

```
004012C0  /. 55             PUSH EBP                   ; This is the entry point
004012C1  |. 8BEC           MOV EBP,ESP
```

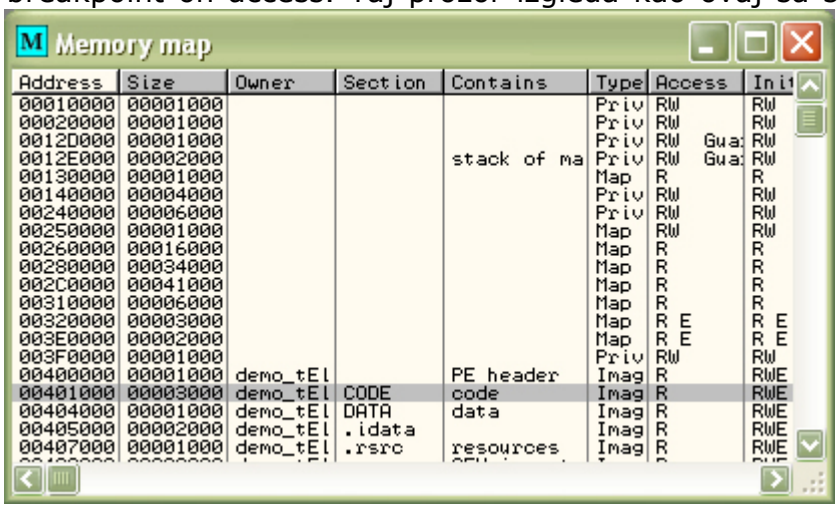
Sada treba da uradimo standardan memory dump i da popravimo importe. Dump cemo uraditi preko LordPEa, uradicemo full dump. Sve do sada je bilo lako, i ostatak je lak samo morate da naucite par novih trikova. Otvoricemo ImpRec i ucitacemo crackme.petite.exe u njega. Moramo da promenimo OEP podesavanja kako bismo dobili tabelu importa za korektan OEP. Promenicemo OEP na adresu 000012C0, pritisnucemo IATAutoSearch, pa GetImports. Kao sto vidimo nisu svi importi tacni, za oba .dll fajla nam fali po nekoliko importa. Kliknucemo na dugme Show Invalid, pa cemo onda pronaci sve netacne importe desnim klikom na selektovane importe i selekcijom Trace Level1 dobicemo sve importe koji nam fale. Ostaje samo da popravimo dumpovan fajl klikom na Fix Dump i to je to, uspesno smo otpakovali PETite.



# tElock 0.80

Moram da priznam da sam fajl sa ovom zastitom slucajno nasao medju svojom kolekcijom targeta. Naime ovaj fajl se nalazio zajedno sa programom pod imenom DZA patcher crackerske grupe TNT! Posto nisam nikada pre otpakivao ni jedan program pakovan sa tElockom odlucio sam da probam. Naravno posto nigde nisam nasao tutorijal kako bih otpakovao ovu zastitu (*a nisam ni trazio posto pri ruci nisam imao internet pa sam samo prekopao svoj hard disk*) odlucio sam da ovo uradim onako muski, na nevidjeno. Meta za ovaj deo poglavlja ce biti fajl demo.tElock.exe, ovaj fajl cemo ucitati u Olly i pokusacemo da ga otpakujemo. Imajte na umu da sam se ja mucio oko ovoga nesto malo manje od pola sata i da cu neke stvari ovde izgledati mozda malo nelogicne ali su tu kako bi ubrzale proces otpakivanja, stoga ih shvatite kao apriore, to jest kao cinjenice koje su takve i ne dokazuju se.

Posle vecanja kako da otpakujem ovu metu odlucio sam da je najlakse da postavim jedan memorjski breakpoint na CODE sekciju fajla i da odatle krenem sa reversingom. Da bismo ovo uradili moramo da odemo u prozor modules ALT+M i da desnim klikom na sekciju CODE postavimo Set Memory breakpoint on access. Taj prozor izgleda kao ovaj sa slike dole. Ja sam vec




vise puta startovao ovaj program pa znam da ce vam on praviti gomilu exceptiona i da necete moci da ih sve zaobidjete pritiskom na CTRL+F9. Zato cemo pritiskati CTRL+F9 onoliko puta koliko je potrebno da se predju svi ovi exceptioni i da se dodje do prvog memorijskog breakpointa. Ja sam pritisnuo CTRL+F9 cak 10x kako bih stigao do prvog memorijskog breakpointa. Posto sam ja ovaj kod vise puta analizirao otkrio sam da on sluzi za otpakivanje koda u memoriju. Sada se mozemo vratiti u prozor Memory map i iskljuciti breakpoint na CODE sekciji. Kada se vratimo nazad u CPU prozor videcemo gde smo to stali sa izvrsavanjem koda.

```
0040868F   AC              LODS BYTE PTR DS:[ESI]
00408690   F6D8            NEG AL
00408692   04 01           ADD AL,1
00408694   8AD2            MOV DL,DL
```

Ovaj deo koda nam je totalno nerazumljiv ali posle dosta (*stvarno dosta*) traceovanja dosao sam do zakljucka da i ovaj deo koda sluzi za otpakivanje u memoriju. Da bismo preskocili ceo ovaj kod za otpakivanje skrolovacemo dole do adrese 004087BA na kojoj se nalazi jedna JMP EAX komanda. Na nju cemo postaviti obican breakpoint. Ako vas Olly nesto pita odgovorite mu sa YES. Sada cemo pritisnuti F9 da bismo dosli do tog breakpointa. Pritisnucemo

F9 jos 2x kada stignemo do tog breakpointa. Zasto? Zato sto ce se dole u Hex Dumpu (donji levi deo prozora) prikazivati otpakovani bajtovi. Kada i treci put pritisnemo F9 u Hex Dumpu cemo imati ovo:

```
00404000  55 4E 52 45 47 49 53 54  UNREGIST
00404008  45 52 45 44 21 21 21 21  ERED!!!!
00404010  21 21 21 21 21 21 21 21  !!!!!!!!
00404018  21 00 54 68 69 73 20 69  !.This i
00404020  73 20 74 68 65 20 55 4E  s the UN
00404028  52 45 47 49 53 54 45 52  REGISTER
00404030  45 44 20 76 65 72 73 69  ED versi
00404038  6F 6E 2C 79 6F 75 20 6D  on,you m
00404040  75 73 74 20 50 41 59 21  ust PAY!
00404048  00 00 00 00 00 00 00 00  ........
```

a ovo izgleda kao otpakovani kod. Mozete da pritisnete F9 i cetvrti put ali ce se program startovati i mi necemo stici do naseg OEPa. Verujte mi probao sam vise puta ovo. Zato cemo pritisnuti F8 sto ce nas odvesti do sledece adrese:

```
00408975   61              POPAD
00408976   56              PUSH ESI
00408977   57              PUSH EDI
```

Polako sa F8 cemo vrsiti trace nadole sve dok ne stignemo do adrese:

```
004089AD  ^\7F D3          JG SHORT demo_tEl.00408982
```

Ovo je jedan jakooo dugacak loop u verujte mi ne zelite da ga izvrsavate celog, stoga cemo postaviti breakpoint odmah ispod ove adrese, postavicemo breakpoint na:

```
004089AF   5F              POP EDI
```

I dalje traceujemo na dole sa F8 sve dok ne dodjemo do adrese:

```
00408BBF   FF95 889C4000   CALL DWORD PTR SS:[EBP+409C88]
```

kada se u registrima pojavljuje ime user32.dll fajla. Posle detaljne analize shvatio sam da ovaj deo koda sluzi za otpakivanje i rekonstrukciju IATa (*import tabele*) stoga sam traceovao do zadnje adrese u ovom loopu koji sluzi za otpakivanje IATa. Drzacemo F8 sve dok ne dodjemo do ovoga:

```
00408E49   8803            MOV BYTE PTR DS:[EBX],AL
00408E4B   43              INC EBX
00408E4C   3803            CMP BYTE PTR DS:[EBX],AL
00408E4E  ^ 75 F9          JNZ SHORT demo_tEl.00408E49
00408E50   8385 1A9D4000 0>ADD DWORD PTR SS:[EBP+409D1A],4
00408E57  ^ E9 5DFEFFFF    JMP demo_tEl.00408CB9
```

Kada dodjemo do ovoga takodje cemo imati loop koji ubacuje API pozive u IAT tabelu. Postavicemo stoga breakpoint na 00408E57 i pritisnucemo F9 da dodjemo do njega. Sa F8 se ponovo vracamo na pocetak ovog loopa za otpakivanje IATa. Pritisnucemo F9 10x sto je ekvivalentno tome da smo otpakovali u memoriju deset API poziva. Zasto bas 10x? Pa mozete sa F8 traceovati ceo ovaj loop 10x i videcete da cete stalno dolaziti do breakpointa na adresi 00408E57. Sada cemo sa F8 preci jos 1x ovaj deo koda za otpakivanje IATa i ponovo cemo stici od 00408E57. Drzacemo sada ponovo F8 kako bismo otpakovali sledecu API funkciju, ali... Izgleda da nema vise API funkcija jer smo sada stigli do jednog drugog loopa koji kako vidimo prepravlja memoriju. Ovaj loop se nalazi ovde:

```
00408EFB   AC              LODS BYTE PTR DS:[ESI]
00408EFC   3206            XOR AL,BYTE PTR DS:[ESI]
00408EFE   AA              STOS BYTE PTR ES:[EDI]
00408EFF  ^ E2 FA          LOOPD SHORT demo_tEl.00408EFB
```

Posto je i ovo veoma dugacak loop postavicemo breakpoint na CALL koji se nalazi odmah ispod LOOPD komande, postavicemo breakpoint na adresu:

```
00408F01   E8 2A3E3E3E       CALL 3E7ECD30
```
i pritisnucemo F9 da dodjemo do te adrese. Posto ovaj CALL izgleda zanimljivo pritisnucemo F7 da udjemo u njega i naci cemo se ovde:
```
00408F1A   FFB5 DE9D4000     PUSH DWORD PTR SS:[EBP+409DDE]     ; kernel32.77E60000
00408F20   FF95 849C4000     CALL DWORD PTR SS:[EBP+409C84]
00408F26   40                INC EAX
```
Sada cemo polako ici sa F8 sve dok ne dodjemo do adrese:
```
00408F32   E8 0C000000       CALL demo_tEl.00408F43
```
U ovaj CALL moramo uci sa F7 jer ako pokusamo da ga predjemo sa F8 program ce se startovati i mi cemo izgubiti nas OEP. Prebacili smo se malo nize i sada se nalazimo ovde:
```
00408F43   FFB5 DE9D4000     PUSH DWORD PTR SS:[EBP+409DDE]   ; kernel32.77E60000
00408F49   FF95 849C4000     CALL DWORD PTR SS:[EBP+409C84]
00408F4F   40                INC EAX
00408F50   48                DEC EAX
```
Sa F8 cemo izvrsiti sve dok ne dodjemo do adrese:
```
00408F63   E8 11000000       CALL demo_tEl.00408F79
```
kada cemo pritisnuti F7 da udjemo u CALL, a onda cemo pritiskati F8 sve dok ne dodjemo do:
```
00408FF9   0000              ADD BYTE PTR DS:[EAX],AL
00408FFB   0000              ADD BYTE PTR DS:[EAX],AL
00408FFD   F3:AA             REP STOS BYTE PTR ES:[EDI]
00408FFF   66:AB             STOS WORD PTR ES:[EDI]          <- Ovde
00409001   EB 02             JMP SHORT demo_tEl.00409005
```
kada cemo pritisnuti F8 2x i pusticemo da se izvrsi skok 00409001 kao i sledeci JMP skok na adresi:
```
00409001   /EB 02            JMP SHORT demo_tEl.00409005
00409003   |CD 20            INT 20
00409005   \61               POPAD
00409006  - FF6424 D0        JMP DWORD PTR SS:[ESP-30]          ; demo_tEl.00401000
0040900A   98                CWDE
0040900B   F8                CLC
0040900C   73 02             JNB SHORT demo_tEl.00409010
```
Posle izvrsetka ovog skoka nalazimo se tacno ovde:
```
00401000    6A               DB 6A                    ;  CHAR 'j'
00401001    00               DB 00
00401002    68               DB 68                    ;  CHAR 'h'
00401003    1C               DB 1C
00401004    1040 00          ADC BYTE PTR DS:[EAX],AL
```
A to je verovali ili ne OEP. Ako pritisnete CTRL+A videcete i kako on izgleda. On izgleda bas ovako:
```
00401000   .  6A 00          PUSH 0                    ; /lParam = NULL
00401002   .  68 1C104000    PUSH demo_.0040101C ; |DlgProc = demo_tEl.0040101C
00401007   .  6A 00          PUSH 0                    ; |hOwner = NULL
00401009   .  6A 01          PUSH 1                    ; |pTemplate = 1
```
Sada mozemo uraditi dump (*uradite ga preko LordPEa jer Ollydump nece da uradi posao kako treba, ako vam KAV prijavi virus ignorisite ga jer virusa nema*) i popravicemo importe pomocu ImpReca (*OEP=00001000*). I to je to, otpakovan je i dobri stari tElock. Ako vam nesto nije jasno u vezi otpakivanja tElocka (a *sigurno vam nesto nije jasno*), tipa zasto smo usli u ovaj CALL a ne u ovaj. Odgovor je: Probao sam da predjem preko svih CALLova sa F8, kada bi se program startovao posle nekog CALLa zakljucio sam da u njega treba da udjem sa F7 i tako redom. Reversing je egzaktna nauka, ali morate pasti mnogo puta kako biste ustali, i stali cvrsto na svoje noge.

# tELock 0.96

Pre nego sto pocnemo sa otpakivanjem pomocu Ollyja moracemo da ga podesimo tako da stignemo do OEPa najbrze sto mozemo (bice to sa tri klika). Dakle otvorite Olly i idite u meni Debugging options (ALT + O), selektujte tab Exceptions i selektujte samo sledece checkboxove:

- *Ignore memory access violations in Kernel32*
- *INT3 breaks*
- *Single-step break*
- *Integer division by 0*

Zasto ovo radimo? Jednostavno tELock ce nam "bacati" ove exceptione pokusavajuci da detektuje da li je debugger aktivan. Pored ovoga je potrebno da imate ukljucen OllyInvisible ili HideOlly plugin. Kada ovo uradimo, mozemo da starujemo nasu metu (..\Cas10\CrackMe10.tElock 0.98b1.exe) pomocu Ollyja i da pocnemo sa otpakivanjem. Prve linije koje se nalaze na OEPu izgledaju ovako:

```
00407BD6 > $^\E9 25E4FFFF   JMP CrackMe1.00406000
00407BDB     00             DB 00
00407BDC     00             DB 00
00407BDD     00             DB 00
00407BDE     A8             DB A8
00407BDF     02             DB 02
00407BE0     D6             DB D6
```

Postoji vise nacina na koje mozemo doci do samog OEPa ali mi cemo to uraditi na najbrzi moguci nacin. Pritisnite F9 kao da pokusavate da startujete program, ali cete umesto startovanja programa zavrsiti ovde:

```
004066A8     8D             DB 8D
004066A9     C0             DB C0
004066AA     74             DB 74
```

Posto Olly nije stigao da analizira ovaj kod, pritisnite CTRL + A da biste videli sta se stvarno nalazi ovde:

```
004066A8   ? 8DC0           LEA EAX,EAX      ; Illegal use of register
004066AA     74             DB 74          ; CHAR 't'
004066AB     03             DB 03
004066AC   . CD 20          INT 20
004066AE   > 64:67:8F06 00> POP DWORD PTR FS:[0]
```

Kao sto vidite LEA EAX,EAX je ilegalna komanda i zato Olly ne zna sta da uradi ovde. Da bismo presli preko ove komande potrebno je da pritisnemo SHIFT + F9 posle cega cemo doci ovde:

```
00406BA6     CD             DB CD
00406BA7     68             DB 68          ; CHAR 'h'
00406BA8     66             DB 66          ; CHAR 'f'
00406BA9     05             DB 05
```

Posto je ovo access violation greska i ovde cemo pritisnuti SHIFT + F9 da bismo ponovo pokusali da startujemo program, ali cemo kao i prosli put zavrsiti na LEA EAX, EAX komandi ovde:

```
004076F1   ? 8DC0           LEA EAX,EAX      ; Illegal use of register
004076F3   ? EB 01          JMP SHORT CrackMe1.004076F6
```

Ako sledeci put pritisnemo SHIFT + F9 program ce se startovati! Posto ovo znamo (mozete to i da proverite) mozemo da postavimo jedan memorijski breakpoint na glavnu sekciju kako bismo presreli izvrsenje koda koji se nalazi u glavnoj sekciji. Samim presretanjem cemo se naci na OEPu. Otici cemo u Memory window (ALT + M) i postavicemo Memory breakpoint on access na

glavnu .code sekciju. Posle pritiska na SHIFT + F9 (da bismo presli preko LEA EAX,EAX) komande naci cemo se ovde:

```
0040104C    68              DB 68          ; CHAR 'h'
0040104D    481F4000        DD CrackMe1.00401F48
00401051    E8              DB E8
00401052    EE              DB EE
00401053    FF              DB FF
00401054    FF              DB FF
00401055    FF              DB FF
00401056    00              DB 00
00401057    00              DB 00
00401058    00              DB 00
00401059    00              DB 00
0040105A    00              DB 00
0040105B    00              DB 00
0040105C    30              DB 30          ; CHAR '0'
0040105D    00              DB 00
0040105E    00004000        DD CrackMe1.00400000
00401062    00              DB 00
```

Da li je ovo pravi OEP? Pritisnite CTRL + A da biste analizirali kod i videcete sledece:

```
0040104C  > /68 481F4000    PUSH CrackMe1.00401F48
00401051  . |E8 EEFFFFFF    CALL CrackMe1.00401044
00401056  . |0000           ADD BYTE PTR DS:[EAX],AL
00401058  . |0000           ADD BYTE PTR DS:[EAX],AL
0040105A  . |0000           ADD BYTE PTR DS:[EAX],AL
```

I kao sto je bilo i za ocekivati adresa 0040104C je pravi OEP. Ovde mozemo uraditi dump i popravku importa.

*Dump:* Otvorite LordPE i selektujte proces koji trenutno debuggujete. Selektujte CrackMe10.exe i uradite full dump desnim dugmetom.

*Importi:* Ostavljajuci Olly otvoren i ne pomerajuci se sa adrese 0040104C startujte ImpRec i u listi procesa ponovo selektujte CrackMe10.exe. Posle ovoga potrebno je da ispunite sledece polje. U OEP polje unesite 0040104C - 00400000 = 104C i pritisnite IAT AutoSearch i Get Imports dugme. Posto su svi pronadjeni importi validni mozemo da popravimo prethodno dumpovan fajl pomocu LordPEa. Pritisnite Fix dump i selektujte snimljeni fajl posle cega ce tELock 0.98b1 biti uspesno otpakovan!

## tELock 0.98b1

Posto smo vec konfigurisali Olly kako treba i posto smo u proslom delu ovog poglavlja vec naucili kako se dolazi do OEPa, to mozemo da iskoristimo i ovde. Otvoricemo Olly i nasu drugu metu pomocu njega. Bez neke velike mudrosti dolazimo do OEPa na adresi 00401000 i tu cemo uraditi dump. Sam OEP izgleda ovako:

```
00401000  > /68 E5314000    PUSH decryptm.004031E5
00401005  . |68 73314000    PUSH decryptm.00403173
0040100A  . |68 23314000    PUSH decryptm.00403123
0040100F  . |68 D3304000    PUSH decryptm.004030D3
```

Mozemo da pokusamo da popravimo importe pomocu ImpReca ali cemo umesto validnih importa pronaci 5 importa koje ImpRec ne moze da prepozna. Ove importe cemo popraviti tako sto cemo snimiti ovo drvo

pomocu klika na Save Tree dugme. Kada snimimo ovaj fajl otvoricemo ga pomocu notepada i videcemo sledece:

```
Target: decryptme1.tElock0.96.exe
OEP: 00001000 IATRVA: 00002000      IATSize: 00000014

FThunk: 00002000      NbFunc: 00000005
0       00002000      ?       0000    009E0000
0       00002004      ?       0000    009E001D
0       00002008      ?       0000    009E003A
0       0000200C      ?       0000    00850000
0       00002010      ?       0000    0085001D
```

Sada cemo traceovati kroz nasu metu pomocu Ollyja. Interesuju nas svi CALLovi ka API funkcijama. Dakle uci cemo u sledeci CALL sa F7:

```
00401028  .  E8 23060000      CALL decryptm.00401650
```

I nacicemo se ovde:

```
00401650  $- FF25 10204000 JMP NEAR DWORD PTR DS:[402010] <- Ovde smo
00401656  $- FF25 0C204000 JMP NEAR DWORD PTR DS:[40200C]
0040165C  $- FF25 04204000 JMP NEAR DWORD PTR DS:[402004]
00401662  $- FF25 00204000 JMP NEAR DWORD PTR DS:[402000]
```

Kao sto vidimo ovaj niz skokova je lista skokova ka API pozivima, odnosno ka njihovoj redirekciji. Sada moramo da povezemo svaku ovu adresu sa jednim APIjem. Ovo cemo uraditi tako sto cemo pratiti svaki ovaj JMP skok sa F7. U prvom slucaju cemo se posle klika na F7 naci ovde:

```
0085001D   F9                STC
0085001E   72 01             JB SHORT 00850021
00850020   0C EB             OR AL,0EB
00850022   02B8 663D8000     ADD BH,BYTE PTR DS:[EAX+803D66]
00850028   FF20              JMP NEAR DWORD PTR DS:[EAX]
0085002A   B8 D64AA000       MOV EAX,0A04AD6
0085002F   F8                CLC
00850030  ^ 72 EB            JB SHORT 0085001D
00850032   B8 46008500       MOV EAX,850046
00850037   FF20              JMP NEAR DWORD PTR DS:[EAX]
00850039   90                NOP
0085003A   B8 20000000       MOV EAX,20
0085003F   0000              ADD BYTE PTR DS:[EAX],AL
00850041   0076 64           ADD BYTE PTR DS:[ESI+64],DH
00850044   D6                SALC
00850045  ^ 77 E3            JA SHORT 0085002A
00850047   A6                CMPS BYTE PTR DS:[ESI],BYTE PTR ES:[EDI]
00850048   D4 77             AAM 77
```

Sada nam ostaje da ispratimo ovaj kod za redirekciju sa F8. Tokom ovoga cemo naici na sledece:

```
00850037  - FF20              JMP NEAR DWORD PTR DS:[EAX];user32.wsprintfA
```

Kada vratimo iz ovog skoka ka user32.wsprintfA APIu naci cemo se ovde:

```
0040102D  .  83C4 20          ADD ESP,20
00401030  .  68 36324000      PUSH decryptm.00403236
00401035  .  E8 28060000      CALL decryptm.00401662
```

Uci cemo i u sledeci CALL sa F7 i naci cemo se ovde:

```
00401662  $- FF25 00204000  JMP NEAR DWORD PTR DS:[402000]
```

Ponovnim pritiskom na F7 dolazimo opet do API redirekcije. Ovde cemo ponoviti sve sto smo radili i prilikom pronalaska wsprintf APIja samo sto cemo ovaj put pronaci sledece:

```
009E001A  - FF20              JMP NEAR DWORD PTR DS:[EAX];kernel32.lstrlenA
```

Posle izlaska iz kernel32.dll fajla naci cemo se ovde:

```
0040103A  .  33FF             XOR EDI,EDI
0040103C  >  3BC7             CMP EAX,EDI
```

Sada cemo traziti kroz kod sve dok ne dodjemo do nekog sledeceg CALLa ka nekoj API funkciji. To ce se desiti ovde:

```
00401635   .  6A 40            PUSH 40
00401637   .  68 00304000      PUSH decryptm.00403000
0040163C   .  68 36324000      PUSH decryptm.00403236
00401641   .  6A 00            PUSH 0
00401643   .  E8 0E000000      CALL decryptm.00401656
```

Ovde cemo ponovo biti prinudjeni da udjemo u taj CALL sa F7 i doci cemo ovde:

```
00401656   $- FF25 0C204000   JMP NEAR DWORD PTR DS:[40200C]
```

Klikom na F7 i analizom koda zakljucujemo da je ovo skok ka APIu:

```
0085001A  - FF20              JMP NEAR DWORD PTR DS:[EAX];user32.MessageBoxA
```

Posle izvrsenja ovog CALLa ka MessageBoxA API i naravno posle njegovog zatvaranja i vracanja iz kernel32.dll fajla naci cemo se ovde:

```
00401648   .  6A 00            PUSH 0
0040164A   .  E8 0D000000      CALL decryptm.0040165C
```

Naravno i ovo je poziv ka nekom APIju. Na isti nacin kao i do sada dolazimo do zakljucka da se radi o:

```
0040165C   $- FF25 04204000   JMP NEAR DWORD PTR DS:[402004]
...
009E0037  - FF20              JMP NEAR DWORD PTR DS:[EAX];kernel32.ExitProcess
```

kernel32.ExitProcess APIu.

Sada imamo sve podatke koji su nam potrebni za rekreiranje IAT.txt fajla. Njegova struktura je jedinstvena i potrebna je potpuna modifikacija do sada snimljenog fajla u sledecu strukturu:

```
Target: decryptme1.tElock0.96.exe
OEP: 00001000 IATRVA: 00002000      IATSize: 00000014

FThunk: 00002000      NbFunc: 00000002
1       00002000      kernel32.dll    0000    lstrlenA
1       00002004      kernel32.dll    0000    ExitProcess

FThunk: 0000200C      NbFunc: 00000002
1       0000200C      user32.dll      0000    MessageBoxA
1       00002010      user32.dll      0000    wsprintfA
```

Da pojasnim. Prvi red se ispunjava tako sto za FThunk vrednost stavlja poziv ka prvom APIju iz nekog .dll fajla. Posto smo mi odlucili da to bude 00002000 (RVA 00402000) onda taj API mora odgovarati APIju koji se dobija skokom sa JMP NEAR DWORD PTR DS:[402000], to jest mora odgovarati lstrlenA APIju u kernel32.dll fajlu. Ali ovo je vec sadrzaj drugog reda u kojem postoje dve konstante. Jedan na pocetku reda i 0000 koje se odnose na ordinalni broj importa, ali ovaj broj cemo podesiti pomocu ImpReca. Ovo treba ponoviti za sve .dll fajlove koje nasa meta ponavlja. Vec uradjena IAT struktura se nalazi u istoj arhivi kao i nasa meta. Kada konacno konacno snimite ovaj fajl otvorite ga pomocu Load Tree opcije u ImpRecu i duplim klikom na svaki od ovih importa cete konacno popraviti sve ordinale. Kada konacno i ovo zavrsite moci cete da uradite Fix dump i da vidite da je i tELock 0.98 uspesno otpakovan.

# PeCompact 2.22

Moram da priznam da sam svojevremeno mislio da je PeCompact zastita bolja nego sto jeste. Sada sam konacno odlucio da se posvetim rucnom otpakivanju aplikacije zasticene PeCompactom 2.22. Posto sam sa interneta ranije skinuo sam trial PeCompact pakera primetio sam da se mogu birati moduli kojima ce se pakovati exe fajlovi. Default podesavalje ukljucuje samo jedan dll i on se zove pecompact_ffss.dll ali se meta pakovana PeCompactom otpakuje isto kao da imamo i ukljucen dll pecompact_crc32.dll fajl. Meta za ovaj deo knjige je pakovana sa trial verzijom PeCompacta mada licno mislim da nema veze da li je trial ili ne, moram ovo da naglasim ako razlike ipak postoje. Sama meta se zove crackme.pecompact22.exe a nalazi se u folderu Cas10. Ovu metu cemo otvoriti pomocu Ollyja, i na OEPu cemo imati sledece:

```
004012C0 > $  B8 D07D4000    MOV EAX,crackme_.00407DD0
004012C5   .  50             PUSH EAX
004012C6   .  64:FF35 00000> PUSH DWORD PTR FS:[0]
004012CD   .  64:8925 00000> MOV DWORD PTR FS:[0],ESP
004012D4   .  33C0           XOR EAX,EAX
004012D6   .  8908           MOV DWORD PTR DS:[EAX],ECX
```

Ovaj "OEP" izgleda dosta cudno. Metu cemo otpakovati na nacin na koji sam je ja otpakovao. Preci cemo deo OEPa sa F8 sve dok ne dodjemo do adrese 004012D6 kada ce Olly prikazati Access violation. Ovo cemo iskoristiti u nasu korist. Pritisnucemo CTRL+F7 kako bismo stigli do reda iznad kojeg se desio Access violation. To se desilo upravo ovde:

```
77FB4DB3  |.  8B1C24         MOV EBX,DWORD PTR SS:[ESP]
77FB4DB6  |.  51             PUSH ECX
77FB4DB7  |.  53             PUSH EBX
```

Vidimo da se ne nalazimo u kodu crackmea nego se nalazimo u kodu ntdll.dll fajla. Sa F8 cemo se kretati kroz kod sve dok ne dodjemo do ove adrese:

```
77FB4DC6  |.  E8 480BFCFF    CALL ntdll.ZwContinue
```

tada cemo pritisnuti F7 da udjemo u ovaj CALL jer ce nas to vratiti na mesto iznad kojeg se desio Access violation. Sada se nalazimo ovde

```
77F75913 >/$  B8 20000000    MOV EAX,20
77F75918  |.  BA 0003FE7F    MOV EDX,7FFE0300
77F7591D  |.  FFD2           CALL EDX
77F7591F  \.  C2 0800        RET 8
```

ali smo i dalje u ntdll.dll fajlu, pa cemo zato sa F8 izvrsiti sve komande do ukljucujuci i RET 8 komandu. Tada cemo se naci ovde:

```
00407DF3    B8 796D40F0      MOV EAX,F0406D79
00407DF8    64:8F05 0000000>POP DWORD PTR FS:[0]
```

ovo je vec deo koda PeCompacta koji sluzi za otpakivanje fajla u memoriju. Ovaj deo koda cemo izvrsiti sa F8 sve dok ne dodjemo do adrese

```
00407E94    5D               POP EBP
00407E95  - FFE0             JMP EAX                 ; crackme_.<ModuleEntryPoint>
```

i kao sto vidimo JMP EAX vodi do ModuleEntryPoint-a to jest do samog OEPa pa cemo i ovaj skok izvrsiti sa F8 i naci cemo se na OEPu, koji posle analize sa CTRL+A izgleda ovako:

```
004012C0 >/$  55             PUSH EBP
004012C1  |.  8BEC           MOV EBP,ESP
004012C3  |.  6A FF          PUSH -1
```

Sada ostaje samo da dumpujemo memoriju sa LordPEom i da popravimo importe sa ImpRecom, posle cega smo uspesno otpakovali PeCompact.

# PeCompact 1.40

Sada sam 100% siguran da se pitate: "Zasto je napisao prvo kako se otpakuje PeCompact 2.22 a tek onda kako PeCompact 1.40"? Naravno imam ja svoje razloge zasto je to ovako uradjeno. Jedan od glavnih razloga je taj sto je mnogo lakse otpakovati novu 2.22 verziju nego staru 1.40. Ne verujete mi? Videcemo :) Meta koja je pakovana sa 1.40 verzijom PeCompacta se zove Notepad.pecompact140.exe. Pre nego sto otvorimo metu pomocu Ollyja prvo cu vam pokazati jedan jako interesantan trik sa PeIDom. Naime PeID ima jedan veoma interesantan plugin koji nam omogucava da unapred odredimo OEP mete. Dakle otvoricemo metu pomocu PeIDa koji ce nam prvo reci da je meta pakovana sa: PECompact 1.40 - 1.45 -> Jeremy Collake a ako kliknemo na dugme -> koje se nalazi u donjem desnom uglu i izaberemo Plugins -> Generic OEP finder, posle cega ce PeID prikazati poruku da se OEP nalazi na adresi 004010CC. Posle ovoga mozemo da zatvorimo PeID i da ucitamo metu u OllyDBG.
Pakerov OEP izgleda ovako:

```
0040AC5E > /EB 06          JMP SHORT Notepad_.0040AC66
0040AC60  |68 CC100000     PUSH 10CC
0040AC65  |C3              RET
0040AC66  \9C              PUSHFD
```

Sada vidite razliku izmedju verzija. Btw da li ste primetili parametar koji ima PUSH komanda? Da, da znam PUSH 10CC, to je PUSH OEP. Bez obzira na ovo, pritiskacemo F8 sve dok ne dodjemo do CALLa na adresi 0040AC68 u koji moramo da udjemo. Taj CALL izgleda bas ovako:

```
0040AC68   E8 02000000     CALL Notepad_.0040AC6F
```

stoga cemo pritisnuti F7 da udjemo u ovaj CALL kada dodjemo do njega, to nas vodi ovde:

```
0040AC6F   8BC4            MOV EAX,ESP
0040AC71   83C0 04         ADD EAX,4
0040AC74   93              XCHG EAX,EBX
0040AC75   8BE3            MOV ESP,EBX
```

Posto se dole nalazi jedna RET komanda a izmedju nema nikakvih skokova izvrsicemo sve komande sa F8, ukljucujuci i RET komandu, posle cega smo ovde:

```
0040D1C3   BD 53310000     MOV EBP,3153
0040D1C8   57              PUSH EDI
0040D1C9   5E              POP ESI
0040D1CA   83C6 42         ADD ESI,42
0040D1CD   81C7 53110000   ADD EDI,1153
0040D1D3   56              PUSH ESI
```

dalje i nema neke velike mudrosti jednostavno drzimo F8 sve dok ne dodjemo do adrese:

```
0040D350   57              PUSH EDI
0040D351   8BBD D7A44000   MOV EDI,DWORD PTR SS:[EBP+40A4D7]
0040D357   03BD A6A04000   ADD EDI,DWORD PTR SS:[EBP+40A0A6]
0040D35D   8B8D DBA44000   MOV ECX,DWORD PTR SS:[EBP+40A4DB]
```

malo ispod toga imamo dosta loopova koji se izvrsavaju dosta puta, ti loopovi izgledaju ovako:

```
0040D36E   /74 72          JE SHORT Notepad_.0040D3E2
0040D370   |78 70          JS SHORT Notepad_.0040D3E2
0040D372   |66:8B07        MOV AX,WORD PTR DS:[EDI]
0040D375   |2C E8          SUB AL,0E8
0040D377   |3C 01          CMP AL,1
```

```
0040D379  |76 38           JBE SHORT Notepad_.0040D3B3
0040D37B  |66:3D 1725      CMP AX,2517
0040D37F  |74 51           JE SHORT Notepad_.0040D3D2
0040D381  |3C 27           CMP AL,27
0040D383  |75 0A           JNZ SHORT Notepad_.0040D38F
0040D385  |80FC 80         CMP AH,80
0040D388  |72 05           JB SHORT Notepad_.0040D38F
0040D38A  |80FC 8F         CMP AH,8F
0040D38D  |76 05           JBE SHORT Notepad_.0040D394
0040D38F  |47              INC EDI
0040D390  |43              INC EBX
0040D391  ^|EB DA          JMP SHORT Notepad_.0040D36D
```

Da bismo presli sva ova ponavljanja pronaci cemo negde dole zadnji skok koji ce nas vratiti negde gore u kod. Taj skok se nalazi ovde:

```
0040D3E0  ^\EB 87          JMP SHORT Notepad_.0040D369
```

Posto zelimo da preskocimo sve ove loopove jednostavno cemo postaviti breakpoint na adresu odmah ispod ovog skoka, postavicemo ga na adresu 0040D3E2 i pritisnucemo F9 da dodjemo do njega. Dalje cemo nastaviti sa izvrsavanjem koda sa F8 sve dok ne stignemo do adrese 0040D48F.

```
0040D488   9D              POPFD
0040D489   50              PUSH EAX
0040D48A   68 CC104000     PUSH Notepad_.004010CC
0040D48F   C2 0400         RET 4
```

Kada se izvrsi i ova RET 4 komanda mi cemo se naci ovde:

```
004010CC    55             DB 55                ; CHAR 'U'
004010CD    8B             DB 8B
004010CE    EC             DB EC
```

Kao sto nam je PeID rekao na adresi 004010CC se nalazi pravi OEP. Posle analize ovog koda sa CTRL+A cemo videti da je ovo zaista pravi OEP,

```
004010CC /.  55            PUSH EBP
004010CD |.  8BEC          MOV EBP,ESP
004010CF |.  83EC 44       SUB ESP,44
004010D2 |.  56            PUSH ESI
```

i da ovde mozemo da uradimo dump pomocu LordPEa. Posle dumpa nam ostaje samo da popravimo importe pomocu ImpReca na standardan nacin.

Videli smo da PeID bez greske moze da nadje OEP u PeCompactovanim fajlovima i da se bez brige mozemo pouzdati u njegovu procenu OEPa. Ova informacija nam moze posluziti da napravimo neku skriptu koja bi za nas otpakovala program pakovan PeCompactom ili je mozemo koristiti da se proverima da li smo mi na pravom mestu i da li je to pravi OEP. Kao sto vidite mnogo je lakse otpakovati noviju nego stariju verziju PeCompacta, mislim lako je otpakovati i jednu i drugu verziju samo nam za ovu novu verziju treba manje posla.

# PE Pack 1.0

Prelistavajuci neke tudje keygeneratore koje imam na svom hard disku naisao sam na jedan zanimljiv keygen za CDRLabel 4.1 koji su napravili momci iz CORE crackerske grupe. Meni licno sam keygen nije bio zanimljiv nego mi je bila zanimljiva zastita kojom je pakovan ovaj keygen. U pitanju je PE Pack koji nikada ranije nisam rucno otpakovao pa sam odlucio da se oprobam i sa ovim pakerom. Molim vas da imate na umu da mene, a ne bi trebalo ni vas, ne interesuje sta ova meta radi, nego me interesuje kako bih ja to mogao da je otpakujem. Meta se nalazi u folderu Cas10 a zove se crcdl41.pepack10.exe. Ovu metu cemo otvoriti pomocu Ollyja i pogledacemo sta se nalazi na OEPu.

```
00401212 > $ /74 00          JE SHORT cr-cdl41.00401214
00401214   >-\E9 E74D0000    JMP cr-cdl41.00406000
```

Dosta cudno, ali nema veze, sa F8 cemo izvrsiti oba skoka i naci cemo se ovde:

```
00406000   60                PUSHAD
00406001   E8 00000000       CALL cr-cdl41.00406006
00406006   5D                POP EBP
00406007   83ED 06           SUB EBP,6
```

Sa F8 cemo izvrsavati red po red sve dok ne dodjemo do jednog dugackog loopa:

```
004060D4   /73 38            JNB SHORT cr-cdl41.0040610E
004060D6   |48               DEC EAX
004060D7   |74 35            JE SHORT cr-cdl41.0040610E
004060D9   |78 33            JS SHORT cr-cdl41.0040610E
004060DB   |66:8B1C39        MOV BX,WORD PTR DS:[ECX+EDI]
004060DF   |80FB E8          CMP BL,0E8
004060E2   |74 0F            JE SHORT cr-cdl41.004060F3
004060E4   |80FB E9          CMP BL,0E9
004060E7   |74 0A            JE SHORT cr-cdl41.004060F3
004060E9   |66:81FB FF25     CMP BX,25FF
004060EE   |74 0F            JE SHORT cr-cdl41.004060FF
004060F0   |41               INC ECX
004060F1  ^|EB E3            JMP SHORT cr-cdl41.004060D6
004060F3   |294C39 01        SUB DWORD PTR DS:[ECX+EDI+1],ECX
004060F7   |83C1 05          ADD ECX,5
004060FA   |83E8 04          SUB EAX,4
004060FD  ^|EB D7            JMP SHORT cr-cdl41.004060D6
004060FF   |295439 02        SUB DWORD PTR DS:[ECX+EDI+2],EDX
00406103   |83C1 06          ADD ECX,6
00406106   |83EA 04          SUB EDX,4
00406109   |83E8 05          SUB EAX,5
0040610C  ^|EB C8            JMP SHORT cr-cdl41.004060D6
0040610E   \C685 D3000000 F>MOV BYTE PTR SS:[EBP+D3],0F8
00406115   5B                POP EBX
00406116   5A                POP EDX
00406117   5E                POP ESI
00406118  ^ E9 76FFFFFF      JMP cr-cdl41.00406093
0040611D   6A 04             PUSH 4
```

Da ne bismo izvrsavali ovaj dugacak loop postavicemo breakpoint na adresu 0040611D, odnosno na PUSH 4 komandu jer cemo tu sigurno stici odmah posle izvrsavanja ovog loopa to jest posle otpakivanja u memoriju. Sa F8 cemo polako izvrsavati kod sve dok ne dodjemo do adrese:

```
004061C2   0385 47050000     ADD EAX,DWORD PTR SS:[EBP+547]
004061C8   52                PUSH EDX                    ; cr-cdl41.00404000
```

kada ce se u EAXu pojaviti user32.dll string. Znaci da je na red doslo otpakivanje IATa u memoriju pa cemo potraziti loop koji ce to uraditi. On se nalazi ovde:

```
00406252   83C7 04          ADD EDI,4
00406255  ^ EB BD           JMP SHORT cr-cdl41.00406214
```

to jest ovo je poslednja adresa loopa koji sluzi za otpakivanje API poziva u memoriju, dok se odmah ispod nalazi drugi skok:

```
00406257   83C2 14          ADD EDX,14
0040625A  ^ E9 55FFFFFF     JMP cr-cdl41.004061B4
```

koji sluzi za otpakivanje imena svih dll fajlova koji sadrze otpakovane API pozive. Zbog ovoga cemo postaviti dva breakpointa na adrese:

```
004061C2   0385 47050000    ADD EAX,DWORD PTR SS:[EBP+547]
004061C8   52               PUSH EDX                    ; cr-cdl41.00404014
```

da bismo videli imena dll fajlova koja se nalaze u IATu. Treba nam jos jedan breakpoint za slucaj da pritisnemo F9 jednom vise puta nego sto treba. Zbog ovoga cemo postaviti breakpoint na adresu:

```
0040625F   8B85 57050000    MOV EAX,DWORD PTR SS:[EBP+557] <- Ovde
00406265   0385 8B050000    ADD EAX,DWORD PTR SS:[EBP+58B]
0040626B   894424 1C        MOV DWORD PTR SS:[ESP+1C],EAX
0040626F   61               POPAD
00406270   FFE0             JMP EAX
```

jer ce se posle ova dva loopa nastaviti sa izvrsavanjem unpacking koda. Sada cemo pritiskati F9 i belezicemo imena dll fajlova koja ce se pojavljivati u EAXu na adresi 004061C8. Ti dll fajlovi su:

```
user32.dll
msvcrt.dll
kernel32.dll
```

i kada sledeci put pritisnemo F9 zastacemo na adresi 0040625F koja predstavlja nas safe breakpoint. Ovo znaci da se API pozivi nalaze samo u tri dll fajla. Ovo smo mogli da uradimo i sa imenima API poziva, ali za to nema potrebe. Bitni su nam samo dll fajlovi posle kada budemo popravljali IAT pomocu ImpReca. Sa F8 cemo izvrsiti par redova sve dok ne dodjemo do adrese:

```
0040626F   61               POPAD
00406270  - FFE0            JMP EAX                     ; cr-cdl41.004010D0
```

Navikli smo da se na OEP ide pomocu skokova i RET komandi, tako da skok koji se izvrsava odmah posle POPAD funkcije moze voditi pravo na OEP. Izvrsicemo i njega sa F8 i videcemo gde ce nas to odvesti. Doci cemo ovde:

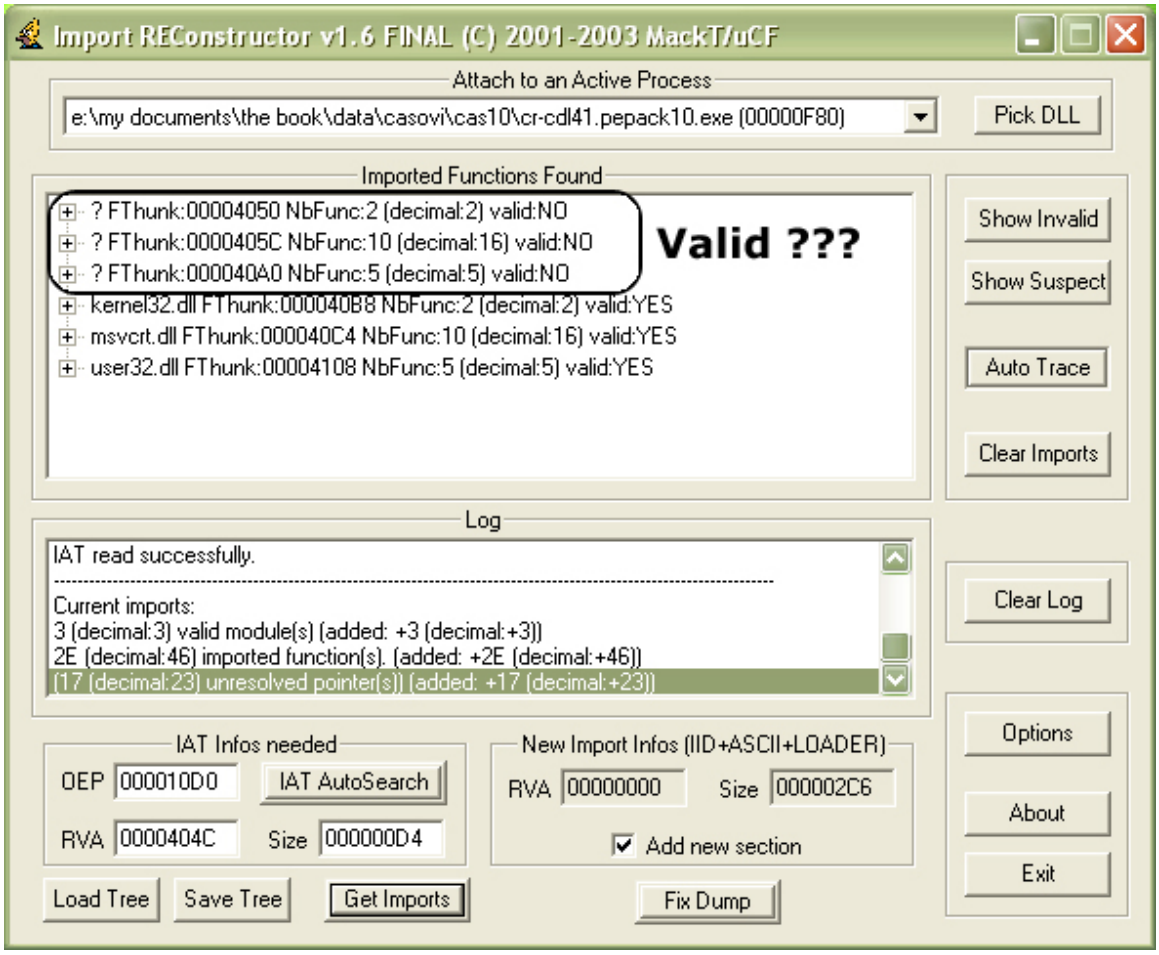
```
004010D0    55              DB 55                       ;  CHAR 'U'
004010D1    8B              DB 8B
004010D2    EC              DB EC
004010D3    6A              DB 6A                       ;  CHAR 'j'
004010D4    FF              DB FF
```

Ovo naravno lici na OEP i posle analize koda pritiskom na CTRL+F2 i zakljucujemo da je ovo pravi OEP jer izgleda bas ovako:

```
004010D0  . 55              PUSH EBP
004010D1  . 8BEC            MOV EBP,ESP
004010D3  . 6A FF           PUSH -1
004010D5  . 68 00204000     PUSH cr-cdl41.00402000
```

Dakle adresa na kojoj mozemo da izvrsimo memory dump je 004010D0. Dump cemo uraditi na standardan nacin pomocu LordPEa a rekonstrukciju importa cemo uraditi pomocu ImpReca. Sada ce na scenu stupiti ono sto je bilo potrebno da zapisemo. Sada ce nam trebati imena dll fajlova iz kojih se citaju importi. Ovo smo uradili zato sto ce se desiti da ImpRec nece moci da nadje sve dll fajlove iz kojih se citaju API pozivi pa cemo mi morati rucno da

modifikujemo i / ili obrisemo neispravne API pozive. Ne brinite ovo ce biti izuzetno lako posto znamo imena dll fajlova iz kojih se citaju API pozivi. Otvoricemo ImpRec i ucitacemo proces cr-cdl41.pepack10.exe u njega. Promenicemo OEP na pravu vrednost, promenicemo ga na 000010D0, pritisnucemo IAT AutoSearch i dobicemo importe kao na slici ispod.




Ono sto je bitno da se vidi je da se svi dll fajlovi koje smo pronasli prilikom otpakivanja mete vec uneti i da je ovaj visak koji je na slici zaokruzen stvarno visak i da se moze odseci iz nove IAT tablice. Ovo cemo uraditi klikom na dugme Show Invalid, pa cemo kliknuti desnim dugmetom na selektovane API pozive i izabracemo opciju Cut thunk(s). Kada ostanu samo ispravni API pozivi iz ona tri pronadjena dll fajla moci cemo da uradimo popravku importa u dumpovanom fajlu pomocu opcije Fix Dump. I to je to uspeli smo uspesno da otpakujemo i PE Pack.

# ASProtect 1.2 / 1.2c

Dugo vremena sam bio plasen od strane drugih (doduse manje iskusnih) crackera kako je najvece zlo koje stoji odmah uz Armadillo, protektor pod imenom ASProtect. Danas ja pokusavam da dokazem suprotno. Iako je verzija koju cemo mi reversovati pomalo zastarela, jer je u opticaju verzija 1.3x, i dalje je dobar primer kako se to reversuje ASProtect. Ne shvatite ovaj deo knjige olako, jer i dalje postoji mnogo komercijalnih programa pakovanih bas ovom verzijom programa. Postoje neke prednosti kod reversinga ove verzije ASProtecta, a to su: nema previse teskih importa, nema "ukradenih bajtova" i sto je najlepse od svega ima exceptiona koji nam pomazu da pronadjemo pravi OEP. Meta se zove crackme.asprotect.exe a nalazi se u folderu Cas10. Ucitacemo je u Olly i na pakerovom OEPu cemo imati sledece:

```
00401000 >/$  68 01704000    PUSH crackme_.00407001
00401005  \.  C3             RET
```

Zanimljivo ili ne, ali mi se necemo baktati sa ovim. Kao sto sam rekao Olly ce nam pomoci preko exceptiona da nadjemo OEP. Pritisnite F9 i naci cete se ovde:

```
0087009D   3100              XOR DWORD PTR DS:[EAX],EAX
0087009F   EB 01             JMP SHORT 008700A2
008700A1   68 648F0500       PUSH 58F64
008700A6   0000              ADD BYTE PTR DS:[EAX],AL
```

Sada cemo pritiskati CTRL+F9 da bismo presli preko ovih exceptiona. Ovo cemo uraditi onoliko puta koliko je potrebno da se sama zapakovana meta ne startuje. To jest doci cemo do dela koda gde kada bi pritisnuli CTRL+F9 jos jednom NAG ekran iz crackmea bi se pojavio. Neki to rade brojanjem ali ovde nema potrebe za brojanjem jer je to mesto isto za sve programe i izgleda ovako:

```
00882FCC   FE02              INC BYTE PTR DS:[EDX]
00882FCE  ^ EB E8            JMP SHORT 00882FB8
00882FD0   E8 0A000000       CALL 00882FDF
```

Znaci pritiskacemo CTRL+F9 sve dok ne dodjemo do ovog mesta u kodu. Sada cemo umesto CTRL+F9 pritisnuti CTRL+F8 da bismo presli preko ovog exceptiona za jednu liniju koda i kao poruceno zavrsavamo ovde:

```
77FB4DB3  |.  8B1C24         MOV EBX,DWORD PTR SS:[ESP]
77FB4DB6  |.  51             PUSH ECX
77FB4DB7  |.  53             PUSH EBX
77FB4DB8  |.  E8 ACBDFAFF    CALL ntdll.77F60B69
77FB4DBD  |.  0AC0           OR AL,AL
```

Da, da zahvaljujuci Windowsu XP zavrsavamo u ntdll.dll-u. Ovo je dobra stvar jer cemo odatle izaci tacno ispod exceptiona. Pritiskajte F8 sve dok ne dodjete do prvog CALLa, a onda pritisnite F7 da udjete u njega. I dalje smo u istom dll-u pa cemo se kretati sa F8 sve dok ne dodjemo do ovog CALLa:

```
77F60BEA   .  FF75 0C        PUSH DWORD PTR SS:[EBP+C]
77F60BED   .  53             PUSH EBX
77F60BEE   .  56             PUSH ESI
77F60BEF   .  E8 528F0100    CALL ntdll.77F79B46              <- Ucemo u ovaj CALL (F7)
77F60BF4   .  F605 4A32FC77> TEST BYTE PTR DS:[77FC324A],80
77F60BFB   .  8BF8           MOV EDI,EAX
77F60BFD   .  0F85 896F0200  JNZ ntdll.77F87B8C
```

u koji cemo uci sa F7. A to nas vodi dublje u sam dll fajl. Sada smo ovde:

```
77F79B46   $  BA B89BF777    MOV EDX,ntdll.77F79BB8
77F79B4B   .  EB 07          JMP SHORT ntdll.77F79B54
```

```
77F79B4D  /$  BA DF9BF777    MOV EDX,ntdll.77F79BDF
77F79B52  |.  8D09           LEA ECX,DWORD PTR DS:[ECX]
77F79B54  |>  53             PUSH EBX
77F79B55  |.  56             PUSH ESI
77F79B56  |.  57             PUSH EDI
77F79B57  |.  33C0           XOR EAX,EAX
77F79B59  |.  33DB           XOR EBX,EBX
77F79B5B  |.  33F6           XOR ESI,ESI
77F79B5D  |.  33FF           XOR EDI,EDI
77F79B5F  |.  FF7424 20      PUSH DWORD PTR SS:[ESP+20]          ; /Arg5
77F79B63  |.  FF7424 20      PUSH DWORD PTR SS:[ESP+20]          ; |Arg4
77F79B67  |.  FF7424 20      PUSH DWORD PTR SS:[ESP+20]          ; |Arg3
77F79B6B  |.  FF7424 20      PUSH DWORD PTR SS:[ESP+20]          ; |Arg2
77F79B6F  |.  FF7424 20      PUSH DWORD PTR SS:[ESP+20]          ; |Arg1
77F79B73  |.  E8 06000000    CALL ntdll.77F79B7E                 ; \ntdll.77F79B7E
77F79B78  |.  5F             POP EDI
77F79B79  |.  5E             POP ESI
77F79B7A  |.  5B             POP EBX
77F79B7B  \.  C2 1400        RET 14
....
77F79BA2  |.  FFD1           CALL ECX
77F79BA4  |.  64:8B25 00000> MOV ESP,DWORD PTR FS:[0]
77F79BAB  |.  64:8F05 00000> POP DWORD PTR FS:[0]
77F79BB2  |.  8BE5           MOV ESP,EBP
77F79BB4  |.  5D             POP EBP
77F79BB5  \.  C2 1400        RET 14
```

Sada cemo se kretati sa F8 preko svih ostalih ASM komandi sem CALLova u koje cemo ulaziti sa F7. Zadnji CALL u koji cemo ovde uci je CALL ECX posle cega stizemo ovde:

```
00882FBE   8B6424 08        MOV ESP,DWORD PTR SS:[ESP+8]
00882FC2   EB 0C            JMP SHORT 00882FD0
00882FC4   2BD2             SUB EDX,EDX
00882FC6   64:FF32          PUSH DWORD PTR FS:[EDX]
```

ovde se ne desava nista bitno po otpakivanje pa cemo sa F8 ici kroz kod sve dok ne dodjemo do skoka koji bi nas vratio daleko gore u kod. Taj skok se nalazi ovde:

```
0088307D   /E3 03           JECXZ SHORT 00883082
0088307F   |59              POP ECX
00883080  ^|EB C8           JMP SHORT 0088304A              <- Ovaj skok
00883082   \59              POP ECX
00883083   1BC3             SBB EAX,EBX
00883085   F8               CLC
00883086   61               POPAD
00883087   F8               CLC
00883088   03C6             ADD EAX,ESI
0088308A   C3               RET
```

Zato cemo postaviti breakpoint na prvu komandu ispod ovog skoka, na POP ECX, i pritisnucemo F9 da dodjemo do nje. Doci cemo do prve sledece RET komande pritiskajuci F8 i izvrsicemo i nju sa F8 da bismo se nasli ovde:

```
00882F9E   5B               POP EBX                         ; crackme_.00400000
00882F9F   58               POP EAX
00882FA0   05 6618F45A      ADD EAX,5AF41866
00882FA5   5C               POP ESP
00882FA6   0BC9             OR ECX,ECX
00882FA8  ^ 74 E3           E SHORT 00882F8D
00882FAA   8901             MOV DWORD PTR DS:[ECX],EAX
00882FAC   03C3             ADD EAX,EBX
00882FAE   894424 1C        MOV DWORD PTR SS:[ESP+1C],EAX
00882FB2   61               POPAD
00882FB3   FFE0             JMP EAX
```

Da li i vama ona komanda JMP EAX izgleda zanimljivo :) Sa F8 cemo izvrsiti sve komande do skoka pa i sam skok, JMP EAX i naci cemo se ovde:

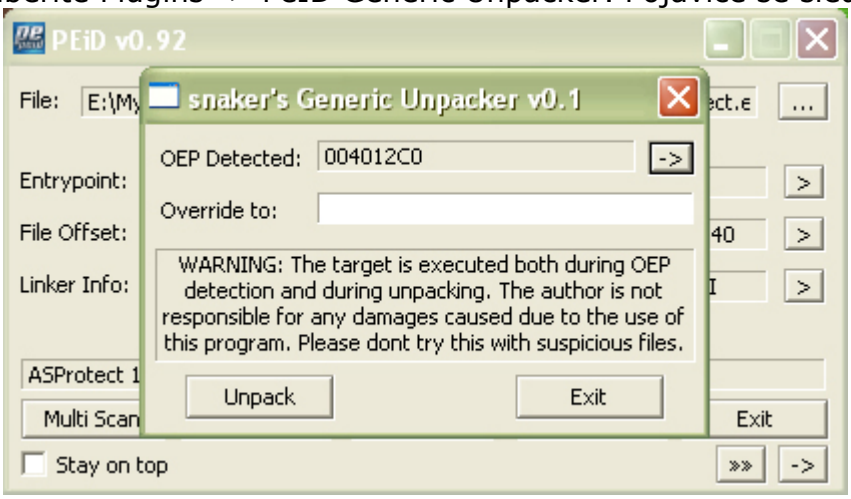
```
004012C0    55              DB 55                   ; CHAR 'U'
004012C1    8B              DB 8B
004012C2    EC              DB EC
004012C3    6A              DB 6A                   ; CHAR 'j'
004012C4    FF              DB FF
004012C5    68              DB 68                   ; CHAR 'h'
```

Verovali ili ne ovo je nas pravi OEP. Pritisnite CTRL+A da bi Olly analizirao ovaj deo koda i pojavice se sledece:

```
004012C0 /. 55              PUSH EBP
004012C1 |. 8BEC            MOV EBP,ESP
004012C3 |. 6A FF           PUSH -1
004012C5 |. 68 F8404000     PUSH crackme_.004040F8
```

Nasli smo mesto na kome cemo uraditi dump pomocu LordPEa. Zapamtite samo adresu na kojoj se nalazi OEP posto ce nam trebati, ta adresa je 004012C0 ili samo 000012C0 ako oduzmemo image base PE sekcije. Sada nam ostaje samo da pronadjemo sve importe i da ih zalepimo za dumpovan fajl. Da bismo ovo uradili ucitacemo ovaj proces u ImpRec. Promenicemo OEP na kome ce ImpRec traziti importe na 000012C0 i pritisnucemo IAT AutoSearch pa Get Imports. Sta je ovo? Nemamo ni jedan validan API poziv, pa cak ni jedan validan dll. Ne nismo mi nista uradili pogresno nego samo treba da potrazimo importe. Kliknimo na dugme Show Invalid, pa cemo kliknuti desnim dugmetom na nepoznate importe i selektovacemo Trace Level1 sto ce nam vratiti neke API pozive, ali ne i sve. Ponovo cemo kliknuti na dugme Show invalid pa cemo i taj poslednji API poziv naci na sledeci nacin. Kliknucemo desnim dugmetom na njega i izabradjemo Plugin Tracers -> ASProtect 1.22 i tako cemo pronaci i taj zadnji API poziv. Sada mozemo da pritisnemo dugme Fix dump i da popravimo dumpovan fajl. To je sve sto treba da uradimo kako bismo uspesno otpakovali ASProtect 1.2x.

Opisani nacin nije jedini na koji se moze otpakovati ASProtectom zasticena aplikacija. Ova verzija ASProtecta se takodje moze otpakovati i pomocu PeIDa, i to mnogo brze nego rucno. Otvorite PeID i skenirajte nasu metu pomocu njega, zatim selektujte dugme -> koje se nalazi u donjem desnom cosku i izaberite Plugins -> PeID Generic Unpacker. Pojavice se sledece:




Kada se pojavi ovaj prozor pritisnite dugme -> da detektujete OEP i pritisnite Unpack. Kada se zavrsi sa dumpovanje PeID ce vas pitati da li da popravi importe, a vi odgovorite sa Yes. Posle nekoliko sekundi imacete otpakovani program u istom direktorijumu kao i originalni fajl. Brzo i lako, ali lame :)

# ASProtect 2.0x

ASProtect 2.0x je novija verzija vec opisivanog protektora. Naravno i ovaj put ASProtect koristi IsDebuggerPresent API kako bi proverio da li je zasticena meta otvorena pomocu nekog debugera pa cemo morati i ovaj put da iskoristimo HideOlly plugin. Meta koju cemo ovaj put otpakivati zasticena je sa svim opcijama osim sa opcijom koja "krade kod" sa OEPa originalnog programa.Ova meta se nalazi u folderu Cas10 a zove se kgme2.ASProtect2.exe.

Kada otvorimo metu pomocu Ollyja videcemo sledeci OEP:

```
00401000 >/$  68 01504000    PUSH kgme2_AS.00405001
00401005  |.  E8 01000000    CALL kgme2_AS.0040100B
0040100A  \.  C3             RET
```

Sada bi trebalo da podesimo Olly kako bi smo sto brze dosli do OEPa otpakovane mete. Otvoricemo Debugging options prozor klikom na ALT+O i iskljuciti sve exceptione u Exceptions tabu osim memory access violation. Sada mozemo da pokusamo da startujemo metu, ali kao sto cemo videti meta se nece startovati i mi cemo se umesto na OEPu naci ovde:

```
00A7B497   90                NOP
00A7B498   EB 01              JMP SHORT 00A7B49B
00A7B49A   6966 81 FE47467>IMUL ESP,DWORD PTR DS:[ESI-7F],744647FE
```

Olly je zastao ovde jer je naisao na INT3 komandu. Pritisnucemo SHIFT+F9 kako bismo presli preko ovog exceptiona. Naizgled ce se ciniti da se nismo pomerili sa ove adrese ali to nije tacno, jer iako smo i dalje na ovoj adresi ASProtect je uspesno otpakovao sve sekcije. Sada mozemo da presretnemo izvrsenje otpakovanog koda postavljajuci memorijski breakpoint on access na glavnu .CODE sekciju. Kada ovo uradimo i ponovo pritisnemo SHIFT+F9 naci cemo se na pravom OEPu nase mete, to jest naci cemo se ovde:

```
00401000 >/$  25 0000FF00    AND EAX,0FF0000
00401005  |.  6A 00          PUSH 0
00401007  |.  68 1D104000    PUSH kgme2_AS.0040101D
0040100C  |.  6A 00          PUSH 0
0040100E  |.  6A 01          PUSH 1
00401010  |.  50             PUSH EAX
00401011  |.  E8 5C040000    CALL kgme2_AS.00401472
00401016  |.  6A 00          PUSH 0                          ; /ExitCode = 0
00401018  \.  E8 49040000    CALL kgme2_AS.00401466          ; \ExitProcess
```

Ovde mozemo da uradimo dump, ali ce popravka importa biti malo teza. Zasto pitate se? Pa iako ce ImpRec pronaci sve importe bez koriscenja Traceinga ili bilo kakvog dodatnog plugina, JMP pozivi ka API lokacijama ce biti unisteni. Jos jedan trik od strane ASProtecta koji moramo da prevazidjemo.

Da bismo ovo uradili popravicemo IAT pomocu ImpReca na sasvim normalan nacin pa cemo ne gaseci ImpRec otvoriti fajl koji je ImpRec popravio (*najcesce dumped_.exe*) pomocu Ollyja. Posto pokusavamo da popravimo IAT traceovacemo kroz kod sa F7 sve do poziva ka prvom APIju. Dakle uci cemo u sledeci CALL:

```
00401005  |.  6A 00          PUSH 0
00401007  |.  68 1D104000    PUSH dumped_.0040101D
0040100C  |.  6A 00          PUSH 0
0040100E  |.  6A 01          PUSH 1
00401010  |.  50             PUSH EAX
00401011  |.  E8 5C040000    CALL dumped_.00401472           <- Ulazimo u ovaj CALL
00401016  |.  6A 00          PUSH 0                                ; /ExitCode = 0
00401018  \.  E8 49040000    CALL <JMP.&kernel32.ExitProcess>      ; \ExitProcess
```

i zavrsicemo ovde:

```
00401472  $  E8 89EB8800   CALL 00C90000
```

Sta predstavlja ovaj CALL ka nepostojecoj adresi? ASProtect je koristio API redirekciju koja se nalazi na adresi 00C90000 pa je kao ostatak tog koda ostala ova invalidna redirekcija ka APIju koji bi trebalo da bude pozvan. Dakle sada treba da povezemo skokove ka API funkcijama sa samim APIjima. To znaci da cemo morati da izmenimo sledeci kod:

```
00401466  .- FF25 04704200   JMP kernel32.ExitProcess;
0040146C  $  E8 8FEB8800     CALL 00C90000
00401471    D8               DB D8
00401472  $  E8 89EB8800     CALL 00C90000
00401477    21               DB 21                    ;  CHAR '!'
00401478  $- FF25 10704200   JMP user32.EndDialog;
0040147E  $  E8 7DEB8800     CALL 00C90000
00401483    9C               DB 9C
00401484  $  E8 77EB8800     CALL 00C90000
00401489    D5               DB D5
0040148A  $- FF25 1C704200   JMP user32. SendDlgItemMessageA;
00401490  $- FF25 20704200   JMP user32.SendMessageA;
```

u validne skokove ka APIjima. Postoji vec par ovakvih validnih skokova i oni su u ovom isecku koda obelezeni zelenom bojom. Jedini problem koji trenutno imamo je neznanje koja adresa pripada kojem skoku ka APIju. Svaka adresa je rezervisana za samo jedan skok ka nekom APIju, zbog cega svi CALLovi koji pozivaju jedan API pozivaju istu adresu iz opsega od 00401466 do 00401490. Mi treba da otkrijemo koji to skok ka kojem APIju stoji na adresama na kojima se nalaze CALLovi ka ASProtect redirect kodu. Ovo bi inace bio veoma tezak posao da nemamo alat pod imenom ImpRec. Naime ImpRec sam radi grupisanje API poziva po .dll fajlovima u kojim se nalaze funkcije koje fajlovi pozivaju. To znaci da ce svi pozivi ka APIjima u ImpRecu biti poredjanji po redu, odnosno bice poredjani kao RVA lokacije APIja u fajlu koji otpakujemo. Sada mozemo da redom pristupamo adresama od 00401466 do 00401490 i da unosimo sledeci skok:

```
JMP NEAR DWORD PTR DS:[xxxxxxxx]
```

pri cemu cemo unositi RVA (RVA + ImageBase) lokacije koje vidimo u ImpRecu. Posto je prvi skok ka ExitProcess APIju validan on kao ASM funkcija izgleda ovako: JMP NEAR DWORD PTR DS:[427004]. Sledeci skok bi bio skok ka jedinoj preostaloj funkciji u kernel32.dll fajlu GetModuleHandleA a taj skok bi izgledao ovako JMP NEAR DWORD PTR DS:[427000]. I tako dalje sve dok ne ispunite sve preostale API pozive, odnosno dok skokovi ka APIjima ne izgledaju ovako:

```
00401466   - FF25 00704200   JMP NEAR DWORD PTR DS:<&kernel32.GetModuleHandleA>
0040146C   - FF25 04704200   JMP NEAR DWORD PTR DS:<&kernel32.ExitProcess>
00401472   - FF25 0C704200   JMP NEAR DWORD PTR DS:<&user32.DialogBoxParamA>
00401478   - FF25 10704200   JMP NEAR DWORD PTR DS:<&user32.EndDialog>
0040147E   - FF25 14704200   JMP NEAR DWORD PTR DS:<&user32.LoadIconA
00401484   - FF25 18704200   JMP NEAR DWORD PTR DS:<&user32.MessageBoxA
0040148A   - FF25 1C704200   JMP NEAR DWORD PTR DS:<&user32.SendDlgItemMessageA>
00401490   - FF25 20704200   JMP NEAR DWORD PTR DS:<&user32.SendMessageA>
```

Da bismo tacno odredili redosled JMP skokova ka APIjima treba da traceujemo kroz kod gledamo koje parametre uzima API koji se trenutno izvrsava i pokusavamo sve moguce kombinacije dok se konacno pravi API ne izvrsi i nas program ne napravi ni jedan exception ili *memory access violation*. Ovo je izuzetno komplikovano i zahteva dosta vremena ali se srecom po nas u ovom primeru redoslem JMPova poklopio sa RVA redosledom APIja.

# ReCrypt 0.15

Da bismo poceli sa otpakivanjem programa otvoricemo metu RETrace.ReCrypt015.exe pomocu OllyDBGa. Ono sto cemo videti na packer/crypter OEPu bice sledece:

```
00406000 >  60              PUSHAD
00406001    BE 2C604000     MOV ESI,RE-Trace.0040602C
00406006    B9 2C000000     MOV ECX,2C
0040600B    83F9 00         CMP ECX,0
0040600E    74 0A           JE SHORT RE-Trace.0040601A
```

Ovo ocigledno nije pravi OEP pa cemo nastaviti sa traceovanjem kroz kod sa F8. Pritiskali smo F8 sve dok nismo stigli do sledece komande:

```
00406018  ^\EB F1           JMP SHORT RE-Trace.0040600B
```

Kao sto vidimo svaki put kada se ova komanda izvrsi kod ce se vratiti na adresu 0040600B. Posto ce se ovaj deo koda izvrsavati dosta puta postavicemo jedan breakpoint odmah ispod ovog JMPa. Dakle postavicemo breakpoint na 0040601A, posle cega cemo pritisnuti F9 da bismo stigli do ovog breakpointa. Kao sto vidimo deo koda ispod se drasticno promenio. Bez obzira na ovo nastavicemo traceovanje kroz kod sa F8 sve dok ne dodjemo do sledeceg JMPa:

```
0040603A  ^\EB F0           JMP SHORT RE-Trace.0040602C
```

Posto ce se i ovaj JMP izvrsavati dosta puta postavicemo breakpoint odmah ispod njega i sa F9 cemo doci do njega. To ce nas dovesti ovde:

```
0040603C    8BC3            MOV EAX,EBX
0040603E    8BFE            MOV EDI,ESI
00406040    41              INC ECX
00406041    83F9 01         CMP ECX,1
00406044  ^ 74 E6           JE SHORT RE-Trace.0040602C
00406046    61              POPAD
00406047    B8 58604000     MOV EAX,RE-Trace.00406058
```

Iako smo izasli iz proslog JMPa videcemo da se i sledeci JE skok izvrsava veci broj puta pa cemo postaviti breakpoint na POPAD komandu. Posle ovoga cemo ponovo pritisnuti F9 da bismo dosli do naseg breakpointa. Konacno smo stigli do kraja algoritma za dekripciju:

```
00406047    B8 58604000     MOV EAX,RE-Trace.00406058
0040604C    83C0 10         ADD EAX,10
0040604F    8B00            MOV EAX,DWORD PTR DS:[EAX]
00406051    05 00004000     ADD EAX,RE-Trace.00400000
00406056    FFE0            JMP NEAR EAX
```

Sada samo treba da postavimo breakpoint na komandu JMP EAX i pritisnemo F9 da dodjemo do nje i F8 da se nadjemo na OEPu.

```
00401031  .  6A 00          PUSH 0
00401033  .  E8 94040000    CALL <JMP.&kernel32.GetModuleHandleA>
```

Kao sto vidimo na pravom smo mestu. Ovde treba da uradimo dump pomocu LordPEa. Ako pogledate malo detaljnije kod zakljucicete da je ovo samo kripter i da importi nisu pokvareni stoga nema potrebe za koriscenjem ImpReca. Bilo kako bilo naucili smo kako da brzo i lako dodjemo do OEPa ovog custom cryptera.

# ReCrypt 0.74

Da bismo poceli sa otpakivanjem programa otvoricemo metu packed.ReCrypt074.exe pomocu Ollyja. Ova novija verzija pakera donosi znacajan broj novina. Za pocetak importi se sada dinamicki alociraju a koristi se i API redirekcija kako bi se dodatno otezala popravka IATa, a i sam kod za odlazak na OEP je razbacan po CALLovima. Ali bez obzira na ovo mi cemo iskoristiti ovaj paker kako bismo naucili kada da rucno popravimo importe i kako da otpakujemo jos jedan paker.

Samo otpakivanje ovog pakera ce biti izuzetno lako ako se drzimo naseg sablonskog postupka. Dakle postavicemo memorijski breakpoint on access na glavnu .CODE sekciju da bismo posle pritiska na F9 zavrsili ovde:

```
004086CD   8B17             MOV EDX,DWORD PTR DS:[EDI]
004086CF   81F2 13151415    XOR EDX,15141513
004086D5   8917             MOV DWORD PTR DS:[EDI],EDX
004086D7   83C7 04          ADD EDI,4
004086DA   83C0 FC          ADD EAX,-4
004086DD  ^ EB E9           JMP SHORT packed_R.004086C8
004086DF   C9               LEAVE
004086E0   C2 0400          RET 4
```

Posto ce ovaj kod za dekripciju vise puta pristupati glavnoj .CODE sekciji potrebno je da uklonimo nas memorijski breakpoint i postavimo obican breakpoint na RET 4 komandu. Naravno posle pritiska na F9 naci cemo se na RET 4 komandi. Sada mozemo ponovo da postavimo breakpoint na glavnu sekciju, posto je ceo kod dekriptovan, i pristisnemo F9 kako bismo se nasli na OEPu dekriptovane mete. To jest naci cemo se ovde:

```
00401416  /.  55              PUSH EBP
00401417  |.  8BEC            MOV EBP,ESP
00401419  |.  6A FF           PUSH -1
0040141B  |.  68 E0504000     PUSH packed_R.004050E0
00401420  |.  68 0C204000     PUSH packed_R.0040200C              ;  SE handler installation
00401425  |.  64:A1 0000000>  MOV EAX,DWORD PTR FS:[0]
0040142B  |.  50              PUSH EAX
```

Posto je ovo ocigledno OEP ovde cemo uraditi memory dump i popravku importa. Ono sto je specificno za ovaj protektor je API redirekcija, to jest specifican je nacin na koji se ona izvodi tako da cemo morati da primenimo manje intervencije u ImpRecu kako bismo u potputnosti popravili IAT. Dakle posto pritisnete GetImport pojavice se netacni importi. Ove importe cemo popravljati tako sto cemo selektovati svaki od njih i desnim klikom cemo izabrati Disassemble/Hex View posle cega cemo videti sledece podatke:

```
00145F7F       push 77E694F2    // = kernel32.dll/0147/GetEnvironmentStrings
00145F84       jmp 00145F58
```

Ovo nam govori je prvi netacan import na adresi 00145F7F ustvari poziv ka GetEnvironmentStrings APIu koji se nalazi u kernel32.dll fajlu. Posto ovo znamo ostaje nam da zatvorimo ovaj Disassemble prozor i da duplim klikom na prvi netacan import otvorimo novi prozor koji nosi naziv Import Editor. U njemu cemo korigovati nas netacan import selekcijom .dll fajla u kojem se on nalazi i konacnom selekcijom tacnog APIja popravimo invalidan import. Ovaj postupak treba ponoviti za sve netacne pozive, posle cega nam ostaje da konacno popravimo dumpovan fajl pomocu ImpRecove opcije FixDump.

# ReCrypt 0.80

Najnovija verzija ReCrypta je donela jos par veoma zanimljivih novina koje su dizajnirane da nas sprece da otpakujemo ovu metu. Bilo kako bilo zahvaljujuci Cruddu koji mi je poslao najnoviju verziju ovog kriptera naucicemo nesto novo. Meta za ovaj deo poglavlja je Genesis.ReCrypt80.exe i nalazi se u folderu Cas10. Kada otvorimo ovu metu pomocu Ollyja videcemo da format .exe fajla ne zadovoljava PE standard zbog cega cemo se naci ovde:

```
77F767CE   C3              RET
```

Posto zbog ove cinjenice necemo biti u mogucnosti da pristupamo sekcijama moracemo da otvorimo fajl pomocu LordPEa da bismo videli u kojoj sekciji se nalazi OEP i na kojoj je on adresi. Dakle adresa OEPa je 00405000 a on se nalazi u poslednjoj sekciji RE-Crypt. Zbog ovoga cemo pritisnuti CTRL+G u Ollyju i otici cemo na adresu 00405000, na kojoj cemo postaviti obican breakpoint. Klikom na F9 stizemo do naseg breakpointa, odnosno nalazimo na packer OEPu:

```
00405000   60              PUSHAD
00405001   E8 00000000     CALL Genesis_.00405006
00405006   5D              POP EBP
00405007   81ED F31D4000   SUB EBP,Genesis_.00401DF3
```

Sada bismo trebali da postavimo memorijski breakpoint na glavnu sekciju, ali posto nismo u mogucnosti to da uradimo (Olly ne vidi ni jednu sekciju) iskoristicemo nas STACK trik. Dok se nalazimo na PUSHAD komandi pritisnucemo F8 posle cega ce ESP registar dobiti novu vrednost. Ispraticemo tu vrednost u dumpu posle cega cemo selektovati prva cetiri bajta u hex dump prozoru:

```
0012FFA4  FF FF FF FF 6A 16 F5 77  ....j..w
0012FFAC  F0 FF 12 00 C4 FF 12 00  ........
```

na ova cetiri cemo postaviti Hardware breakpoint -> On access -> dword… Kada sada pokusamo da startujemo metu klikom na F9 doci cemo do sledece lokacije:

```
0040501B   60              PUSHAD
0040501C   E8 00000000     CALL Genesis_.00405021
```

Nismo bas daleko otisli, ali ako kliknemo na F9 jos jedan put naci cemo se ovde:

```
004053F6   50              PUSH EAX                ; Genesis_.00401000
004053F7   33C5            XOR EAX,EBP
004053F9   E8 1D000000     CALL Genesis_.0040541B
```

Naravno vidimo da EAX sadrzi vrednost koja moze biti ili vrednost prve adrese u prvoj sekciji ili adresa OEPa. Iz ovog razloga cemo traceovati kroz kod sa F7, traceujuci kroz CALL na adresi 004053F9, stizuci do sledeceg koda:

```
0040541B   58              POP EAX                 ; Genesis_.004053FE
0040541C   C3              RET
```

Sada je vec ocigledno da je 00401000 pravi OEP iz razloga sto ce se posle izvrsavanja komande PUSH EAX, adresa 00401000 naci na STACKu, a posle izvrsavanja RET komande i sam program ce skociti na adresu 00401000, na kojoj se bez sumnlje nalazi pravi OEP:

```
00401000   6A 00           PUSH 0
00401002   E8 D9000000     CALL Genesis_.004010E0 ; JMP to kernel32.GetModuleHandleA
00401007   A3 40304000     MOV DWORD PTR DS:[403040],EAX
```

# ACProtect 1.4x

ACProtect je komercijalan protektor cija je najnovija verzija datirana na 2004 godinu. U originalu ime protektora je AntiCrack protector ili kako ga PeID identifikuje UltraProtect. I jedan i drugi naziv zvuce veoma privlacno i prosto traze detaljnu analizu. Naravno ovo je izazov koji nikako ne smemo propustiti i stigo cemo otvoriti metu crackme.ACProtect.exe pomocu Ollyja. Na samom OEPu ACProtectora cemo videti sledece:

```
00407000 > $  60              PUSHAD
00407001   . E8 01000000      CALL crackme_.00407007
00407006     E9               DB E9
00407007  /$  830424 06       ADD DWORD PTR SS:[ESP],6
0040700B  \.  C3              RET
```

Ovo je kod koji neodoljivo podseca na ASProtect, ali za razliku od ASProtecta mi cemo ovde iskoristiti stack trik kako bismo veoma brzo dosli do pravog OEPa. Dakle, kao i kod ASPacka, pritisnucemo F8 i videcemo da se ESP registar promenio zbog cega cemo ga izabrati (*a i zbog cinjenice ga on pokazuje na STACK*) i selektovacemo opciju Follow in Dump posle cega cemo videti sledece u Hex View prozoru:

```
0012FFA4  FF FF FF FF 6A 16 F5 77  ....j..w
0012FFAC  F0 FF 12 00 C4 FF 12 00  ........
```

Kao i kod ASPacka selektovacemo prva cetiri bajta na 0012FFA4 adresi (*moze se razlikovati na vasem Windowsu*) i desnim klikom cemo izabrati Breakpoint, pa Hardware on Access, pa DWORD. Sada nam ostaje da pritiskamo F9 dok se ne nadjemo u blizini OEPa. Posle prvog klika na F9 naci cemo se ovde:

```
004070A3   . 51               PUSH ECX
004070A4   . 8F05 B1784000    POP DWORD PTR DS:[4078B1]
004070AA   . 60               PUSHAD
```

Ne ovo nije OEP, pritisnucemo 2x F9 (*CTRL + A*) i ovde smo:

```
004071A3   . 51               PUSH ECX
004071A4   . 8F05 E9784000    POP DWORD PTR DS:[4078E9]
004071AA   . 60               PUSHAD
```

I dalje nismo na OEPu pa cemo pritisnuti F9 jos 2x i naci cemo se ovde:

```
0040728D   . 56               PUSH ESI                  ; ntdll.77F5166A
0040728E   . 8F05 7D794000    POP DWORD PTR DS:[40797D]
00407294   . 60               PUSHAD
```

Pritiskamo F9 sve dok ne dodjemo dovde:

```
0041D6A8     EB               DB EB
0041D6A9     01               DB 01
0041D6AA     7F               DB 7F
```

a kada i ovde pritisnemo F9 pojavljuje se ACProtect NAG. Ovo znaci da smo blizu OEPa, pa cemo posle klika OK na NAGu analizirati mesto gde se nalazimo, klikom na CTRL+A dobijamo:

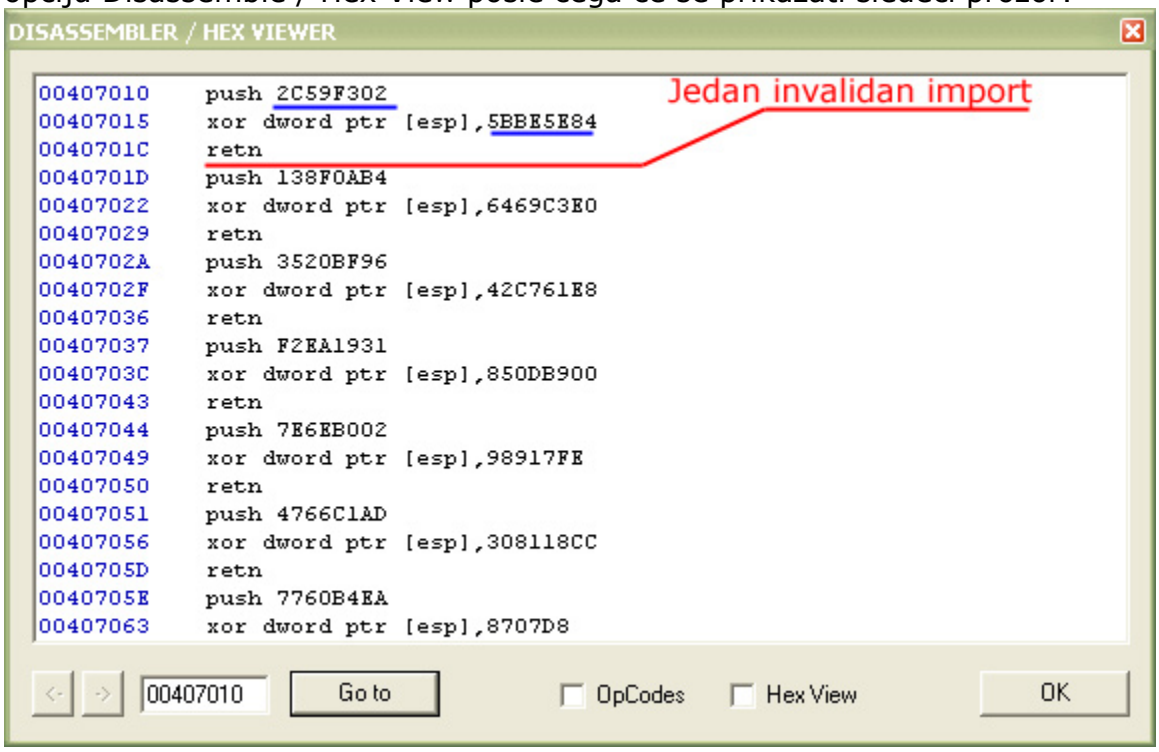
```
0041E106   . /EB 01           JMP SHORT crackme_.0041E109
0041E108     |E8             DB E8
0041E109   >-\FF25 4BE14100 JMP NEAR DWORD PTR DS:[41E14B]    ; crackme_.004012C0
0041E10F   . 60               PUSHAD
```

Izvrsicemo JMP skok na kojem se nalazimo sa F8 jer nam sledeci skok koji vodi na 004012C0 izgleda sumnljivo (*a i otpakivali smo ovaj crackme toliko puta da vec znamo da je OEP 004012C0*). Posle izvrsavanja i ovog sumnljivog skoka na 004012C0 dolazimo ovde:

```
004012C0  /. 55               PUSH EBP
004012C1  |. 8BEC             MOV EBP,ESP
004012C3  |. 6A FF            PUSH -1
```

I kao sto smo i mislili 004012C0 je OEP i ovde mozemo uraditi memory dump pomocu LordPEa, posle cega nam ostaje samo da popravimo importe pomocu ImpReca da bismo uspesno otpakovali ovaj protektor.

Ali kao sto smo vec navikli kada su u pitanju komercijalni protektori i ovaj ima API redirekciju zbog cega se IAT mora popravljati rucno. Da bismo ovo uradili selektovacemo bilo koji invalidan import iz liste i izabracemo opciju Disassemble / Hex View posle cega ce se prikazati sledeci prozor:




U njemu sam vec oznacio sve sto se odnosi na selektovani import i kao sto se i vidi sa slike radi se o izuzetno jednostavnoj redirekciji koja se zasniva na PUSHovanju konstante na STACK i na njenom XORovanju slucajnom vredoscu kako bi se dobila adresa na kojoj se nalazi import. Ako izracunamo 2C59F302 XOR 5BBE5E84 dobicemo 77E7AD86, sto sigurno lici na adresu na kojoj se nalazi import. Stoga cemo otici na doticnu adresu u Olly i videcemo koji se tamo import nalazi:

```
77E7AD86 >  837C24 04 00      CMP DWORD PTR SS:[ESP+4],0
77E7AD8B    0F84 37010000     JE kernel32.77E7AEC8
...
77E7AD9E    FF70 04           PUSH DWORD PTR DS:[EAX+4]
77E7ADA1    E8 27060000       CALL kernel32.GetModuleHandleW
77E7ADA6    C2 0400           RET 4
```

Naravno mislim da svi vide da je ovo poziv ka GetModuleHandleW APIju stoga mozemo izabrati njega duplim klikom na invalidan import u glavnom prozoru ImpReca. Naravno ovaj postupak treba ponoviti za sve invalidne API pozive stoga predlazem da umesto rucnog popravljanja importa izaberete ImpRec plugin pod imenom acpr koji ce za vas popraviti sve importe osim jednog koji se nalazi na adresi 00407273. Ako pogledate njega na gore opisan nacin videcete da on pokazuje na 00415E4B, odnosno nazad na ACProtect kod zbog cega se ovaj import moze obrisati komandom Cut trunks.

# WinUPack 0.2x

WinUPack je jedan odlican novi packer sa zavidnim nivoom pakovanja, koji je jedan od trenutno najmanjih pakera koje sam do sada sreo i testirao. Nasa meta, koju cemo otpakivati, je pre pakovanja imala 5.0 kb a posle pakovanja ima 3.58 kb. Poredjenja radi samo MEW 1.1 je bolji sa ratiom od 3.07 kb, ali ta razlika nestaje i prelazi u koristi WinUPacka prilikom kompresovanja vecih fajlova. Meta za ovaj deo poglavlja je WinUPack.exe a nalazi se u folderu Cas10. Njeno otpakivanje je krajnje jednostavno jer je WinUPack paker a ne protektor, stoga se otpakivanje svodi na pretrazivanje koda za skokom koji vodi u glavnu .Code sekciju, sto je slicno postupku otpakivanja FSGa. Bez nekog detaljisanja pronalazimo skok ka OEPu:

```
00405B87   85C0              TEST EAX,EAX
00405B89  - 0F84 C1B6FFFF    JE WinUPack.00401250
00405B8F   56                PUSH ESI
```

na koji postavljamo obican breakpoint i pritiskamo F9. WinUPack ce doci do ovog breakpointa vise puta pa stoga posmatramo Olly i cekamo da se skok izvrsi. Nadgledamo deo ekrana koji se nalazi odmah ispod glavnog code windowa, a sadrzi sledeci tekst:

```
Jump is NOT taken
00401250=WinUPack.00401250
```

Naravno sada nam ostaje da pritiskamo F9 sve dok se tekst koji posmatramo ne promeni u sledece:

```
Jump is taken
00401250=WinUPack.00401250
```

Posle cega nam ostaje samo da pritisnemo F8 i naci cemo se na OEPu koji trenutno izgleda ovako:

```
00401250    55        DB 55                ; CHAR 'U'
00401251    8B        DB 8B
00401252    EC        DB EC
00401253    83        DB 83
00401254    EC        DB EC
00401255    44        DB 44                ; CHAR 'D'
00401256    56        DB 56                ; CHAR 'V'
```

Zbog cega se naravno mora uraditi analiza koda pomocu klika na CTRL+A, posle cega kod od gore postaje:

```
00401250 /. 55             PUSH EBP                              ; USER32.77D40000
00401251 |. 8BEC             MOV EBP,ESP
00401253 |. 83EC 44          SUB ESP,44
00401256 |. 56              PUSH ESI
00401257 |. FF15 0C204000   CALL NEAR DWORD PTR DS:[40200C]     ; [GetCommandLineA]
0040125D |. 8BF0            MOV ESI,EAX
0040125F |. 8A06            MOV AL,BYTE PTR DS:[ESI]
00401261 |. 3C 22           CMP AL,22
00401263 |. 75 14           JNZ SHORT WinUPack.00401279
```

Naravno kao sto smo to radili sa svakim pakerom ovde radimo memory dump pomocu LordPEa ili ProcDumpa i na dumpovanom fajlu radimo popravku importa pomocu ImpReca, koji ce naravno pronaci sve importe odmah jer se ovde radi o obicnom pakeru, pa nema redirekcije ili enkripcije API poziva. Jos jednom ovo je odlican paker koji ce drasticno smanjiti velicinu vasih .exe fajlova stoga predlazem da ga koristite ako radite na fajlovima koji prelaze velicinu jednog megabajta.

# Neolite 2.0

Evo kako se otpakuje jedan od najlaksih pakera koje sam ikada video. Meta se nalazi u folderu Cas10 a zove se Artemis.Neolite20.exe. Sam OEP izgleda ovako:

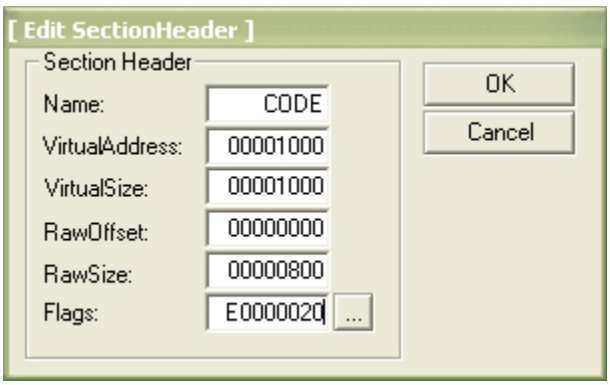
```
00411110 > $ /E9 A6000000    JMP Artemis_.004111BB
...
004111BB   > \8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]
004111BF   . 2305 21114100   AND EAX,DWORD PTR DS:[411121]
004111C5   . E8 ED040000     CALL Artemis_.004116B7
004111CA   . FE05 BA114100   INC BYTE PTR DS:[4111BA]
004111D0   . FFE0            JMP EAX
004111D2   . 803D BA114100>  CMP BYTE PTR DS:[4111BA],0
004111D9   . 75 13           JNZ SHORT Artemis_.004111EE
004111DB   . 90              NOP
004111DC   . 90              NOP
004111DD   . 90              NOP
004111DE   . 90              NOP
004111DF   . 50              PUSH EAX
004111E0   . 2BC0            SUB EAX,EAX
004111E2   . E8 D0040000     CALL Artemis_.004116B7
004111E7   . 58              POP EAX
004111E8   . FE05 BA114100   INC BYTE PTR DS:[4111BA]
004111EE   >  C3             RET
```

Da biste uspesno otpakovali ovaj paker samo treba da pritiskate F8 polako izvrsavajuci jedan po jedan red sve dok ne izvrsite i JMP EAX komandu, posle cega cete se naci ovde:

```
00401000   6A 00             PUSH 0
00401002   E8 2A060000       CALL Artemis_.00401631
00401007   A3 36214000       MOV DWORD PTR DS:[402136],EAX
0040100C   C705 7B214000 0>  MOV DWORD PTR DS:[40217B],0B
00401016   C705 7F214000 E>  MOV DWORD PTR DS:[40217F],Artemis_.00401>
00401020   C705 83214000 0>  MOV DWORD PTR DS:[402183],0
```

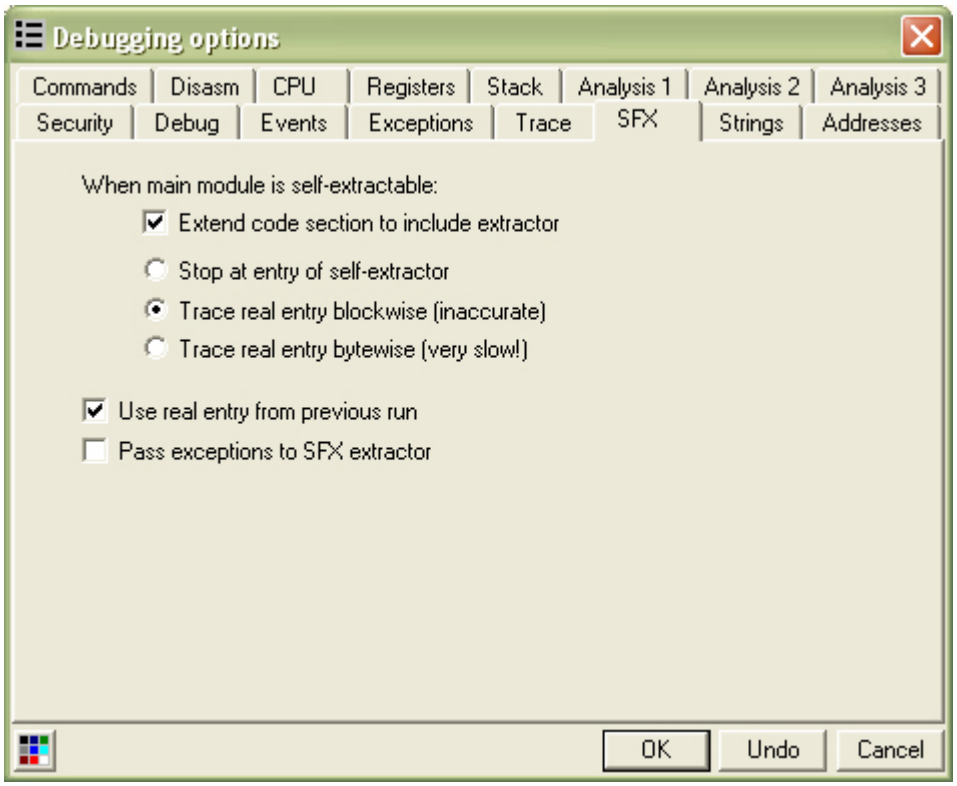
a ovo je ujedno i pravi OEP. Ovde samo treba da uradimo full memory dump i da popravimo importe. Ako necete Neolite 2.0 da otpakujete rucno mozete to da uradite i pomocu PeIDovog generickog unpackera. Lako zar ne?

**NAPOMENA:** Ako ovaj packer odpakujete rucno moracete da podesite karakteristike sekcije CODE na E0000020. Ovo se moze uraditi pomocu PE editora koji se nalazi u LordPEu. Kada otvorite fajl u LordPEu onda izaberite dugme Sections pa na CODE, desnim dugmetom izaberite Edit Section Header i u polje Flags unesite E0000020. Ovo radimo zato sto sekcija CODE mora da ima takve atribute tako da se moze citati, izvrsavati ali i da se po noj moze pisati. Mada moze i bez ovoga ali je ovako svakako sigurnije.



# PE Lock NT 2.04

Na ovom packeru cu pokazati kako nam neke opcije iz Ollyja mogu pomoci da nadjemo OEP za samo jednu sekundu. Meta se nalazi u folderu Cas10 a zove se TSC_crkme10.PeLockNt204.exe, da li ovaj paker radi samo na NT sistemima ne znam, ali znam da ima jednu gresku. Naime kada je kod pakovan sa ovim pakerom crackme jednostavno nece da se startuje. Ovo cemo da resimo jednostavnim otpakivanjem. Pre nego sto pocnemo podesicemo Olly ovako:




Ova opcija nam omogucuje da analiziramo SFX kod tako da se odmah nadjemo na samom OEPu. Ovo je jako korisno jer nas kod ovog pakera vodi direktno da sam OEP koji izgleda ovako:

```
00401260   .  55              PUSH EBP                    ;  Real entry point of SFX code
00401261   .  8BEC            MOV EBP,ESP
00401263   .  6A FF           PUSH -1
00401265   .  68 00304000     PUSH TSC_crkm.00403000
0040126A   .  68 78224000     PUSH TSC_crkm.00402278      ;  SE handler installation
0040126F   .  64:A1 0000000>  MOV EAX,DWORD PTR FS:[0]
```

Posto se vec nalazimo na OEPu mozemo slobodno da uradimo memory dump i da popravimo importe pomocu ImpReca i zavrsili smo brzo i lako i sa ovim pakerom.

# Virogen Crypt 0.75

Na PELocku sam vam vec pokazao kako se konfigurise Olly da bi sam pronasao pravi OEP. Ovaj primer ce vam pokazati kako iskoristite ovu opciju kada vas Olly dovede samo na pola puta do OEPa. Meta na kojoj cu vam ovo pokazati se zove crackme #1.VirogenCrypt075.exe, kada je ucitamo u Olly on ce za nas naci "pravi" OEP. On izgleda ovako:

```
0040100E  > \66:B8 0000      MOV AX,0                ;  Real entry point of SFX code
00401012  . EB 00            JMP SHORT crackme_.00401014
00401014  >  40              INC EAX
00401015  . 83F8 64          CMP EAX,64
00401018  .^ 72 FA           JB SHORT crackme_.00401014
0040101A  .  BB 64000000     MOV EBX,64
0040101F  . 3BC3             CMP EAX,EBX
00401021  . 8D3D F7104000    LEA EDI,DWORD PTR DS:[4010F7]
00401027  . 2BFB             SUB EDI,EBX
00401029  . FFE7             JMP EDI
```

Naravno ovo sigurno nije pravi OEP. Ako pocnemo da izvrsavamo ovaj dugacak loop:

```
00401018  .^\72 FA           JB SHORT crackme_.00401014
```

to ce potrajati, pa cemo postaviti jedan breakpoint odmah ispod ovog skoka. Postavicemo breakpoint na 0040101A i pritisnucemo F9. Kada se ovaj loop zavrsi mi cemo se naci na nasem breakpointu. Sada cemo izvrsiti sve do skoka JMP EDI sa F8 i kada izvrsimo i taj skok naci cemo se ovde:

```
00401093  . 6A 00            PUSH 0
00401095  . 8D05 52304000    LEA EAX,DWORD PTR DS:[403052]
0040109B  . 50               PUSH EAX
0040109C  . 8D1D 00304000    LEA EBX,DWORD PTR DS:[403000]
004010A2  . 53               PUSH EBX
```

Ovo lici na sam OEP, a da li je? Naravno mozete probati i da izvrsite kod na dole i videcete da je ovo u stvari pravi OEP. Znaci adresa na kojoj mozemo izvrsiti dump je 00401093. Posle ovoga nam ostaje samo da popravimo importe.

**NAPOMENA:**

Kako znamo da je ovo pravi OEP?

Pa i ne znamo u stvari mozemo samo da nagadjamo. Ali ako se dugo vremena bavite reversingom videcete da se sami OEPi razlikuju samo ako reversujete exe koji su kompajlovani drugim kompajlerima. Tako ce VB uvek imati isti OEP, Delphi isto, i svi ostali programski jezici ne racunajuci TASM / MASM jer kod njih OEP moze da izgleda proizvoljno i zavisi od samog kodera sto nije slucaj sa VB,Delphijem,...

# EZIP 1.0

Ovo je jos jedan lak paker za otpakivanje. Otvorite primer zapakovan ovim pakerom koji se zove Unpackme12.EZIP1.exe pomocu Ollyja. Prva linija pakera ce izgledati ovako:

```
004060BE > $ /E9 19320000          JMP Unpackme.004092DC
```

Izvrsite ovaj skok sa F8 i zavrsicete ovde:

```
004092DC  /> \55                   PUSH EBP
004092DD  |.  8BEC                  MOV EBP,ESP
004092DF  |.  81EC 28040000        SUB ESP,428
004092E5  |.  53                    PUSH EBX
```

Mozete da traceujete kroz ovaj kod ali nema potrebe :) samo odskrolujte do samog kraja ovog CALLa i videcete sledece:

```
00409688  |.  FFE0                 JMP EAX       <- BPX ON
0040968A  |>  5F                   POP EDI
0040968B  |.  5E                   POP ESI
0040968C  |.  5B                   POP EBX
0040968D  |.  C9                   LEAVE
0040968E  \.  C3                   RET
```

Sada samo treba da postavite jedan obican breakpoint na JMP EAX komandu, pritisnete F9 da biste dosli do te komande i izvrsite je sa F8. Posle ovoga cete se naci na nasem pravom OEPu, to jest ovde:

```
00401000   6A 00                   PUSH 0
00401002   E8 0D010000             CALL Unpackme.00401114
```

Sada mozete da uradite dump pomocu OllyDMP plugina. Default podesavanja su OK. Posle ovoga cete dobiti uspesno otpakovan EZIPovan fajl.

# SPEC b3

Kao sto smo videli kod EZIPa, JMP EAX komanda nas kod nekih pakera moze odvesti do samog OEPa. Ovo cemo iskoristiti za otpakivanje SPEC b3a. Primer se nalazi u folderu Cas10, a zove se Notepad.SPEC3.exe. Ucitajte ga u Olly i samo odskrolujte dole dok ne naidjete na:

```
0040D14F   FFE0                    JMP EAX
```

Postavite breakpoint na JMP EAX, pritisnite F9 da biste dosli do tog breakpointa i sa F8 izvrsite skok i zavrsicete na OEPu. Posle ovoga samo uradite dump pomocu OllyDMP plugina i otpakivanje je gotovo :)

# CExe 1.0a - 1.0b

Ovo je klasican primer kako ne treba pisati zastitu za vase aplikacije. Naime iako se fajl nalazi enkriptovan unutar samog .exe fajla on se prilikom dekripcije kompletno otpakuje na disk. Primer ove veoma lose zastite se nalazi u folderu Cas10 a zove se unpackme9.CExe.exe. Kako cete otpakovati program “zasticen” ovim pakerom? Jednostavno startujte zapakovani fajl i sacekajte da se on pokrene, kada se pokrene primeticete da se u istom direktorijumu nalazi jedan .tmp fajl. Vi samo treba da iskopirate taj tmp fajl gde zelite i preimenujete ga u .exe fajl. Dekriptovanje je gotovo, a nismo cak ni upalili Olly :)

# MEW v.1.1 SE

Ovo je jos jedan lak paker za otpakivanje. Otvorite primer zapakovan ovim pakerom koji se zove unpackme18.MEW11SE.exe pomocu Ollyja. Prva linija pakera ce izgledati ovako:

```
0040732B >- E9 248EFFFF          JMP unpackme.00400154
00407330    0C 60                OR AL,60
```

Izvrsite ovaj skok sa F8 i zavrsicete ovde:

```
00400154    BE 1C604000          MOV ESI,unpackme.0040601C
00400159    8BDE                 MOV EBX,ESI
0040015B    AD                   LODS DWORD PTR DS:[ESI]
0040015C    AD                   LODS DWORD PTR DS:[ESI]
```

Iako ovaj kod izgleda malo komplikovano otpakuje se za samo nekoliko sekundi. Odskrolujte do samog kraja ovog CALLa, to jest do prve RET komande. Ona se nalazi ovde:

```
004001F0    FF53 F4              CALL DWORD PTR DS:[EBX-C]
004001F3    AB                   STOS DWORD PTR ES:[EDI]
004001F4    85C0                 TEST EAX,EAX
004001F6  ^ 75 E5                JNZ SHORT unpackme.004001DD
004001F8    C3                   RET
```

Secate se sta smo rekli? Do OEPa se moze doci pomocu JMP, CALL, RET komandi. Probacemo ovu teoriju u praksi i postavicemo jedan breakpoint na adresu 004001F8, to jest na RET komandu. Posle toga cemo izvrsiti sav kod koji je potreban da se stigne do nje sa F9. Kada konacno stanemo kod nje izvrsicemo i nju sa F8 i to ce nas odvesti ovde:

```
00401000    6A                   DB 6A                ; CHAR 'j'
00401001    00                   DB 00
00401002    E8                   DB E8
00401003    0D                   DB 0D
00401004    01                   DB 01
00401005    00                   DB 00
```

Da li je ovo mesto koje smo trazili? Da li je ovo OEP? Znam da vam ovo izgleda kao neke kuke i kvake ali ako analiziramo ovaj kod pritiskom na CTRL + A videcemo sta se to zaista nalazi na adresi 00401000. Videcemo ovo:

```
00401000  . 6A 00                PUSH 0
00401002  . E8 0D010000          CALL unpackme.00401114
00401007  . A3 00304000          MOV DWORD PTR DS:[403000],EAX
```

E ovo vec izgleda kao pravi OEP pa stoga ovde mozemo uraditi memory dump pomocu alata koji vi izaberete. Posto MEW nema nikakve API redirekcije ili kriptovanje API poziva dump i popravku APIa mozemo da uradimo preko standardnog OllyDMP plugina i sve ce biti OK.

# PEBundle 2.0x - 2.4x

Sa pakerom Jeremy Collaka smo se vec sreli kada smo otpakivali PECompact. Sada je na redu jos jedan njegov paker PEBundle. Otvorite primer zapakovan ovim pakerom koji se zove unpackme21.PEBundle2x.exe pomocu Ollyja. Prva linija pakera ce izgledati ovako:

```
00442000 >  9C                  PUSHFD
00442001    60                  PUSHAD
00442002    E8 02000000         CALL unpackme.00442009
```

Pritisnucemo F7 3x da bismo usli u prvi CALL. To ce nas odvesti ovde:

```
00442009    8BC4                MOV EAX,ESP
0044200B    83C0 04             ADD EAX,4
```

sada cemo samo odskrolovati do samog kraja ovog CALLa, to jest do prve RET komande. Ovo radimo zato sto smo usli u CALL, a jedini nacin da se vratimo iz CALLa je pomocu RET komande. Prvu RET komandu cemo naci tek ovde:

```
00442466    61                  POPAD
00442467    9D                  POPFD
00442468    68 00B04200         PUSH unpackme.0042B000
0044246D    C3                  RET
```

Postavicemo jedan breakpoint na adresu 0044246D, to jest na samu RET komandu. Pritisnucemo F9 da bismo dosli do te RET komande, a sa F8 cemo izvrsiti i tu RET komandu posle cega cemo se naci ovde:

```
0042B000    9C                  PUSHFD
0042B001    60                  PUSHAD
0042B002    E8 02000000         CALL unpackme.0042B009
```

Pritisnucemo F7 3x da bismo usli i u ovaj CALL i ponovo cemo potraziti najblizu RET komandu. Ona se nalazi ovde:

```
0042B466    61                  POPAD
0042B467    9D                  POPFD
0042B468    68 EC154000         PUSH unpackme.004015EC
0042B46D    C3                  RET
```

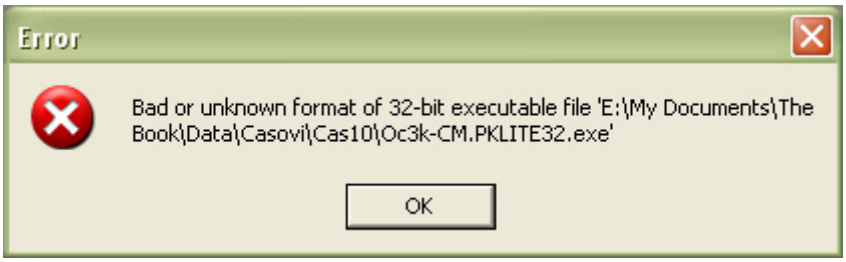
Posle ovoga cemo postaviti breakpoint na RET komandu sa F9 cemo doci do nje i sa F8 cemo je izvrsiti, posle cega cemo zavrsiti ovde:

```
004015EC     68                 DB 68                    ;  CHAR 'h'
004015ED     8C264000           DD unpackme.0040268C     ;  ASCII "VB5!6&*"
004015F1     E8                 DB E8
004015F2     EE                 DB EE
```

Kao sto vidimo nalazimo se na OEPu koji i bez i sa analizom izgleda ovako kako izgleda jer je u pitanju Visual Basic aplikacija. Ovo je adresa na kojoj mozemo da uradimo dump pomocu LordPEa ili OllyDMPa. Posto PEBundle ne enkriptuje IAT mozemo da uradimo dump pomocu OllyDMPa koji ce za nas popraviti i importe.

# PkLite 32 v.1.1

I evo i jednog pakera koji potice jos iz vremena DOS aplikacija. Bas iz ovog razloga sam dosta dugo i trazio verziju koja bi mogla da kompresuje i Win aplikacije. Jedna takva meta se nalazi u folderu Cas10 a zove se Oc3k-CM.PKLITE32.exe.Otpakovacemo je pomocu Ollyja. Primeticete prilikom otvaranja fajla sa Ollyjem sledecu poruku:




Nista ne brinite oko ove poruke! Ona se pojavljuje samo iz razloga sto tvorci PkLitea nisu ispostovali strukturu PE fajla do samog kraja. Bez obzira na ovaj mali propust mi cemo uspeti da otpakujemo ovaj paker u samo par jednostavnih poteza. Pre nego sto pocnete pogledajte kako izgleda OEP pakera:

```
0043D000 > $  68 80D04300    PUSH Oc3k-CM_.0043D080          ; /Arg3 = 0043D080
0043D005   .  68 53A54500    PUSH Oc3k-CM_.0045A553          ; |Arg2 = 0045A553
0043D00A   .  68 00000000    PUSH 0                          ; |Arg1 = 00000000
0043D00F   .  E8 3FD50100    CALL Oc3k-CM_.0045A553          ; \Oc3k-CM_.0045A553
0043D014   .^ E9 AB0FFFFF    JMP Oc3k-CM_.0042DFC4
```

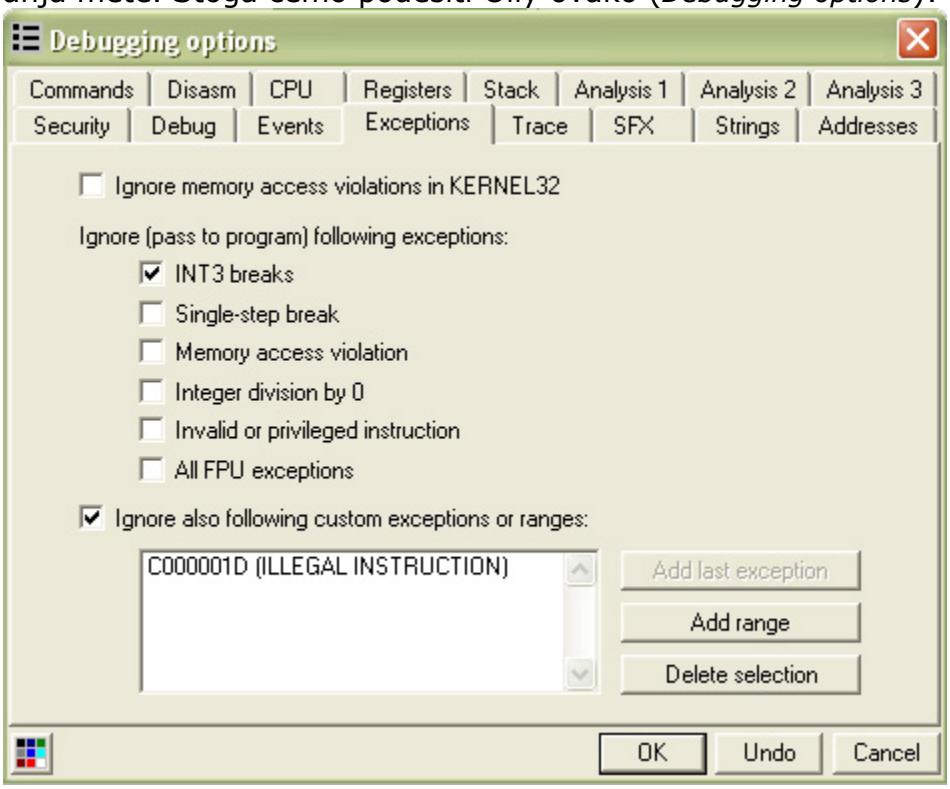
Mozete ako zelite da udjete u ovaj prvi CALL, ali nema potrebe za tim posto cete na kraju kada se izvrsi poslednja RET komanda ovog CALLa vraticemo se bas na onaj JMP skok ispod CALLa. A kada se i on izvrsi zavrsicemo na OEPu.

```
0042DFC4   55               PUSH EBP
0042DFC5   8BEC             MOV EBP,ESP
0042DFC7   83C4 F4          ADD ESP,-0C
0042DFCA   B8 F4DE4200      MOV EAX,Oc3k-CM_.0042DEF4
0042DFCF   E8 846FFDFF      CALL Oc3k-CM_.00404F58
0042DFD4   A1 DCF94200      MOV EAX,DWORD PTR DS:[42F9DC]
0042DFD9   8B00             MOV EAX,DWORD PTR DS:[EAX]
0042DFDB   E8 A0A7FFFF      CALL Oc3k-CM_.00428780
0042DFE0   A1 DCF94200      MOV EAX,DWORD PTR DS:[42F9DC]
0042DFE5   8B00             MOV EAX,DWORD PTR DS:[EAX]
0042DFE7   BA 24E04200      MOV EDX,Oc3k-CM_.0042E024   ASCII "OutCast3k Crack me :P"
0042DFEC   E8 B7A4FFFF      CALL Oc3k-CM_.004284A8
0042DFF1   8B0D 50FA4200    MOV ECX,DWORD PTR DS:[42FA50]   ; Oc3k-CM_.00430744
0042DFF7   A1 DCF94200      MOV EAX,DWORD PTR DS:[42F9DC]
0042DFFC   8B00             MOV EAX,DWORD PTR DS:[EAX]
0042DFFE   8B15 FCD84200    MOV EDX,DWORD PTR DS:[42D8FC]   ; Oc3k-CM_.0042D93C
0042E004   E8 8FA7FFFF      CALL Oc3k-CM_.00428798
0042E009   A1 DCF94200      MOV EAX,DWORD PTR DS:[42F9DC]
```

Sada ostaje samo da dumpujete fajl pomocu LordPEa ili OllyDMPa i da popravite importe pomocu ImpReca. Sami importi su OK i nema potrebe za bilo kojim levelom tracea.

# PeX 0.99

PeX je jedan jako zanimljiv a ujedno i lak paker za reversing. Zasto zanimljiv? Iz razloga sto ima jako zanimljivu tehniku kojom detektuje debuggere. Meta pakovana ovim pakerom se nalazi u folderu Cas10 a zove se Phoenix2.PEX099. Naime program pakovan ovim pakerom se ne moze startovati pomocu Ollyja jer ce biti detektovan i program se nece startovati jer program detektuje takozvani stepping kroz kod. Ovo znaci da ce PeX detektovati Olly samo ako debuggujemo program sa F7/F8 ali ne i sa F9! Sada ostaje samo da se resimo par problema koji ce nas "zaustaviti" i onemoguciti PeXu da odmah otpakuje metu. Ova dva problema su INT3 stop i Illegal Instruction greska. Na njih smo naisli prilikom pritiska na F9 i startovanja mete. Stoga cemo podesiti Olly ovako (*Debugging options*):



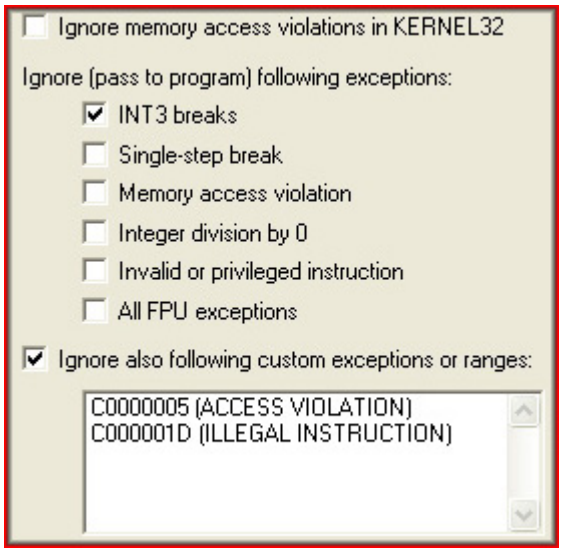

Posle ovoga ostaje da odemo u Memory map prozor (*ALT+M*) i postavimo jedan memorijski break point (*on access*) na .code sekciju nase mete.
Sada cemo pokrenuti program sa F9 i sacekacemo da se on zaustavi na OEPu. Ali na zalost nismo se zaustavili odmah na OEPu, nasli smo se ovde:

```
0085006B   F3:A4            REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

Posto se REP komanda izvrsava mnogo puta vraticemo se u Memory map prozor i skinucemo memory break point, pa cemo se vratiti u CPU prozor gde cemo postaviti obican breakpoint na adresu 0085006D. Sada cemo pritisnuti F9 da bismo stigli do tog breakpointa, a kada se to desi uklonicemo ga sa F2 i ponovo cemo postaviti memory breakpoint na .CODE sekciju. I posle jos jednog pritiska na F9 doci cemo do samog OEPa nase mete. Ovde mozete uraditi dump, dok cete prilikom popravljanja importa morati da uradite Level1 trace da bi ste vratili sve importe.

# EXEStealth 2.72 - 2.73

Vodjeni ideologijom koju smo usvojili u proslom primeru, na PeXu, pokusacemo da otpakujemo i ExEStealth protektor. Kazem protektor iz razloga sto i ExEStealth detektuje Olly pa cemo morati da iskoristimo HideOlly plugin da bismo zaobisli detekciju Ollyja. Meta se nalazi u folderu Cas10 a zove se Lopke.EXEStealth.exe. Pored ovoga cemo morati da podesimo Ollyjeve Debugging opcije bas ovako:



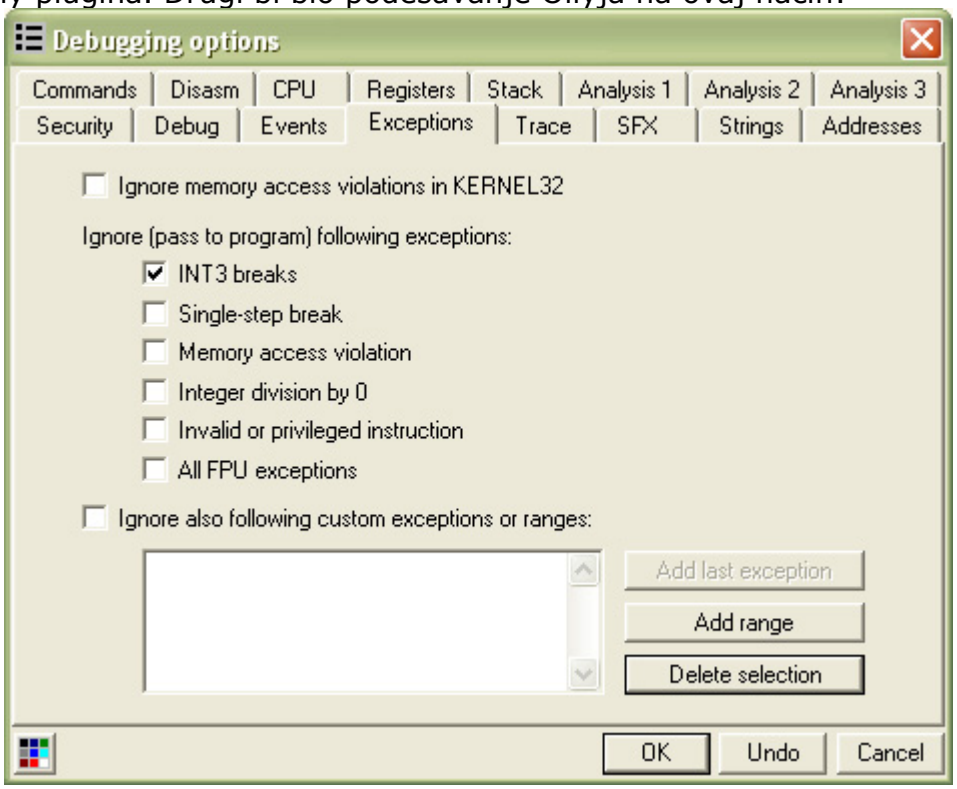

Posle ovoga ostaje nam samo da postavimo i Memory breakpoint on access na glavnu .CODE sekciju mete. Kada ovo uradimo pritisnucemo F9 u nadi da ce nas to odvesti do samog OEPa, ali to se nece desiti. Umesto OEPa zavrsicemo na jednoj LODS adresi koja je zaduzena za ucitavanje sadrzaja adrese 00401000 u EAX registar. A ako prodjete ostatak koda do adrese LOOPD doci cete i do jedne STOS komande koja je zaduzena za snimanje novoizracunate vrednosti registra AL na poziciju 00401000. Posto se ovaj deo koda izvrsava vise puta postavicemo jedan breakpoint na adresu 0040548E koja sadrzi RET komandu, posle cega moramo da uklonimo nas memorijski breakpoint. Pritiskom na F9 zavrsicemo na nasem breakpointu. Posle ovoga cemo ukloniti ovaj breakpoint pritiskom na F2 i ponovo cemo postaviti memoriski breakpoint na .CODE sekciju. I konacno posle pritiska na F9 naci cemo se na pravom OEPu koji izgleda ovako:

```
00401000   ./EB 13            JMP SHORT Lopke_EX.00401015
00401002   .|2D 2D 20 52 4>   ASCII "-- REGISTERED! -"
00401012   .|2D 00            ASCII "-",0
00401014    |00               DB 00
00401015   > \6A 00           PUSH 0
00401017   .  E8 8E030000     CALL Lopke_EX.004013AA
0040101C   .  A3 00304000     MOV DWORD PTR DS:[403000],EAX
00401021   .  EB 03           JMP SHORT Lopke_EX.00401026
00401023   .  41 31 00        ASCII "A1",0
00401026   >  6A 00           PUSH 0
00401028   .  68 45104000     PUSH Lopke_EX.00401045
0040102D   .  6A 00           PUSH 0
0040102F   .  68 23104000     PUSH Lopke_EX.00401023          ; ASCII "A1"
00401034   .  FF35 00304000   PUSH DWORD PTR DS:[403000]
0040103A   .  E8 8F030000     CALL Lopke_EX.004013CE
0040103F   .  50              PUSH EAX
00401040   .  E8 5F030000     CALL Lopke_EX.004013A4
00401045   .  55              PUSH EBP
00401046   .  8BEC            MOV EBP,ESP
00401048   .  83C4 F4         ADD ESP,-0C
```

Sada nam ostaje samo da uradimo dump pomocu LordPEa i da popravimo importe pomocu ImpReca. Kada budete popravljali importe moracete da uradite jedan level1 trace da biste ih sve nasli i to je to. Izgleda da ExEStealth i nije toliko “nevidljiv”...

# ARM Protector 0.1

Jos jedan jako zanimljiv paker koji se moze raspakovati bez stepovanja kroz kod, a u stvari i potrebno je da se to ovako uradi, jer program detektuje stepping kroz kod. Ali i ovo moze da se resi sa relativno malo koraka. Meta pakovana ovim pakerom se nalazi u folderu Cas10 a zove se unpackme3.ARMProtector.exe. Prvi bi bio da sakrijemo Olly od ARMa pomocu HideOlly plugina. Drugi bi bio podesavanje Ollyja na ovaj nacin:




Posto packer OEP izgleda ovako:

```
00406000 >/$  E8 04000000    CALL unpackme.00406009
00406005  |.  8360 EB 0C      AND DWORD PTR DS:[EAX-15],0C
00406009  |$  5D              POP EBP
0040600A  |.  EB 05          JMP SHORT unpackme.00406011
```

ostaje nam da pritisnemo F9 i da dodjemo do sledeceg Exceptiona koji izgleda ovako:

```
00407249     8B          DB 8B
0040724A     00          DB 00
0040724B     EB          DB EB
```

Posto treba da predjemo preko ovog exceptiona pritisnucemo SHIFT+F8, a onda F8 sve dok se ne izvrsi sledeca linija ASM koda u ntdll.dll fajlu

```
77FB4DC6    E8 480BFCFF      CALL ntdll.ZwContinue
```

Posle cega cemo se naci na OEPu gde mozemo da uradimo dump i popravku importa. Sam OEP izgleda ovako:

```
00401000   .  6A 00          PUSH 0                      ; /pModule = NULL
00401002   .  E8 0D010000    CALL unpackme.00401114      ; \GetModuleHandleA
```

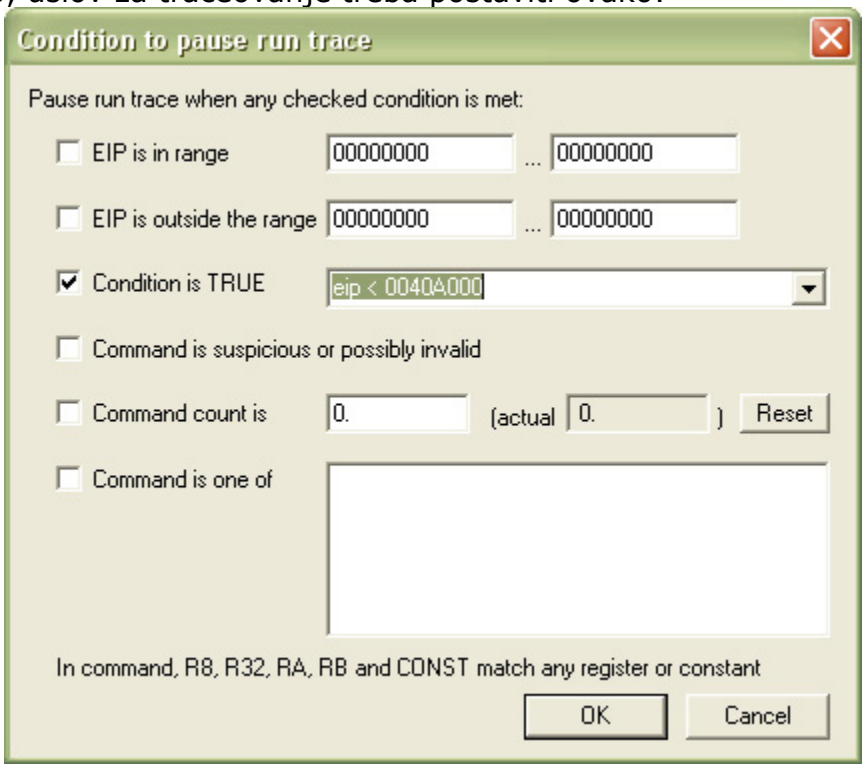
# EXE32Pack 1.3x

Ovo je jedan jako podmukao paker. Kazem podmukao jer on za razliku od vecine ostalih pakera ne saopstava da je detektovao debugger nego podmuklo unistava OEP! Bez obzira na ovaj stvarno podao trik i ova zastita se veoma lako zaobilazi! Meta pakovana ovim pakerom se nalazi u folderu Cas10 a zove se unpack.exe32pack.exe. Prvo cemo kao i uvek iskoristiti OllyDBG plugin HideOlly da bismo izbegli detekciju debuggera. Kada ovo uradimo do samog OEPa cemo doci pomocu obicnog traceovanja. Samo treba da znamo kako i zasto da postavimo uslov za traceovanje.

Posto packer OEP izgleda ovako:

```
0040A00C >  3BC0              CMP EAX,EAX
0040A00E    74 02             JE SHORT unpack_e.0040A012
0040A010    8183 553BC074 0>ADD DWORD PTR DS:[EBX+74C03B55],53838102
0040A01A    3BC9              CMP ECX,ECX
0040A01C    74 01             JE SHORT unpack_e.0040A01F
0040A01E    BC 563BD274       MOV ESP,74D23B56
0040A023    0281 8557E800     ADD AL,BYTE PTR DS:[ECX+E85785]
0040A029    0000              ADD BYTE PTR DS:[EAX],AL
0040A02B    003B              ADD BYTE PTR DS:[EBX],BH
0040A02D    DB                ???                          ; Unknown command
```

a u 99% posto situacija OEP se nalazi na adresama izmedju 00401000 - 00402000, uslov za traceovanje treba postaviti ovako:




Naravno do ovog prozora se stize pomocu menija Debug -> Set condition ili pritiskom na CTRL+T. Kada konacno postavimo uslov za trace ovako ostaje nam da pritisnemo CTRL+F11 dugme ili da selektujemo Debug-Trace into meni. I posle malo vremena cemo se naci na pravom OEPu koji se nalazi na adresi 00401264 na kojoj mozemo da uradimo dump, posle cega kao i uvek sledi popravka importa pomocu ImpRECa.

# PC-Guard 5.0

PC-Guard je jedan veoma zanimljiv protektor. Kazem da je zanimljiv iz razloga sto je njegov autor sa naseg podneblja i zbog cinjenice da ima stvarno zanimljive opcije zastite kroz mrezu i slicno. Definitivno izvrstan protektor i prava meta za nas poduhvat! Ja sam za ovu priliku zastitio jedan moj keygenerator ovim pakerom i stavio ga u folder Cas10 a nazvao unpack.PC-Guard 5.0.exe. Jedini problem sa kojim se mozete sresti prilikom otpakivanja ovog protektora je cinjenica da je fajl zasticen trial verzijom PC-Guarda pa ce se stoga sa vremena na vreme pojavljivati NAG posle kojeg ce se program iskljuciti, takodje se aplikacija moze startovati samo 20 puta, budite obazrivi! Ako se ovo desi jednostavno u Ollyju pritisnite CTRL+F2 da biste restartovali debuggovanu aplikaciju. Iz zelje da i ovaj protektor ne bude opisan kao puka konfiguracija Ollyja resio sam da vam objasnim kako da sami konfigurisete Olly i da prepoznate Exceptione. Pocecemo pritiskom na F9 kako bi smo videli da li ce se program startovati, to jest testiramo da li ima antidebugger zastite. Ali pre nego sto se program oglasi po pitanju antidebugging zastite naisli smo na sledece:

```
004098DE   8913              MOV DWORD PTR DS:[EBX],EDX          ; unpack_P.0040998E
```

a ako pogledamo dole u status bar Ollyja videcemo da se radi o Access violation Exceptionu. Sada cemo otici u Debugging opcije pritiskom na ALT+O i u tabu Exceptions cemo otkaciti checkbox Memory Access violation. Posle ovoga moramo da restartujemo Olly pritiskom na CTRL+F2 i da je ponovo startujemo sa F9. Posto je Olly uspesno presao preko proslog violationa zaustavio se na novom:

```
00409C73   /EB 01            JMP SHORT unpack_P.00409C76
```

koji se zove single step event. I njega cemo dodati u prozoru Debbuging options klikom na Add last Exception i otkacinjanjem checkboxa Ignore also following custom exceptions or ranges. Ponovo sledi clik na CTRL+F2 i F9, posle cega cemo zastati na sledecem Exceptionu:

```
77F767CE   C3                RET
```

A posto je u pitanju INT3 exception i njega cemo otkaciti u prozoru Debugging options. A posle restarta aplikacije i njenog ponovnog starta vidimo program ocigledno ima antiOlly zastitu jer se proces zavrsio pozivom

```
ESI 77F7663E                 ntdll.ZwTerminateProcess
```

Zbog ovoga cemo posle svakog restarta debuggovane aplikacije sakriti Olly pomocu HideOlly plugina. Sada cemo restartovati program, sakriti Olly i postavicemo Memory breakpoint on access na glavnu .CODE sekciju. Posle pritiska na F9 naci cemo se ovde:

```
0041632D   281F              SUB BYTE PTR DS:[EDI],BL
```

Posto ce se program vracati ovde vise puta postavicemo obican breakpoint na JMP komandu ispod LOOPD komande, skinucemo memoriski breakpoint i pritisnucemo F9 da dodjemo do naseg obicnog breakpointa. Kada smo dosli do njega skinucemo ga pritiskom na F2 i vraticemo memorijski breakpoint na glavnu sekciju. Posle pritiska dugmeta F9 naci cemo se na pravom OEPu, koji se nalazi na adresi 00401264 (*prvo ga analizirajte sa CTRL+A*) gde mozemo da uradimo dump i popravimo importe pomocu ImpRECa.

# yoda Crypter 1.3 - yC

yC je jedan od kriptera/protektora koji je napisao y0da, takodje zasluzan za alat, koji ste do sada koristili na stotine puta, LordPe. U folderu Cas10 se nalazi meta unpackme40.yC13.exe zasticena ovim kripterom. Otvorite metu pomocu Ollyja i pre nego sto pocnete otpakivanje sakrijte Olly pomocu HideOlly plugina. Kada ste ovo uradili mozete da pokusate da pokrenete program klikom na F9. Zavrsicete ovde:

```
00405982   0000              ADD BYTE PTR DS:[EAX],AL
```

Posto je ovo Access violation otkacicemo odgovarajuci checkbox u dijalogu Debugging options (*ALT+O*) i restartovacemo Olly i postavicemo Memory breakpoint na glavnu .CODE sekciju. Ne zaboravite da ponovo ukljucite HideOlly plugin! Ponovnim klikom na F9 zavrsavamo ovde:

```
00405456   AC                LODS BYTE PTR DS:[ESI]
```

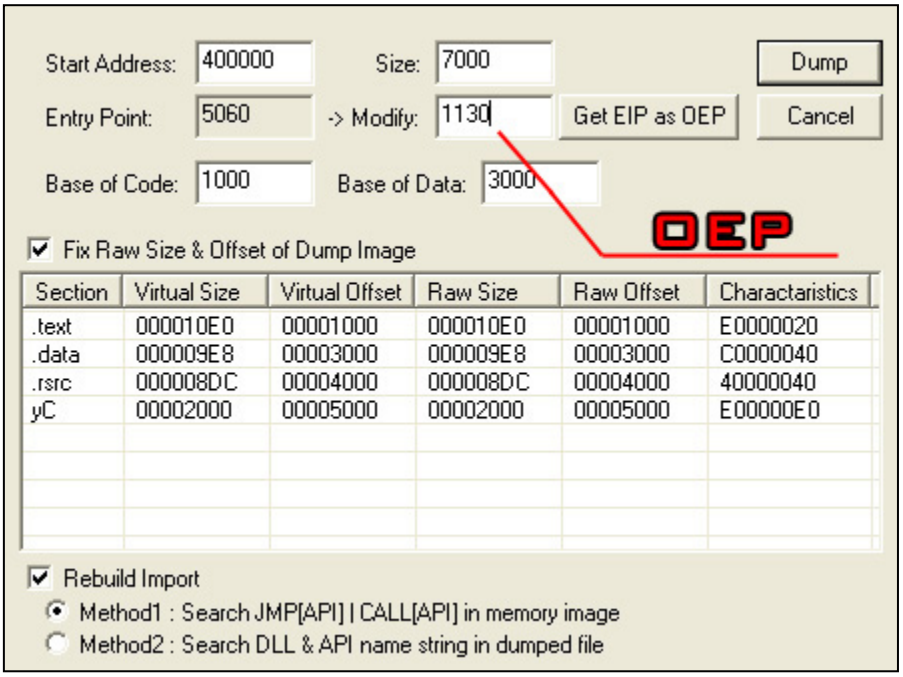
Posto je ovo komanda za ucitavanje bajta iz .CODE sekcije kliknucemo ponovo na Run i zavrsicemo ovde:

```
00405487   AA                STOS BYTE PTR ES:[EDI]
00405488  ^ E2 CC            LOOPD SHORT unpackme.00405456
0040548A   C3                RET
```

Posto se nalazimo u LOOPu koji otpakuje .CODE sekciju postavicemo breakpoint na RET komandu, uklonicemo memorijski braekpoint i pritisnucemo F9 da bismo stigli do naseg obicnog breakpointa. Kada stignemo do RET komande uklonicemo breakpoint klikom na F2. Posto je ocigledno da se pakerski kod nalazi na adresama od 00405000 postavicemo uslov za trazenje (*condition is true*) EIP < 00405000 u prozoru Condition to pause Run trace (*CTRL+T*). Kada ovo uradimo pokrenucemo trace klikom na CTRL+F11 kako bismo stigli do OEPa. I posle malo cekanja dosli smo do:

```
00401135   E8                DB E8
```

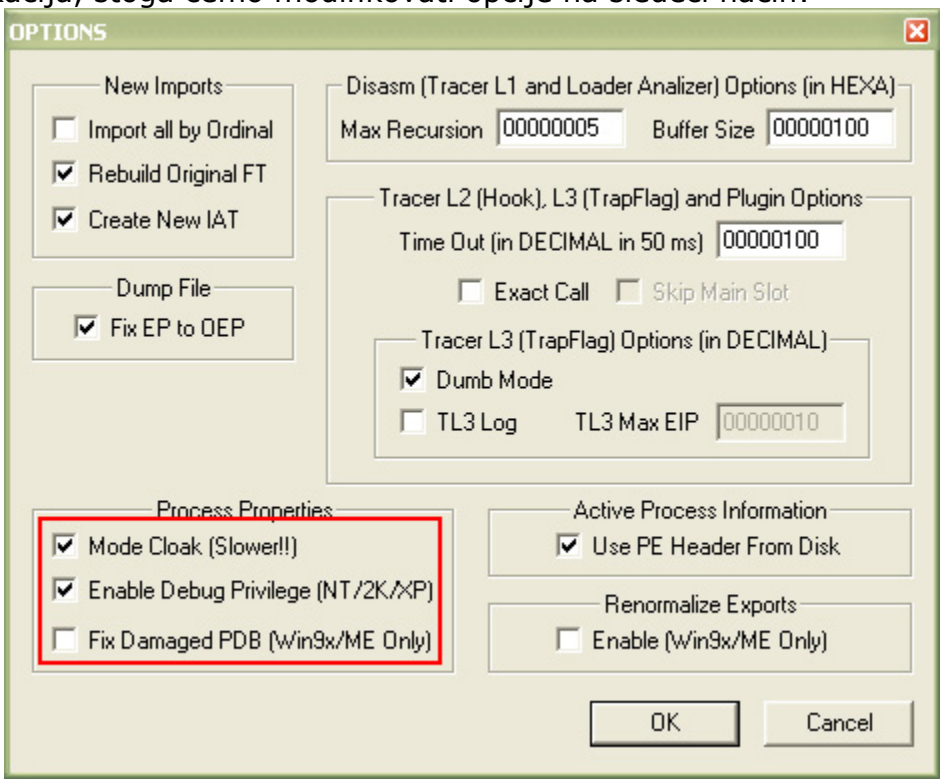
Ovaj kod nije potrebno analizirati jer on stvarno predstavlja OEP standardnog VB 6.0 programa. Sada cemo izvrsiti dump i popravku importa pomocu Olly Dump plugina. Ovde samo treba da primetite da smo morali da izmenimo OEP na njegovu pravu vrednost jer je yC progutao prvih par bajtova. *Kako znamo da je ovo pravi OEP?*




*Jednostavno cemo pogledati neki obican VB fajl i videcemo njegov OEP.*

# SVKP 1.3x

Jedan od jako dobrih protektora je SKVP, Pavola Cervena. Pre nego sto pocnemo moram da napomenem da je dosadasnja konfiguracija ImpRECa pogresna. Ne, naravno da nije totalno pogresna nego joj je potrebna mala modifikacija, stoga cemo modifikovati opcije na sledeci nacin:




Ovo je potrebno jer drugacije program ne bi mogao da uradi popravku importa, zbog nemogucnosti da nadje proces. Meta se nalazi u folderu Cas10 a zove se unpack.SKVP13.exe. Ovu metu cemo otvoriti pomocu Ollyja. Sam packer OEP izgleda ovako:

```
0040A000 >  60              PUSHAD
0040A001    E8 00000000     CALL unpack_S.0040A006
```

Uci cemo u prvi CALL sa F7 i izvrsavacemo kod sa F7 sve dok ne dodjemo do:

```
0040A026   /75 67           JNZ SHORT unpack_S.0040A08F
```

Sada cemo podesiti Debugging opcije (*ALT+O*) i ukljucicemo INT 3 i Memory Access Violation, posle cega cemo pritisnuti F9 i zbog cega ce se pojaviti SKVP NAG. Posle zatvaranja tog NAGa naci cemo se na sledecem exceptionu:

```
00A6137F   6285 0E0B0000    BOUND EAX,QWORD PTR SS:[EBP+B0E]
```

Posto ce se i ovaj exception pojaviti vise puta ponovo cemo podesiti Debugging opcije i dodacemo ovaj exception klikom na Add last exception. Posto se nalazimo u delu koda koji sluzi za otpakivanje .CODE sekcije postavicemo jedan memorijski breakpoint (*on access*) na nju. Kada ovo uspesno zavrsite kliknite 2x na F9 da biste stigli do prvog memorijskog breakpointa. On se nalazi ovde:

```
00A8B6B1   8A06             MOV AL,BYTE PTR DS:[ESI]
00A8B6B3   46               INC ESI
00A8B6B4   47               INC EDI
```

Posto ce se ovaj breakpoint pojavljivati vise puta najbolje je da uklonimo memorijski breakpoint on access i postavimo obican breakpoint na adresu:

```
00A8B6E1   C2 1400          RET 14
```

Posto smo uklonili memorijski breakpoint pritisnucemo F9 da dodjemo do breakpointa na adresi 00A8B6E1. Kada se ovo desi mozemo da uklonimo breakpoint sa RET 14 komande i da vratimo memorijski breakpoint na .CODE sekciju, sa tom razlikom sto cemo postaviti ovaj put write breakpoint. Ovo radimo iz razloga jer koliko vidimo glavna .CODE sekcija jos uvek nije u potpunosti otpakovana. Klikom na F9 zavrsavamo ovde:

```
00A860DD   F3:A5            REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS>
00A860DF   03CA             ADD ECX,EDX
00A860E1   F3:A4            REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[>
00A860E3   5A               POP EDX
```

Posto se i REP komanda izvrsava veci broj puta ponovo cemo ukloniti memorijski breakpoint sa sekcije i pritisnucemo F8 4x da bismo izvrsili obe REP komande. I konacno mozemo da pogledamo stanje .CODE sekcije 2x klikom na nju u Memory window prozoru. Videcemo da je cela sekcija uspesno otpakovana! Da ne bismo izvrsavali kod sve dok ne dodjemo do OEPa uradicemo to pomocu jos jednog memorijskog breakpointa (*access*) na glavnu sekciju. Posle klika na F9 zavrsicemo na pravom OEPu koji posle analize izgleda ovako:

```
00401264  /.  55            PUSH EBP
00401265  |.  8BEC          MOV EBP,ESP
```

Sto je ocigledno VC++ OEP. Ovde mozemo uraditi dump pomocu LordPEa. Ali pre nego sto uradite dump u LordPEu u istom meniju kao Dump full izaberite komandu Correct image size, pa tek onda izvrsite Dump.

Popravka importa ce biti malo teza nego obicno iz razloga sto ImpREC prvo treba podesiti tako da on moze da pristupi standardnom "kacenju" na debuggovani proces gde cemo uraditi popravku importa. Posto smo prvi deo vec uradili ostaje nam da uradimo i drugi to jest da popravimo sve importe. Naravno kao i uvek uradicemo Trace Level 1 disassemble posle cega ce ostati samo jedan nepoznat import u kernel32.dll fajlu. To ce u tree fajlu izgledati ovako:

```
0        00002008       ?          0000    00AF33DD
1        0000200C       kernel32.dll       01A6    GetStartupInfoA
```

Bez obzira na ovaj pokvaren mi cemo 2x kliknuti na njega i dodacemo bilo koji import iz kernel32.dll fajla, recimo:

```
1        00002008       kernel32.dll       0002    AddAtomA
1        0000200C       kernel32.dll       01A6    GetStartupInfoA
```

Kada uradimo Fix dump otvoricemo dumpovan fajl pomocu Ollyja i pocecemo da traceujemo kroz njega sa F8 sve dok ne dodjemo do tacke rusenja:

```
00401368  |.  FF15 0C204000   CALL NEAR DWORD PTR DS:[40200C]
```

Sada kada znamo gde se meta srusila jednostavno cemo otvoriti neki drugi VC++ fajl i prepisacemo koji se to import na tom mestu poziva. Otvorite ...\Casovi\Cas03\keygenme1.exe i jednostavno uporedite VC++ import callove i videcete da je tacan import:

```
1        00002008       kernel32.dll       016F    GetModuleHandleA
1        0000200C       kernel32.dll       01A6    GetStartupInfoA
```

**NAPOMENA:** Ovaj metod je moguce koristiti samo ako je u pitanju neki od poziva koji se nalaze na standardnom compiler OEPu. Ako je u pitanju neki drugi deo koda moracete da analizirate PUSH komande koje prethode samom CALLu, medjutim SKVP nece nikad progutati vise od tri-cetiri importa.

# eXpressor1.2.0

Ovo je jedan prilicno obican paker ali sam odlucio da i njega stavim u knjigu zbog par njegovih specificnosti. Naime ovaj paker ima i standardno shareware upozorenje ali i zanimljiv nacin odlaska na OEP. Kazem zanimljiv ali je on stvarno vise nego standardan, samo sto cu ga ovaj put detaljno objasniti. Meta za ovaj deo poglavlja se nalazi u folderu Cas10 a zove se crackme.eXPressor120.exe. Ovaj fajl cemo otvoriti pomocu Ollyja i na samom OEPu cemo videti sledece:

```
00407638 >/$  55              PUSH EBP
00407639  |.  8BEC            MOV EBP,ESP
0040763B  |.  81EC D4010000  SUB ESP,1D4
00407641  |.  53              PUSH EBX
```

Posto znamo da nasa meta prikazuje NAG izvrsavacemo je sa F8 sve dok ne stignemo do sledeceg koda koji sluzi za prikazivanje NAGa:

```
004076D6  |.  6A 30           PUSH 30
004076D8  |.  68 C4724000     PUSH crackmee.004072C4     ; |Title = "Nfo"
004076DD  |.  68 C8724000     PUSH crackmee.004072C8       "This program was packed...."
004076E2  |.  6A 00           PUSH 0
004076E4  |.  FF15 D8744000  CALL NEAR DWORD PTR DS:[<&USER32.MessageBoxA>]
```

Ovaj NAG cemo ukloniti ili na standardan nacin NOPovanjem ili na laksi nacin tako sto cemo umesto PUSH 30 komande uneti PUSH FF komandu. Naravno ova druga izmena se radi pomocu Binary Edit prozora u Ollyju.

Sledece sto je potrebno da uradimo je da pronadjemo mesto sa kojeg se izlazi iz eXpressor koda i posle cijeg izvrsavanja se dolazi na pravi OEP. Da bismo ovo uradili potrebno je da postavimo memorijski breakpoint on access na glavnu .data sekciju (*ovo je .CODE sekcija samo sto se ovde zove .data*) i pritisnemo F9 posle cega smo ovde:

```
00407764  |.  F3:A5          REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
00407766  |.  8BC8            MOV ECX,EAX
```

Posto se ovde ocigledno upisuju podaci u glavnu .code sekciju, ovaj deo cemo preskociti i pronaci cemo deo koji izgleda kao skok ka OEPu. Dakle skrolovacemo na dole sve dok ne budemo videli ovo:

```
00407D99  |.  A1 B4864000    MOV EAX,DWORD PTR DS:[4086B4]    <- Load Image base
00407D9E  |.  0305 80704000  ADD EAX,DWORD PTR DS:[407080]    <- Load OEP base
00407DA4  |.  8985 54FEFFFF  MOV DWORD PTR SS:[EBP-1AC],EAX   /* Nepotrebno
00407DAA  |>  8B85 54FEFFFF  MOV EAX,DWORD PTR SS:[EBP-1AC]   /* Nepotrebno
00407DB0  |.  5F              POP EDI
00407DB1  |.  5E              POP ESI
00407DB2  |.  5B              POP EBX
00407DB3  |.  81C4 D4010000  ADD ESP,1D4
00407DB9  |.  5D              POP EBP
00407DBA  |.  FFE0            JMP NEAR EAX                     <- JMP to OEP
```

Posto smo vise puta videli ovakav nacin odlaska ka OEPu postavicemo jedan breakpoint odmah ispod gornjeg loopa, to jest postavicemo breakpoint na adresu 00407D99. Kada posle skidanja memoriskog breakpointa sa glavne code sekcije pritisnemo F9 zavrsicemo na novom breakpointu. Ono sto bi trebalo da primetite da se podaci kao sto su ImageBase i base OEP nalazu uvek u fajlu u nezasticenom obliku na staticnim adresama, ciji je sadrzaj DS:[004086B4]=00400000 i DS:[00407080]=000012C0. Iz ovoga sledi da neki pakeri mogu sadrzati ove nama korisne podatke na staticnim lokacijama u svom kodu sto nam je od izuzetne vaznosti kada pravimo inline patchere o kojima ce biti reci u sledecem poglavlju.

# JDPack 1.x / JDProtect 0.9

Ovaj paker / protektor se moze naci u dve verzije i pod dva imena ali se u oba slucaja kod uopste ne razlikuje, naravno ako se izuzme deo koda koji predstavlja deo koda koji se direktno koristi za odlazak na OEP otpakovanog programa. Ova razlika u JDProtect nije velika i predstavlja samo varijaciju na OEP jump iz JDPacka. Ono sto je bitno kada otpakujemo ovaj paker je da uocimo cesto koriscenu rutinu za dekriptovanje odredjenog dela koda. Ovaj deo koda je standardan i sastoji se ili od dve MOV/MOVS komande ili para LODS i STOS komande. Ove komande ne rade nista drugo osim ucitavanja i snimanja bajtova na odredjene adrese. Ono sto je bitno je deo koda koji se izvrsava izmedju ove dve komande a odnosi se na dekriptovanje nekog dela koda. Ovaj postupak bi izgledao ovako:

- Ucitavanje bajta sa neke adrese pomocu komande MOV ili LODS
- Dekripcija bajta pomocu niza komandi
- Snimanje dekriptovanog bajta na lokaciju sa koje je iscitan pomocu komandi MOV ili STOS.

Ovakav kod se lako identifikuje u kodu JDProtectora jer je koristena obicna petlja koja dekriptuje odredjeni niz bajtova. Posto se petlje u ASMu prave ili pomocu uslovnih skokova skokova ili pomocu LOOPD komande, ovaj deo koda se veoma lako identifikuje:

```
00405043   8A85 383C4000    MOV AL,BYTE PTR SS:[EBP+403C38]
00405049   8A1E              MOV BL,BYTE PTR DS:[ESI]
0040504B   32C3              XOR AL,BL
0040504D   8806              MOV BYTE PTR DS:[ESI],AL
0040504F   889D 383C4000     MOV BYTE PTR SS:[EBP+403C38],BL
00405055   46                INC ESI
00405056  ^ E2 EB            LOOPD SHORT crackme_.00405043
```

Posto ovaj deo koda sluzi za direktnu dekripciju koda ispod, drzacemo F8 da bismo dekriptovali prvih par komandi ispod ovog loopa. Kada se dekriptuje prva komanda mozemo da postavimo breakpoint na tu novu dekriptovanu adresu i pritisnemo F9 kako bismo dosli do nje. Prva sledeca adresa je:

```
00405058   9C                PUSHFD
00405059   58                POP EAX
```

Sledece sto treba da uradimo je da nadjemo skok ka OEPu. Ovde cemo iskoristiti mali trik. Naime posto program prikazuje shareware NAG u takozvanom Warning modu (*prikazuje tu ikonu*) parametar koji se mora proslediti MessageBoxA APiju je PUSH 30. Kada ovo uradimo pomocu CTRL+L pronacicemo drugo pojavljivanje ove komande posto se prvo ocigledno odnosi na prijavljivanje gresaka  u procesu otpakivanja. Dakle, ovde smo:

```
0040550F   6A 30             PUSH 30
00405511   8D9D 58394000     LEA EBX,DWORD PTR SS:[EBP+403958]
```

Primeticete da se ove PUSH komande odnose na prikazivanje NAGa, a sledeci kod na pripremu i izvrsavanje skoka ka OEPu. Ovde se desavaju sledece analizirane operacije:

```
00405527   8B95 3D3C4000    MOV EDX,DWORD PTR SS:[EBP+403C3D]   /*Load ImageBase
0040552D   8B85 F8384000    MOV EAX,DWORD PTR SS:[EBP+4038F8]   /*Load OEP base
00405533   03C2              ADD EAX,EDX                         /* Get OEP RVA
00405535   894424 1C         MOV DWORD PTR SS:[ESP+1C],EAX
00405539   61                POPAD
0040553A   50                PUSH EAX                            /* PUSH OEP
0040553B   C3                RET                                 /* JMP to OEP
```

# ApOx Crypt 0.01 - aC

Posto pretpostavljam da ste naucili kako se otpakuje vecina pretnodnih pakera, sada cemo preci na otpakivanje mog licnog kriptera. Ova varijacija istog kriptera je samo ogoljena verzija onoga sto vas ceka na nightmare levelu ove knjige. Stoga bi otpakivanje aC bar na ovom nivou trebalo da bude setnja u parku :) Prvo cemo skenirati metu unpackme#1.aC.exe sa PeIDom da vidimo sta on ima da kaze o njoj. Posto mog kriptera nema u PeIDeovoj biblioteci on ce cutati po pitanju identifikacije ovoga kriptera. Nema veze neka nas to ne obeshrabri probacemo sa otpakivanje aC-a pomocu PeID generickog unpackera. Da vidimo, detect OEP - OK :), unpack OK, rebuild IAT ne zato sto IAT nije kriptovan i kada startujemo "raspakovani" fajl, crash. Naime ovo se desilo zato sto PeID nije detektovao tacan OEP. Izgleda da cemo morati ovaj kripter da odpakujemo rucno, otvorite Olly i pogledajte kako izgleda "OEP":

```
00401000 >/$  60                  PUSHAD
00401001  |.  E8 E3000000         CALL unpackme.004010E9
00401006  \.  C3                  RET
```

Nista specijalno, pa cemo pritiskom 2x na F7 uci u prvi CALL da vidimo sta se tu desava. Kada to uradimo naci cemo se ovde:

```
004010E9  /$  B8 F5104000         MOV EAX,unpackme.004010F5
004010EE  |.  50                  PUSH EAX
004010EF  |.  E8 A7FFFFFF         CALL unpackme.0040109B
004010F4  \.  C3                  RET
```

Ako pogledamo sada prvu liniju videcemo da se u nju smesta adresa 004010F5 ili adresa za koju je PeID mislio da je OEP. Ovo je izgleda ispala moja licna antigeneric zastita. Posto i ovo nije nista specijalno uci cemo u sledeci CALL sa F7. To ce nas dovesti ovde:

```
0040109B  /$  50                  PUSH EAX                ; unpackme.004010F5
0040109C  |.  8BD8                MOV EBX,EAX
0040109E  |.  B9 54010000         MOV ECX,154
004010A3  |>  8033 44             /XOR BYTE PTR DS:[EBX],44
004010A6  |.  83E9 01             |SUB ECX,1
004010A9  |.  43                  |INC EBX
004010AA  |.  83F9 00             |CMP ECX,0
004010AD  |.^ 75 F4               \JNZ SHORT unpackme.004010A3
004010AF  |.  50                  PUSH EAX
004010B0  |.  E8 08000000         CALL unpackme.004010BD
004010B5  |.  50                  PUSH EAX
004010B6  |.  E8 7EFFFFFF         CALL unpackme.00401039
004010BB  |.  58                  POP EAX
004010BC  \.  C3                  RET
```

Ovo izgleda malo ozbiljnije, ali opet nista komplikovano. Jednostavno crackme uzima adresu 004010F5 kao pocetnu i xoruje sledecih 154h bajtova sa vrednoscu 44h. Posle ovog loopa se poziva jos jedan CALL sa istim parametrom EAX. Da ne bismo izvrsavali loop 154h puta postavicemo breakpoint na adresu 00401AF odnosno na PUSH EAX komandu. Kada ovo uradimo pritisnucemo F9 da bismo dosli do naseg breakpointa. Sada cemo sa F7 uci u sledeci CALL i naci cemo se ovde:

```
004010BD  /$  50                  PUSH EAX                ; unpackme.004010F5
004010BE  |.  BB 07104000         MOV EBX,unpackme.00401007
004010C3  |.  B9 7F000000         MOV ECX,7F
004010C8  |>  8033 07             /XOR BYTE PTR DS:[EBX],7
```

```
004010CB  |.  83E9 01               |SUB ECX,1
004010CE  |.  43                    |INC EBX
004010CF  |.  83F9 00               |CMP ECX,0
004010D2  |.^ 75 F4                 \JNZ SHORT unpackme.004010C8
004010D4  |.  8BD8                  MOV EBX,EAX
004010D6  |.  B9 54010000           MOV ECX,154
004010DB  |>  8033 11               XOR BYTE PTR DS:[EBX],11
004010DE  |.  83E9 01               SUB ECX,1
004010E1  |.  43                    INC EBX
004010E2  |.  83F9 00               CMP ECX,0
004010E5  |.^ 75 F4                 JNZ SHORT unpackme.004010DB
004010E7  |.  58                    POP EAX
004010E8  \.  C3                    RET
```

Kao sto vidimo u EBX se smesta adresa 00401007 i 7F bajtova od te adrese se xoruje sa 7h, posle cega se ponovo xoruje 154h bajtova od adrese 004010F5 sa 11h. Posto nema potrebe da prolazimo kroz sve ove loopove pritisnucemo CTRL + F9 sto ce izvrsiti ovaj CALL i vratice nas na sledecu adresu posle izvrsenja RET komande:

```
004010B5  |.  50                    PUSH EAX                 ;  unpackme.004010F5
004010B6  |.  E8 7EFFFFFF           CALL unpackme.00401039
004010BB  |.  58                    POP EAX
004010BC  \.  C3                    RET
```

Kao sto vidimo ovo nas je odvelo do druge CALL komande u gornjem CALLu. Sada cemo sa F7 uci i u ovaj drugi CALL:

```
00401007   . /EB 27                JMP SHORT unpackme.00401030
00401009   . |43 72 43 20 6>       ASCII "CrC of this file"
00401019   . |20 68 61 73 2>       ASCII " has been modifi"
00401029   . |65 64 20 21 2>       ASCII "ed !!!",0
00401030   > \EB 07                JMP SHORT unpackme.00401039
00401032   .  45 72 72 6F 7>       ASCII "Error:",0
00401039   $  50                    PUSH EAX                 ;  unpackme.004010F5
0040103A   .  8BD8                  MOV EBX,EAX
0040103C   .  B9 54010000           MOV ECX,154
00401041   .  BA 00000000           MOV EDX,0
00401046   >  0313                  ADD EDX,DWORD PTR DS:[EBX]
00401048   .  83E9 01               SUB ECX,1
0040104B   .  43                    INC EBX
0040104C   .  83F9 00               CMP ECX,0
0040104F   .^ 75 F5                 JNZ SHORT unpackme.00401046
00401051   .  B8 4A124000           MOV EAX,unpackme.0040124A
00401056   .  BE 80124000           MOV ESI,unpackme.00401280
0040105B   .  50                    PUSH EAX
0040105C   .  56                    PUSH ESI
0040105D   .  E8 28000000           CALL unpackme.0040108A
00401062   .  81FA B08DEB31         CMP EDX,31EB8DB0
00401068   .  74 19                 JE SHORT unpackme.00401083
0040106A   .  6A 30                 PUSH 30
0040106C   .  68 32104000           PUSH 00401032  "Error:"
00401071   .  68 09104000           PUSH 00401009  "CrC of this file has been modified !!!"
00401076   .  6A 00                 PUSH 0
00401078   .  E8 E5010000           CALL unpackme.00401262
0040107D   .  50                    PUSH EAX
0040107E   .  E8 F1010000           CALL unpackme.00401274
00401083   >  E9 96010000           JMP unpackme.0040121E
00401088   .  58                    POP EAX
00401089   .  C3                    RET
```

Ovo je malo duzi CALL ali sam ga ja podelio na celine kako biste vi lakse shvatili sta se ovde desava. Analizu cemo raditi po obelezenim bojama.
Zeleni deo predstavlja samo poruke koje ce se prikazati ako smo modifikovali ovaj fajl, a posto ovo nismo uradili o ovome necemo brinuti.

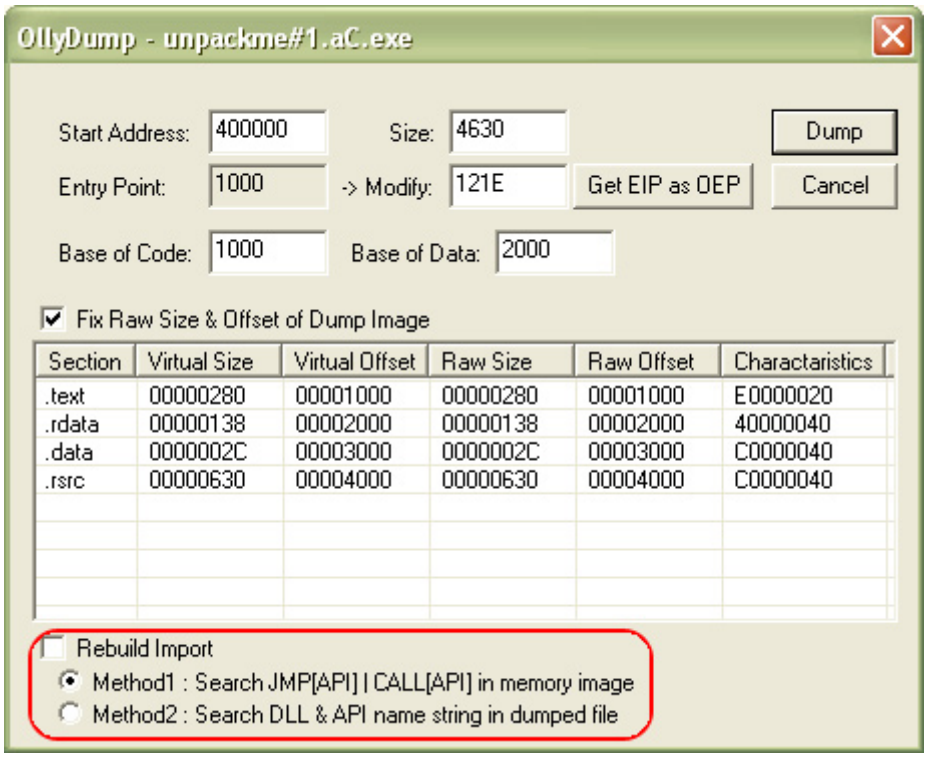
Narandzasti deo predstavlja kod za racunanje "CRCa" dela koda koji pocinje na adresi 004010F5 a dugacak je 154h i predstavlja glavni deo koda. Ni ovo nam nije bitno posto nismo modifikovali ovaj deo koda nego se bavimo samo otpakivanjem. Roze deo koda predstavlja novi CALL sa parametrima 0040124A i 00401280 a izgleda ovako:

```
...
00401090  |>  8033 17              /XOR BYTE PTR DS:[EBX],17
00401093  |.  43                   |INC EBX
00401094  |.  3BD9                 |CMP EBX,ECX
00401096  |.^ 75 F8                \JNZ SHORT unpackme.00401090
...
```

Ako analiziramo malo ovaj kod videcemo da se adrese od 0040124A do 00401280 xoruju sa 17h i da taj deo koda predstavljaju redirekcije ka API funkcijama.
Svetlo plavi deo koda poredi EDX sa vrednoscu tacnog CRCa sekcije od 004010F5 i dugacke 154h bajtova. Ako su CRC vrednosti iste onda se skace na tamno plavi deo koda koji vodi ovde:

```
0040121E > $  6A 00                PUSH 0
...
0040123E   .  E8 0D000000          CALL <JMP.&user32.DialogBoxParamA>
```

Odnosno na pravi OEP. Vidimo da se ovde poziva API funkcija za prikazivanje dijaloga - DialogBoxParamA, zbog cega cemo ovde uraditi dump pomocu OllyDMG plugina. Podesite ovaj plugin ovako:




zato sto IAT tabela nije kriptovana nego je kriptovan samo deo fajla. Neophodno je da razumete kako smo dosli do OEPa ovog kriptera kako biste bili u stanju da nadjete OEP bilo kojeg kriptera ili pakera, a ovo ce nam trebati kasnije u dvanaestom poglavlju zato predlazem da pazljivo procitate ovaj postupak.

# Patching it...

Ovo je veoma bitno poglavlje vezano za cracking jer obradjuje vecinu trenutno koriscenih nacina za modifikaciju gotovih exe fajlova. Ovde ce biti objasnjeni nacini funkcionisanja i pravljenja file patchera i memory patchera. Takodje ce biti objasnjeni i principi patchovanja zapakovanih exe fajlova.

## + "Hard" patchers

Fizicki patchevi su patchevi koji se primenjuju direktno na fajlove tako da kod samog patchovanog fajla on biva modifikovan fizicki na disku. Ova vrsta patcheva ima svoje podvrste koje se primenjuju u zavisnosti od slucaja koji se reversuje. Najcesce podvrste fizickih patchera su:

- Obicni fajl patcheri
- Patcheri koji podrzavaju vise verzija istog programa
- Seek and Replace patcheri koji traze odredjene paterne u fajlovima koje patchuju i zamenjuju ih novim bajtovima. Ovi paterni su u stvari nizovi bajtova koji mogu sadrzati i joker karaktere koji zamenjuju bilo koji bajt.

## + Registry patchers

Ovo je vrsta patchera koja vrsi manipulaciju nad sistemskim registryjem. Ova vrsta patchera moze biti dosta kompleksnija od obicnih .reg fajlova. Tako na primer ovi patcheri mogu vrsiti backup odredjenih kljuceva, mogu raditi periodicno brisanje i / ili modifikaciju celih grana kljuceva. Jedina mana ovih patchera je to sto se najcesce prave posebno za svaku aplikaciju. Ova vrsta patchera nema svoje podvrste.

## + Memory patchers

Posebna vrsta patchera koja se bavi modifikacijom radne (RAM) memorije. Ovi patchevi su korisni u slucajevim kada nije moguce napraviti obican "hard" patch. Do ovakve situacije dovodi pakovanje "meta" eksternim pakerima ili enkripcijom sadrzaja fajla. Tada se prave takozvani loaderi koji startuju program i posle otpakivanja programa ili njegove dekripcije u memoriji vrse patchovanje dekriptovanih / otpakovanih memorijskih adresa. Ovi patcheri mogu biti podeljeni u tri podvrste:

- Game traineri
- Obicni memorijski patcheri (*u stvari isto sto i game traineri*)
- Loaderi koji mogu biti konfigurisani tako da vrse patchovanje posle odredjenog broja sekundi / milisekundi, oni takodje mogu pauzirati procese koje patchuju da bi posle patchovanja nastavili sa radom procesa.

Ova sistematizacija je uradjena iz razloga sto cemo dalje kroz ovo poglavlje biti suoceni sa razlicitim problemima koje cemo morati da resimo. Dalje kroz poglavlje cu vam pokazati kako se patchuju razlicite vrste pakera i kako mozete modifikovati memoriju.

# Inline patching: UPX 0.8x-1.9x

Kao sto je vec objasnjeno loaderi se koriste kada zelimo da napravimo patch za pakovane ili kriptovane programe. Njih je veoma lako napraviti ali posto u vecini slucajeva nisu pouzdani koristicemo klasicne patcheve na pakovanim programima. Zvuci li interesantno? Program specijalno pisan za potrebe ovog poglavlja se nalazi u folderu Cas11 a zove se NAG.exe. Otvoricemo ovaj fajl pomocu Ollyja da bismo nasli mesto koje treba da patchujemo. Da bismo nasli mesto koje treba da patchujemo, prvo cemo otpakovati program i naci bajt koji zelimo da patchujemo. Sam NAG se nalazi ovde:

```
00407F1F  /74 19            JE SHORT NAG_exe_.00407F3A
00407F21  |6A 40            PUSH 40
00407F23  |68 507F4000      PUSH NAG_exe_.00407F50          ; ASCII "NAG:"
00407F28  |68 587F4000      PUSH NAG_exe_.00407F58          ; ASCII "Ovo je NAG screen
koji treba da ubijete !!!"
00407F2D  |53               PUSH EBX
00407F2E  |E8 B5C6FFFF      CALL <JMP.&USER32.MessageBoxA>
```

Ocigledno je da treba samo da patchujemo skok JNZ u JMP, to jest sa 74 u EB. Potrebni podaci za inline patch su adresa 00407F1F i vrednost kojom cemo da patchujemo. Potrebno je samo da nadjemo mesto gde cemo da patchujemo. Ovo moramo da uradimo tik posle sto se program otpakuje. Pre nego sto skoci na sam OEP. Stoga cemo prepraviti skok ispod donje POPAD komande UPX pakera tako da nas preusmeri na kod za patchovanje. Evo kako bi to trebalo da izgleda (*narandzasti su izmenjeni / dodati redovi*)

```
0041291E  > \61             POPAD
0041291F  . EB 14           JMP SHORT NAG2.00412935
00412921    90              NOP
00412922    90              NOP
00412923    90              NOP
00412924    3C294100        DD NAG2.0041293C
00412928    44294100        DD NAG2.00412944
0041292C    08A74000        DD NAG2.0040A708
00412930    00              DB 00
00412931    00              DB 00
00412932    00              DB 00
00412933    00              DB 00
00412934    00              DB 00
00412935    C705 1F7F4000>  MOV DWORD PTR:[407F1F],019EB
0041293F  .- E9 B056FFFF    JMP NAG2.00407FF4
```

Vidite sta se desava. Izmenili smo skok tako da on vodi ka delu za patchovanje, posle cega se sa starim UPX skokom vracamo na OEP. Primeticete oblik komande za patchovanje memorije:

MOV DWORD PTR:[407F1F],019EB

Mislim da je jasno sta ovde radimo. Na adresu 00407F1F cemo ubaciti bajtove EB19. Primetite samo da su bajtovi u obrnutom redosledu, prvo ide 19 pa EB, a uvek ce im prethoditi nula ako patchujete pomocu Ollyja. Bitno je da zapamtite da ako nema mesta za dodavanje bajtova odmah iza patcher koda, morate osloboditi mesto negde pomocu nekog Hex Editora. Ovaj postupak je vec objasnjen kod dodavanja funkcija tako da nema potrebe da ovo ponovo objasnjavam.

# Inline patching: nSPack 2.x

UPX je bio lak primer ubacivanja patch koda u kod pakera. Sada cemo patchovati malo slozeniji paker. Kao i do sada prvo moramo da pronadjemo pogodno mesto za ubacivanje naseg inline patcha. Ovo prakticno znaci da moramo da pronadjemo packer exit point i da ga preusmerimo na nas patch kod. Na packer OEPu imamo sledeci kod:

```
0040101B > $- E9 87660000    JMP crackme_.004076A7
00401020     B4              DB B4
00401021     09              DB 09
```

koji cemo ispratiti klikom na F8, posle cega cemo se naci ovde:

```
004076A7   9C                PUSHFD
004076A8   60                PUSHAD
004076A9   E8 00000000       CALL crackme_.004076AE
```

Ovde cemo primeniti STACK trik tako sto cemo kliknuti 2x F8, posle cega cemo postaviti Hardware break-point on dword access na lokaciju ESPa u memory dumpu. Klikom na F9 dolazimo do sledece lokacije:

```
0040791F   61                POPAD
00407920   9D                POPFD
00407921  - E9 9A99FFFF      JMP crackme_.004012C0
```

JMP skok na adresi 00407921 ocigledno vodi ka OEPu i mozemo ga preusmeriti ka nasem patch kodu. Naravno moramo prvo da odredimo mesto u fajlu koje je dovoljno veliko da sadrzi nas patch. Najcesce se mesto za inline patch trazi u PE Headeru fajla odnosno od adrese 00400300. Mi cemo pritisnuti CTRL+G u Ollyju i otici cemo na adresu 00400333 na kojoj se trenutno nalazi dovoljno praznog mesta za ubacivanje naseg patcha. Naravno ovo mesto i nije bas totalno prazno ali ovaj kod se odnosi na glavnu .exe ikonu tako da nema veze da li ce nas patch kod biti ovde ili ne. Ubacicemo sledeci kod:

```
00400333   C605 35104000 F> MOV BYTE PTR DS:[401035],0FF
0040033A   C605 8A104000 0>MOV BYTE PTR DS:[40108A],0
00400341   68 C0124000      PUSH inline_n.004012C0
00400346   C3                RET
```

Ovo je veoma jednostavan patch kod koji patchuje bajtove na adresama 00401035 i 0040108A, posle cega se skace na OEP pomocu komandi PUSH i RET. Kada snimimo ove promene i restartujemo Olly mozemo i da izmenimo skok na adresi 00407921 tako da on pokazuje na patch kod, odnosno izmenicemo ga u ovo:

```
00407920   9D                POPFD
00407921  - E9 0D8AFFFF      JMP inline_n.00400333
```

Konacno mozemo da snimimo sve promene i pokusamo da startujemo patchovan fajl, ali …. kao sto vidimo negde smo pogresili jer se program nije startovao kao sto je trebalo, cak naprotiv on se “srusio”. Da li smo nesto zaboravili?

Pogledajmo za trenutak spisak importa koji se nalaze u pakovanom .exe fajlu. Ako odemo u prozor executable modules videcemo ovo:

```
Names in crackme_
Address   Section     Type   (  Name                      Comment
004075BC  nsp1        Import (  KERNEL32.VirtualAlloc
004075C0  nsp1        Import (  KERNEL32.VirtualFree
004075B8  nsp1        Import (  KERNEL32.VirtualProtect
```

Par importa zaduzenih za alociranje, zastitu i oslobadjanje memorije. Zastitu memorije? Da, VirtualProtect se koristi da bi se nekoj memorijskoj alokaciji

dodelio status: Full Access, Read/Write, Read only,… Posto se nas inline patch rusi pri pokusaju da upise nas patch u glavnu sekciju zakljucujemo da je glavna sekcija zakljucana za pisanje od strane VirtualProtecta. Ovo bi trebalo da bude lako iz razloga sto se VirtualProtect vec nalazi u import tablici, odnosno ne ucitava se dinamicki. Logicno je da prvo sto nam pada na pamet je postavljanje break-pointa na taj import, ali to nije pravo resenje, pogotovo ne kada se radi o pakerima. Posto Olly nece zastati prilikom postavljanja break-pointa, moramo da primenimo sledeci trik.
Naime posto se import vec nalazi u fajlu ocigledno je da program nema razloga za dinamickim ucitavanjem importa, odnosno packer mora prilikom pozivanje VirtualProtecta pristupiti adresi na kojoj se on nalazi. Ako pogledamo nasu tablicu importa videcemo da se VirtualProtect nalazi na adresi 004075B8 zbog cega bi bilo najbolje da postavimo jedan break-point na toj adresi. Odnosno necemo postaviti obican break-point na toj adresi, nego cemo otici na tu adresu u Hex dumpu kako bismo postavili memorijski break-point on access. Zasto ovo radimo? Pa ako pogledate hex dump:

```
004075B8 >9E 16 E6 77 72 AC E7 77  …wr..w
004075C0 >CB 15 E8 77 FD 98 E7 77  …w…w
```

videcete da je DWORD na adresi 004075B8 u stvari pointer ka adresi 77E6169E, koja je sigurno lokacija APIja u kernel32.dll fajlu. Posle postavljanja memorijskog break-pointa na adresu 004075B8 i klika na F9:

```
004078F5   74 28            JE SHORT crackme_.0040791F
004078F7   43               INC EBX
004078F8   8DB5 6DFEFFFF   LEA ESI,DWORD PTR SS:[EBP-193]
004078FE   8B16             MOV EDX,DWORD PTR DS:[ESI]
00407900   56               PUSH ESI
00407901   51               PUSH ECX
00407902   53               PUSH EBX
00407903   52               PUSH EDX
00407904   56               PUSH ESI
00407905   FF33             PUSH DWORD PTR DS:[EBX]
00407907   FF73 04          PUSH DWORD PTR DS:[EBX+4]
0040790A   8B43 08          MOV EAX,DWORD PTR DS:[EBX+8]
0040790D   03C2             ADD EAX,EDX
0040790F   50               PUSH EAX
00407910   FF95 11FFFFFF   CALL NEAR DWORD PTR SS:[EBP-EF]  ; kernel32.VirtualProtect
00407916   5A               POP EDX
00407917   5B               POP EBX
00407918   59               POP ECX
00407919   5E               POP ESI
0040791D  ^ E2 E1           LOOPD SHORT crackme_.00407900
0040791F   61               POPAD
00407920   9D               POPFD
00407921  - E9 9A99FFFF     JMP crackme_.004012C0                <- Jmp to OEP
```

nalazimo se na adresi koja je oznacena zelenom bojom. Ako analiziramo sve parametre VirtualProtect funkcije videcemo da se ona ponavlja i primenjuje na svaku sekciju otpakovanog fajla. Takodje primecujemo da je ovo poslednja operacija koju nSPack izvrsava pre skoka na OEP, odnosno na nas patch kod. Ovde imamo par mogucih resenja, ali posto je cilj inline patch da smanji velicinu samog patcha mi cemo pronaci najjednostavnije resenje. To jest pogledacemo skok na adresi 004078F5 i videcemo da kada se on izvrsi, ne izvrsava se VirtualProtect zastita. Dakle jednostavnom promenom skoka JNZ u JMP, VirtualProtect se ne izvrsava i nas inline patch radi. Imajte na umu da je VirtualProtect cesto koriscena API funkcija i da bi trebalo da obratite paznju na nju prilikom pravljenja inline patcheva.

# Inline patching: ASPack 1.x - 2.x

Posto je pored UPXa veoma cesto koriscen packer je i ASPack. Stoga je izuzetno bitno naci nacin kako patchovati ovaj paker tako da se velicina patchovanog fajla ne poveca ni za jedan bajt. Ovo moze izgledati komplikovano ali ako se drzimo osnovnih principa patchovanja zapakovanih executable fajlova uspecemo, dakle:

- Pronaci komandu koja se poslednja izvrsava a pripada samom packeru. Ova komanda se izvrsava odmah pre skoka na otpakovan OEP stoga nju treba izmeniti tako da ona pokazuje ka nasem patchovanom kodu.
- Ako ovo prvo ne uspe treba pronaci mesto u kodu posle koga je ceo kod glavne .code sekcije otpakovan u memoriju i tu treba ubaciti pointer ka patch kodu.
- Osloboditi mesto za ubaceni kod
- Ubaciti kod za patchovanje
- U slucaju 'ostecenja' packerovog koda izvrsiti popravku
- Vratiti kontrolu ili pravom OEPu ili nastaviti sa otpakivanjem koda

Posto znamo da otpakujemo ASPack onda znamo i kako izgledaju poslednje komande pre prelaska na pravi OEP:

```
004073B0   /75 08           JNZ SHORT crackme_.004073BA
004073B2   |B8 01000000     MOV EAX,1
004073B7   |C2 0C00         RET 0C
004073BA   \68 C0124000     PUSH crackme_.004012C0
004073BF   C3               RET
```

Ovo znaci da bismo trebali da izmenimo poslednju RET komandu u neki skok ka patch kodu, ali da li je to bas tako? Restartujte metu sa CTRL+F2 i pogledajte sta se to nalazi na adresi:

```
004073B0   /75 08           JNZ SHORT crackme_.004073BA
004073B2   |B8 01000000     MOV EAX,1
004073B7   |C2 0C00         RET 0C
004073BA   \68 00000000     PUSH 0
004073BF   C3               RET
```

Kao sto vidimo kod je veoma slican ali nema onog PUSH 004012C0. Ovo znaci da se OEP prosledjuje na ovu adresu, a ovo znaci da se ovaj deo koda ne moze patchovati! Dakle potrebno je pronaci neki drugi deo koda koji se nece menjati i kojem ce se pristupiti odmah nakon otpakivanja glavne sekcije. Da biste ovo mesto nasli mozete traziti modifikaciju bajtova pomocu Memory Access opcije u Ollyju, ali posto bi ovo objasnjenje bilo presiroko preci cemo odmah na patchovanje. Dakle pronadjite ovo u kodu fajla crackme.aspacked.original.exe

```
00407180   F3:A5           REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS>
00407182   8BC8            MOV ECX,EAX
00407184   83E1 03         AND ECX,3
00407187   F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[>
```

Izmenite ga u sledeci kod:

```
00407180   F3:A5           REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS>
00407182 - E9 AC91FFFF     JMP inline_a.00400333          <- Pointer to patch
00407187   F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[>
```

Sta smo ovde uradili? Jednostavno smo zamenili komande MOV ECX,EAX i AND ECX,3 komandom koja ce izvrsiti patch naseg koda. Ali zasto bas adresa 00400333? Zato sto se ova adresa nalazi u PE headeru fajla gde ima vise

nego dovoljno mesta da mi ubacimo nas kod za patchovanje. Na toj adresi cemo postaviti sledece:

```
00400333   C605 35104000 F>      MOV BYTE PTR DS:[401035],0FF
0040033A   C605 8A104000 0>      MOV BYTE PTR DS:[40108A],0
00400341   8BC8                  MOV ECX,EAX
00400343   83E1 03               AND ECX,3
00400346  - E9 3C6E0000          JMP inline_a.00407187
```

Sta smo ovde uradili? Jednostavno smo patchovali kod pomocu MOV BYTE PTR komande (*necu objasnjavati zasto bas ove adrese*) posle cega smo morali da vratimo komande koje smo slucajno izbrisali. Dakle da bi ovaj kod radio potrebno je da vratimo izbrisane komande MOV ECX,EAX i AND ECX,3. Kada i ovo zavrsimo potrebno je da se vratimo na neizmenjeni deo koda iz ASPack sekcije. Dakle JMP 00407187 nas vraca na:

```
00407187   F3:A4               REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

I program ce nastaviti sa izvrsavanjem sve dok ne dodje do OEPa, a nas kod ce biti patchovan bas kako treba. Stoga posle izvrsavanja ovog patcha nasa meta nece prikazivati ni NAG a i Secret meni ce biti ukljucen.

## Inline patching: EZIP 1.0

Buduci da je ASPack bio malo tezi primer sada cu objasniti veoma lako ubacivanje koda u EZip. Otvorite metu crackme.eziped.original.exe pomocu Ollyja i pogledajte kako izgleda OEP:

```
004070BE > $ /E9 19320000    JMP crackme_.0040A2DC
```

Posto je ovo neka vrsta redirekcije ka kodu pakera ispraticemo ovaj JMP sa F8 posle cega cemo se naci ovde:

```
0040A2DC  /> \55             PUSH EBP
0040A2DD  |. 8BEC            MOV EBP,ESP
0040A2DF  |. 81EC 28040000   SUB ESP,428
0040A2E5  |. 53              PUSH EBX
0040A2E6  |. 56              PUSH ESI
```

Posto smo vec otpakivali EZip znamo da se posle izvrsavanja JMP EAX komande program prebacuje i sa izvrsavanja pakerovog koda prelazi na izvrsavanje otpakovanog koda. To se desava ovde:

```
0040A687  |. 5D              POP EBP
0040A688  |. FFE0            JMP NEAR EAX
```

Posto smo vec rekli da skokove koji vode do prvog OEPa mozemo izmeniti tako da umesto ka OEPu vode ka nasem patch kodu, ovde necemo uraditi nista novo. Samo cemo skok JMP EAX izmeniti u patch kod a sav kod koji ce pri tome biti izgubljen cemo nadoknaditi na racun CC bajtova iza RET komande. Kako znamo da ovo smemo da uradimo? Pa jednostavno samo jedna INT3 komanda ima smisla ali vise njih sluze samo za ispunjenje nekog prostora! Dakle posle patcha nas kod ce izgledati ovako:

```
0040A687   . 5D              POP EBP
0040A688   . 90              NOP
0040A689   . 90              NOP
0040A68A   > C605 35104000   MOV BYTE PTR DS:[401035],0FF
0040A691   . C605 8A104000   MOV BYTE PTR DS:[40108A],0
0040A698   . FFE0            JMP NEAR EAX
```

Napominjem da u fajlu inline.eziped.exe postoji dodatni kod ispod JMP EAX komande koji je isti kao i onaj koji se nalazi u originalnom fajlu. Ovo je takozvano vracanje ukradenog koda. Ovde smo to uradili na racun CC bajtova jer je to u ovom slucaju, kod ovog pakera, moguce.

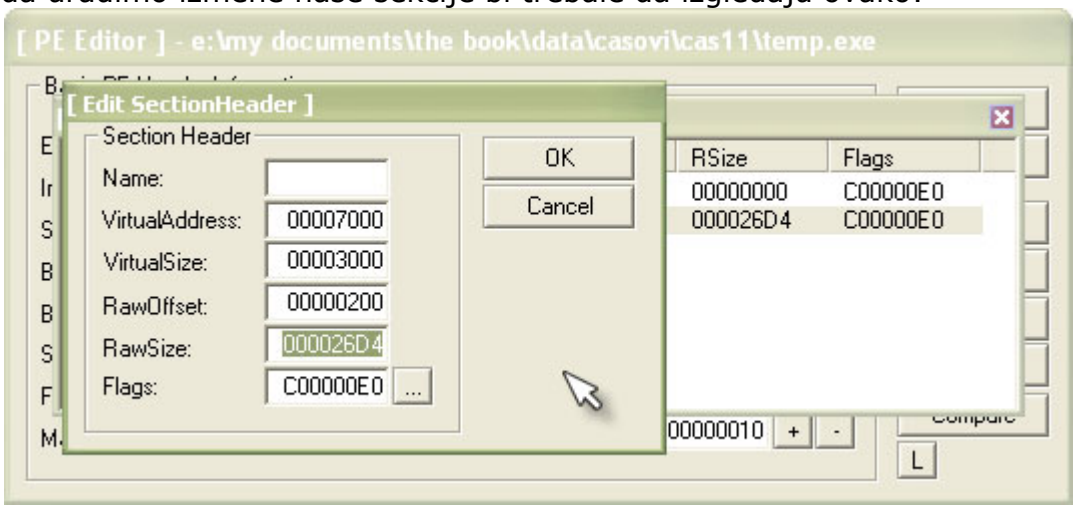
# Inline patching: FSG 1.33

Do sada smo naucili kako da ubacujemo kod u .exe fajl kada u njemu ima dovoljno mesta za ubacivanje koda. Ali sta cemo da radimo kada u .exe fajlu nema dovoljno mesta? E tada je potrebno dodati prazan prostor u .exe fajl kako bismo ubacili nas patch kod. Ovo cemo uraditi na primeru mete crackme.fsged.original.exe koju cemo prvo otvoriti pomocu Ollya kako bi smo nasli skok ka OEPu. Posto smo vec otpakivali FSG znamo da se skok ka OEPu nalazi ovde:

```
0040960A  > \FE0E            DEC BYTE PTR DS:[ESI]
0040960C  .- 0F84 AE7CFFFF   JE crackme_.004012C0
00409612  . 56               PUSH ESI
```

Sada cemo zatvoriti Olly i otvoricemo nasu metu pomocu nekog hex editora i dodacemo na sam kraj fajla 0x71 ili 113 (*decimalno*) 0x00 bajtova. Ovo se recimo u HexWorkShopu radi tako sto odemo na sam kraj fajla, pritisnemo Inset dugme da bismo ubacili 0x00 bajtove i drzimo nulu sve dok ne ubacimo svih 0x71 bajtova.

Kada smo ovo uradili mozemo da zatvorimo HexWorkShop i da otvorimo snimljeni fajl u LordPEu kako bismo prosirili poslednju sekciju. Ovo cemo uraditi klikom na dugme sections u LordPEu dok smo u PE Editor modu. Sada treba izabrati poslednju sekciju i promeniti jos Raw size (*stvarnu fizicku velicinu*) sa 00002664 na 000026D5 - 1 jer smo na kraj fajla dodali 0x71 bajtova, a zelimo da budemo sigurni da ce fajl raditi posle popravke sectiona pomocu LordPEa. Zato umesto vrednosti 000026D5 unosimo 000026D4. Kada uradimo izmene nase sekcije bi trebale da izgledaju ovako:




Ostaje nam jos samo da ubacimo nas kod u novonastali prazan prostor pomocu Ollyja. Sam kod cemo poceti da ubacujemo ovde:

```
00409667    00          DB 00
```

A sam kod koji cemo ubaciti kao i kod ASPacka i EZipa izgleda ovako:

```
00409667    C605 35104000>MOV BYTE PTR DS:[401035],0FF
0040966E    C605 8A104000>MOV BYTE PTR DS:[40108A],0
```

Naravno kada smo ovo uradili potrebno je da skok JE 004012C0 promenimo tako da on pokazuje na nas kod (*u JE 00409667*), to jest na 00409667, i da dodamo jedan skok odmah posle naseg patch koda koji ce voditi na OEP, to jest potrebno je da dodamo JMP 004012C0 komandu na adresu 00409675.

# Inline patching: PeX 0.99

Posto su prethodni primeri bili "previse" laki vreme je da konacno uradimo nesto ozbiljno i da inlineujemo jedan jako tezak paker. Sta je to tako tesko kod ovog pakera? Tesko je to sto se JMP koji vodi na OEP nalazi u delu koji se modifikuje iz memorije koja se dinamicki alocira. Ovo znaci da se deo koda koji vodi do OEPa generise uvek na istom mestu ali posto se memorija sa koje se on generise uvek drugacije alocira, to jest uvek ima drugu adresu, stoga patchovanje ovakve aplikacije predstavlja problem. Na svu srecu po nas ovaj problem nije neresiv.

Prvi u nizu problema sa kojim moramo da se suocimo je problem odredjivanja mesta na koje cemo da ubacimo redirekciju ka nasem patch kodu. Ovo mesto cemo pronaci tako sto cemo postaviti break-point on write na glavnu .code sekciju. Posle pritiska na F9 zavrsicemo ovde:

```
0012FF91  - FFE2              JMP NEAR EDX                ; crackme_.00407154
```

Posto se ovde ne vrsi pisanje u glavnu sekciju pritisnucemo SHIFT+F9, kako bismo presli preko ovog exceptiona, posle cega cemo zavrsiti ovde:

```
004071EE   ?  0F0B            UD2
```

Posto je i ovo jedna od nevaznih ASM komandi, ponovo cemo pritisnuti SHIFT+F9 posle cege cemo se naci bas tamo gde treba:

```
02AB006B   F3:A4              REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
02AB006D   5F                 POP EDI
02AB006E   83C7 28            ADD EDI,28
```

Dakle to je jedna od onih adresa koje se dinamicki alociraju pomocu VirtualAlloc APIja. Na zalost postavljanje bilo kakvog break-pointa na ovaj API nam nece pomoci iz razloga sto se memorija alocira malo drugacije. Ostaje nam samo da pomocu traceinga kroz kod pronadjemo komande STOS, MOVS ili REP koje se koriste za pisanje u memoriju. Naravno uporedjivacemo adrese na koje se pise sa 02AB0068 (*ovo je vrednost za moj kompjuter i za ovaj start programa*) i naci cemo bas ono sto smo trazili na adresi:

```
004072CD   .  8BF7            MOV ESI,EDI
004072CF   .  2BF0            SUB ESI,EAX
004072D1   .  F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
004072D3   .  5E              POP ESI
```

Ovde cemo uraditi redirekciju ka patch kodu. Za uspesno preusmeravanje potreban nam je jos sadrzaj registara kada se izvrsava REP komanda tako da cemo ih zapisati negde kako bismo ih kasnije koristili.

```
EAX 00000001
ECX 00000002
EDX 00407308 crackme_.00407308
EBX 00006473
ESP 0012FF58
EBP 00000001
ESI 009B0003
EDI 009B0004
EIP 004072D1 crackme_.004072D1
```

Kada smo ovo zapisali naisli smo na jos jedan problem. Naime problem je sto se adrese koje su vec zapisane u memoriji pomocu REP komande ponovo koriste pomocu MOVS komande na adresi:

```
00407268   > /A4              MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00407269   > |E8 68000000      CALL crackme_.004072D6
```

Iz tog razloga moramo da uradimo proveru da li je adresa prosla MOVS komandu. Ovo je malo komplikovaniji deo ali je takodje izvodljivo. Komplikovanije je u smislu da treba da odredimo koja je to bezbedna vrednost na kojoj nece doci do ponovnog koriscenja vec zapisanih bajtova. Recimo da je ta vrednost 0x400. Ovo znaci da cemo moci da ubacimo nas patch u memoriju na trenutno otpakovanu lokaciju tek kada program otpakuje sledecih 0x400 bajtova. Zbog ovoga ce nas patch biti program u programu u kojem cemo prvo morati da uradimo redirekciju ka nasem patch kodu koji cemo smestiti na adresu 00400333:

```
004072C9   95              XCHG EAX,EBP
004072CA   8BC5            MOV EAX,EBP
004072CC - E9 6290FFFF     JMP patch.00400333
004072D1   F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

Originalni kod koji se nalazi na ovoj adresi je:

```
004072C9   95              XCHG EAX,EBP
004072CA   8BC5            MOV EAX,EBP
004072CC   56              PUSH ESI
004072CD   8BF7            MOV ESI,EDI
004072CF   2BF0            SUB ESI,EAX
004072D1   F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

Dakle zamenili smo komande PUSH ESI; MOV ESI,EDI; SUB ESI,EAX skokom ka nasem buducem patchu koji ce se nalaziti na adresi 00400333. Ovaj patch ce biti pravi mali program i tu cemo morati da uradimo sledece:

- vratimo ukradeni kod
- proverimo da li je trenutno otpakovana adresa veca za 0x400 od prve otpakovane adrese na dinamicki alociranoj sekciji memorije
- da ubacimo redirekciju OEPa ka nasem patch kodu
- da patchujemo nasu metu

Ali pre nego sto uradimo bilo sta od ovoga potrebno je da saznamo kako to PeX skace na OEP. To jest moramo da znamo gde treba da ubacimo redirekciju PeXovog koda ka nasem patch kodu. Ovo cemo uraditi tako sto cemo prvo restartovati nasu metu u Ollyju i onda cemo postaviti memorijski break-point on write na glavnu .code sekciju. Nakon prelaska preko exceptiona doci cemo do dela koda koji smo vec upoznali i koji se nalazi na virtualno alociranom mestu. Sada treba ukloniti prethodno postavljen break-point i postaviti novi, isti, ali ovaj put na poslednju sekciju koja sadrzi SFX kod i importe. Posle ponovnog pritiska na F9 doci cemo do REP komande koja vrsi pisanje u PeX unpacker sekciju. Ponovo cemo ukloniti nas memorijski break-point i posle 2x klika na F8 naci cemo se na adresi 00407001. Ovo je prva adresa u PeXovoj sekciji i ona se razlikuje od pocetne, od one koju mozete videti kada prvobitno otvorite PeXovan fajl sa Ollyjem i ne izvrsite ni jednu komandu, jer je PeX ovde izvrsio pisanje novog sadrzaja prilikom izvrsavanja poslednje REP komande. Iz ovog razloga cemo traceovati kroz ovaj kod sa F7 sve dok ne dodjemo do:

```
00407032   68 BF124000      PUSH crackme_.004012BF
00407037   EB 01            JMP SHORT crackme_.0040703A
00407039   C7               ???                                ; Unknown command
0040703A   58               POP EAX
0040703B   40               INC EAX                            ; crackme_.004012BF
0040703C   50               PUSH EAX        <- EAX = 004012BF + 1
0040703D   C3               RET             <- Jmp to OEP
```

Posto se komanda PUSH 004012BF ocigledno odnosi na PUSH OEP - 1, stoga je potrebno da pronadjemo sa koje se to adrese menja sadrzaj adrese

00407032 + 1. Ponovo cemo restartovati Olly i ponovo cemo podesiti memorijske break-pointe na vec objasnjen nacin. Kada dodjemo do zadnje REP komande primeticemo da se adresa 00407033 prepisuje sa sadrzajem:

```
ECX=0000000C (decimal 12.)
DS:[ESI]=[009B0396]=68 ('h')
ES:[EDI]=[00407032]=C4
```

privremene adrese 009B0396 (*ovo ce biti drugacije svaki put i za svaki kompjuter*). Problem sa ovim nasim otkricem je da znamo sa koje se adrese upisuje PUSH OEP - 1 komanda ali ne znamo kako ce ta adresa izgledati jer se adresa sa koje se upisuje PUSH menja svaki put. Ali posle vise startovanja nase mete u pokusaju da pronadjemo zavisnost izmedju 009B0396 i 00407033 dolazimo do ovoga:

```
ECX=0000000C (decimal 12.)
DS:[ESI]=[00A10396]=68 ('h')
ES:[EDI]=[0045D032]=C4
```

Kao sto se vidi PUSH OEP - 1 se uvek upisuje sa VirualAlloca + 396 adrese. Ovo nam je pomoglo i sada mozemo da ubacimo nas patch kod na adresu 00400333. Prvo cemo ubaciti kod koji smo ukrali na racun skoka ka 00400333 adresi:

```
00400333   56              PUSH ESI
00400334   8BF7            MOV ESI,EDI
00400336   2BF0            SUB ESI,EAX
```

a posle ovoga moramo da uradimo kod koji ce racunati VirtualAlloc + 396 + 0x400 (*jer je ovo odredjena velicina kao safe za pisanje*). Posto su EDI i ESI registri isti iskoristicemo jedan od njih da ga privremeno zamenimo EAXom i da na EAXu izracunamo novu adresu koja je potrebna da postoji u memoriji da bismo mi mogli da izvrsimo pisanje redirekcije OEPa. Ovo izgleda ovako:

```
00400338   97              XCHG EAX,EDI
00400339   66:33C0          XOR AX,AX
0040033C   05 96080000      ADD EAX,896
00400341   3BF0             CMP ESI,EAX                  // Da li EAX adresa postoji
00400343   7F 0B            JG SHORT patch.00400350
00400345   97              XCHG EAX,EDI                 // Ako ne postoji
00400346   8BFE            MOV EDI,ESI                  // samo se vrati iz patcha
00400348   03F8            ADD EDI,EAX                  // vracajuci registre u
0040034A   68 D1724000      PUSH patch.004072D1          // prethodno stanje
0040034F   C3               RET
00400350   C780 01FBFFFF 5> MOV DWORD PTR DS:[EAX-4FF],0040035B // Patch lokacija
0040035A  ^ EB E9           JMP SHORT patch.00400345     // Vrati se iz patcha
```

I naravno posle ubacivanja adrese - 1 na kojoj se nalazi nas patch kod potrebno je vratiti sve registre u prethodno stanje kako PeX ne bi pravio nikakve greske. Posto smo vec definisali da se novi patch kod nalazi na adresi 0040035B + 1 potrebno je samo da ga ubacimo:

```
0040035C   C605 35104000 F>MOV BYTE PTR DS:[401035],0FF // Patch kod
00400363   C605 8A104000 0>MOV BYTE PTR DS:[40108A],0   // Patch kod
0040036A   68 C0124000      PUSH patch.004012C0          // OEP lokacija
0040036F   C3               RET                          // JMP to OEP
```

I to je to. Uspesno smo patchovali PeXovan program bez obzira na to sto se u PeXu OEP "prepisuje" sa dinamicki alocirane adrese. Ovo jeste bilo malo teze uraditi ali smo dosta toga naucili o inlineingu zahvaljujuci bartu iz xt-a koji je autor PeXa i FSGa.

# Making a loader

Kao sto je vec objasnjeno loaderi se koriste kada zelimo da napravimo patch za pakovane ili kriptovane, ali mi cemo napraviti loader za obican crackme.


Kao sto vidimo primer koji se nalazi u folderu Cas11 a zove se LOADME.exe prikazuje poruku da nije crackovan. Ovo znaci da se negde u programu nalazi poruka o tacno crackovanom programu. Ta poruka se nalazi ovde:

```
00401052   .  47 6F 6F 64 2>          ASCII "Good Cracker !!!"
00401062   .  00                      ASCII 0
```

A da se ona prikazuje ovde:

```
004010BF   .  BB 01000000             MOV EBX,1
004010C4   .  83FB 01                 CMP EBX,1
004010C7   .  75 11                   JNZ SHORT LOADME.004010DA
004010C9   .  68 40104000             PUSH LOADME.00401040   ; /Text = "Bad Cracker !!!"
004010CE   .  6A 64                   PUSH 64
004010D0   .  FF75 08                 PUSH DWORD PTR SS:[EBP+8]
004010D3   .  E8 96000000             CALL <JMP.&user32.SetDlgItemTextA>
004010D8   .  EB 15                   JMP SHORT LOADME.004010EF
004010DA   >  837D 10 02              CMP DWORD PTR SS:[EBP+10],2
004010DE   .  75 0F                   JNZ SHORT LOADME.004010EF
004010E0   .  68 52104000             PUSH LOADME.00401052   ; /Text = "Good Cracker !!!"
004010E5   .  6A 64                   PUSH 64
004010E7   .  FF75 08                 PUSH DWORD PTR SS:[EBP+8]
004010EA   .  E8 7F000000             CALL <JMP.&user32.SetDlgItemTextA>
```

Patchovacemo skok na adresi 004010C7 JNZ u JMP, to jest sa 75 11 u EB 11 i drugi skok na adresi 004010DE sa 75 0F u 90 90. Ovde necemo uraditi fizicki nego memorijski patch. Za ovo cemo iskoristiti R!SC Process Patcher koji se nalazi zipovan u folderu Cas11 kao fajl rpp.zip. Da biste napravili loader morate prvo napraviti jedan .rpp fajl. Njegov sadrzaj ce izgledati ovako:

```
F=LOADME.exe:                 ; PROCESS TO PATCH
O=loader.exe:                 ; LOADER TO CREATE
P=4010C7/75,11/EB,11:         ; JNZ 2 JMP
P=4010DE/75,0F/90,90:         ; JNZ 2 NOP
$
```

Posle ovoga mozete startovati Rpp.exe i pomocu njega kompajlovati ovaj .rpp skript. Kao rezultat dobicete fajl loader.exe koji ce uspesno patchovati memoriju LOADME.exe fajla. Sama struktura .rpp fajla je jednostavna pa je necu objasnjavati, a ako vam na prvi pogled nije jasno uporedite adrese i bajtove u redovima koji pocinju sa P=.

# 12 Nightmare

Ovo poglavlje je koncepcijski zamisljeno malo drugacije od ostalih. U prethodnim poglavljima sam vam detaljno objasnjavao probleme vezane za razlicite RCE probleme. Sada cu vam ja zadavati probleme koje cete morati sami da resite u skladu sa vasim mogucnostima. Naravno i ovde ce biti objasnjena resenja svih problema samo sto se nadam da cete ih slabije citati jer ste do sada trebali sami da shvatite sustinu RCEa i da se osamostalite tako da mozete sami da resavate probleme. Shvatite ovo poglavlje kao jedan konacan ispit.

# BruteForceing the Secret


**Zadatak:**

Relativno lak zadatak ...\Cas12\Secret.exe treba reversovati tako da za uneti serijski broj program prikaze poruku da je tacno crackovan, ovaj prvi deo uraditi bez bruteforceinga. Pored ovoga treba otkriti nacin na koji se racuna serijski broj i napraviti keygenerator (*bruteforcer*).

**Resenje: - Deo I**

Nas je zadatak da rucno izracunamo serijski broj u prvom delu rucno dok cemo za drugi deo napisati bruteforcer. Ucitajte metu u Olly i u nju unesite lazne podatke, unesite 123456 kao serijski broj. Posto je ovo ocigledno netacan serijski broj program nece izbaciti nikakvu poruku. Stoga cemo mi sami potraziti moguce tacne poruke u fajlu pomocu String referenci, i naci cemo sledeci string:

```
Text strings referenced in Secret:CODE, item 2365
 Address=0046A19C
 Disassembly=MOV EAX,Secret.0046A1D8
 Text string=ASCII "Ok!, Now write a Keygen/Bruteforcer"
```

Posto nam je ovo izuzetno zanimljivo kliknucemo dva puta na ovaj string kako bismo videli gde se on poziva. To nas stavlja ovde:

```
0046A194  |.  81FF 84030000   CMP EDI,384
0046A19A  |.  75 0A           JNZ SHORT Secret.0046A1A6
0046A19C  |.  B8 D8A14600     MOV EAX,Secret.0046A1D8 :ASCII      <- Ovde smo
```

Kao sto vidimo direktno iznad ove poruke se nalazi jedno poredjenje i jedan uslovni JNZ skok. Postavicemo jedan break-point na sam pocetak ovog CALLa, na adresu 0046A108, a posle ovoga cemo ponovo pritisnuti dugme Check u crackemeu. Polako cemo proci ovaj serijski CALL sa F8 sve dok ne dodjemo dovde:

```
0046A141  |.  83F8 0C         CMP EAX,0C
0046A144  |.  75 60           JNZ SHORT Secret.0046A1A6
```

EAX ce sadrzati duzinu naseg serijskog broja, a ako duzina nije jednaka 0x0C to jest 12 decimalno onda ce se preskociti skoro ceo CALL. Ovo znaci da nas serijski broj mora da bude dugacak tacno 12 karaktera, a zbog ovoga cemo umesto 123456 uneti 111111111111 kao serijski broj. I sada kada ponovo dodjemo do istog dela koda videcemo da se JNZ skok nece izvrsiti, stoga idemo dalje kroz kod sa F8. Stigli smo do jednog skoka koji nas je takodje

```
0046A14B  |> /8D55 F8         /LEA EDX,DWORD PTR SS:[EBP-8]
0046A14E  |. |8B86 FC020000   |MOV EAX,DWORD PTR DS:[ESI+2FC]
0046A154  |. |E8 47B8FCFF     |CALL Secret.004359A0
0046A159  |. |8B45 F8         |MOV EAX,DWORD PTR SS:[EBP-8]
0046A15C  |. |0FB64418 FF     |MOVZX EAX,BYTE PTR DS:[EAX+EBX-1]
0046A161  |. |83F8 41         |CMP EAX,41
0046A164  |. |7C 40           |JL SHORT Secret.0046A1A6          <- Ovaj skok
0046A166  |. |83F8 5A         |CMP EAX,5A
0046A169  |. |7F 3B           |JG SHORT Secret.0046A1A6          <- Drugi skok
0046A16B  |. |43              |INC EBX
0046A16C  |. |83FB 0D         |CMP EBX,0D
0046A16F  |.^\75 DA           \JNZ SHORT Secret.0046A14B
```

izbacio iz CALLa, a taj skok je obelezen na delu koda. Kao sto vidimo postoji i drugi skok koji se koristi takodje za izbacivanje iz koda. Sta se ovde poredi? Ovde se poredi svako slovo, to jest njegova ASCII vrednost, sa hex brojevima 0x41 i 0x5A. Ovo znaci da se uneti serijski broj mora sastojati od slova koja se nalaze u opsegu A-Z i moraju biti iskljucivo velika. Stoga cemo

opet modifikovati nas uneti serijski broj u AAAAAAAAAAAA. Kao sto vidimo nakon ponovnog pokretanja mete, ona ce sada stici sa izvrsavanjem do:

```
0046A176  |> /8D55 F4         /LEA EDX,DWORD PTR SS:[EBP-C]
0046A179  |. |8B86 FC020000   |MOV EAX,DWORD PTR DS:[ESI+2FC]
0046A17F  |. |E8 1CB8FCFF    |CALL Secret.004359A0
0046A184  |. |8B45 F4        |MOV EAX,DWORD PTR SS:[EBP-C]
0046A187  |. |0FB64418 FF    |MOVZX EAX,BYTE PTR DS:[EAX+EBX-1]
0046A18C  |. |03F8           |ADD EDI,EAX
0046A18E  |. |43             |INC EBX
0046A18F  |. |83FB 0D        |CMP EBX,0D
0046A192  |.^\75 E2          \JNZ SHORT Secret.0046A176
```

Posto je ovo poslednji loop pre poredjenja registra EDI sa vrednoscu 0x384, zakljucujemo da se ovde racuna serijski broj. A kao sto se iz loopa vidi jednostavno se na EDI registar dodaju vrednosti svih slova iz unetog serijskog broja. Ovo znaci zbir ASCII vrednosti slova unetog serijskog broja mora biti jednak 900 decimalno. I sada smo konacno dosli do dela u kojem cemo izracunati serijski broj rucno i to po formuli:

**900 / 12 = 75 iz cega sledi da je validan serijski KKKKKKKKKKKK.**

Zasto? Zato sto se sabira 12 slova, a ako su sva ista deljenjem cemo dobiti vrednost tog slova. Posto je dobijena vrednost 75, onda je trazeno slovo ASCII vrednost broja 75 to jest slovo K. Slobodno probajte serijski broj u nasoj meti i videcete da smo bili u pravu.

**Resenje: - Deo II**

Ostaje nam samo da na osnovu onoga sto znamo napisemo program koji ce generisati validan serijski broj. Ovaj algoritam mozemo jako lako napisati. Zamislite da su sva slova K. Ako svaka dva slova sada pomerimo za po jedan karakter: LJKKKKKKKKKK idalje cemo dobijati tacan serijski broj. Na osnovu ove cinjenice cemo napisati sledeci C++ algoritam:

```
 char serial[12]="";
 char buffer[12]="";
 char bufferd[12]="";
 int i,inc;
for(i=0; i<12; i+=2){
   inc = -10 + (rand() % 21);
   serial[i] = 75 + inc;
   serial[i+1] = 75 - inc;
}
for (i=0;i<12;i++){
        wsprintf(buffer,"%c",serial[i]);
        strcat(bufferd,buffer);
}
```

Kao sto primecujete imamo dve petlje ovde. Prva sluzi za racunanje serijskog broja druga se koristi za konacno formatiranje bufferd stringa koji ce sadrzati tacan serijski broj na kraju ovog loopa (*ovo ovako mora iz nekog razloga*). Mozete probati i bruteforcer koji sam napisao za ovu metu, on se nalazi u istom folderu kao i meta a zove se bruteforce.exe

# Keygening Scarabee #4


**Zadatak:**

Relativno lak zadatak ...\Cas12\crackme#4.exe treba reversovati tako da se ukljuci Check dugme i da posle pritiska na ovo dugme program prikazuje poruku da je program registrovan. Dozvoljeno je patchovati samo bajtove koji sluze za iskljucivanje / ukljucivanje dugmeta. Pored ovoga treba otkriti nacin na koji se racuna serijski broj i napraviti keygenerator.

**Resenje:**

Da bismo ukljucili dugme Check iskoristicemo program DeDe. Ako nemate ovaj program samo patchujete program na objasnjeni nacin.

Ucitajte metu u DeDe i u procedurama izaberiti FormCreate i videcete sledece:

```
0044FBA8   33D2                    xor    edx, edx
* Reference to control TForm1.Button1 : TButton
0044FBAA   8B80F0020000            mov    eax, [eax+$02F0]
0044FBB0   8B08                    mov    ecx, [eax]
* Reference to method TButton.SetEnabled(Boolean)
0044FBB2   FF5164                  call   dword ptr [ecx+$64]
0044FBB5   C3                      ret
```

Kao sto se vidi iz prilozenog kada se kreira forma dugme se iskljuci pozivom na komandu CALL DWORD ptr[ecx+$64], zbog cega cemo NOPovati ovaj CALL. Posle ovoga mozemo otvoriti Olly i nastaviti sa reversingom ove mete. Potrazicemo sve moguce stringove koji bi nam ukazali na to gde se to racuna serijski broj. Videcemo ovu zanimljivu referencu:

```
Text strings referenced in crackme#:CODE, item 1850
 Address=0044FB5D
 Disassembly=MOV EDX,crackme#.0044FB98
 Text string=ASCII "  Registered!"
```

koja ce nas odvesti ovde:

```
0044FB5D  |.  BA 98FB4400   MOV EDX,crackme#.0044FB98          ;  ASCII "  Registered!"
0044FB62  |.  E8 4DF1FDFF   CALL crackme#.0042ECB4
```

Ovo znaci da se CALL koji se koristi za racunanje serijskog broja nalazi izmedju adresa 0044FA64 i 0044FB8C.

Postavicemo break-point na adresu 0044FA64 kako bismo analizirali deo koda koji sluzi za proveru serijskog broj i pritisnucemo dugme Check u samom programu posle cega cemo zavrsiti na nasem break-pointu. Proci cemo kroz kod 1x pritiskajuci F8 da bismo saznali sto vise o kodu za proveru serijskog broja. Primeticemo sledece:

```
0044FA8D  |.  8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8] ; U EAXu je putanja do fajla
0044FAA3  |.  83F8 15          CMP EAX,15                   ; U EAXu je duzina putanje
0044FAA6  |.  0F85 BB000000    JNZ crackme#.0044FB67        ; EAX se poredi sa 15h
```

Kada se ovaj skok izvrsi zavrsicemo odmah ispod poruke o tacnom serijskom broju zbog cega se ovaj skok nikada ne sme izvrsiti. Zbog ovoga cemo ovaj crackme iskopirati u folder c:\aaaaaaaaaaaaaaaaa i reversovacemo ga tu. Ovo radimo jer duzina putanje do exe fajla mora da bude 15h ili 21 decimalno a posto se racuna i c:\ u duzinu putanje folder mora biti bas ovaj ili neki drugi sa istom duzinom. Sada cemo pomocu Ollyja otvoriti fajl koji se nalazi na ovoj novoj lokaciji. Primeticete da se sada ne izvrsava skok na adresi 0044FAA6 i da cemo malo ispod toga naici na sledece provere:

```
0044FAAC  |.  8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]
0044FAAF  |.  8078 09 5C       CMP BYTE PTR DS:[EAX+9],5C
```

```
0044FAB3  |.  0F85 AE000000   JNZ crackme#.0044FB67
0044FAB9  |.  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0044FABC  |.  8078 0F 5C      CMP BYTE PTR DS:[EAX+F],5C
0044FAC0  |.  0F85 A1000000   JNZ crackme#.0044FB67
```

Sledece dve provere na adresama 0044FABC i 0044FAAF su nam jako bitne. Primetite da se odredjena slova iz imena porede sa vrednoscu 5C odnosno sa karakterom '\'. Ovi bajtovi se nalaze na adresama 008A3701 i 008A3707. Ovo ujedno znaci da 9. i 15. slovo iz putanje do fajla moraju biti '\' a ovo znaci da se fajl nalazi u dodatnim poddirektorijumima zbog cega cemo ovaj exe prebaciti u sledeci folder C:\aaaaaa\bbbba\ccccc i primeticemo da se sada nijedan JNZ skoka ne izvrsava i da cemo nastaviti sa analizom koda od adrese 0044FAC6. Dok polako izvrsavate kod videcete u donjem desnom delu CPU prozora Ollyja sledece:

```
0012F3EC  008A2078   ASCII "aaaaaabbbba"
0012F3F0  008A2064   ASCII "bbbba"
0012F3F4  008A2050   ASCII "aaaaaa"
0012F3F8  008A36F8   ASCII "C:\aaaaaa\bbbba\ccccc"
```

Ono sto vidimo je da se putanja rastavlja na poddirektorijume i da se imena prvog i drugog poddirektorijuma slepljuju u jedan string. Doci cemo do sledeceg loopa koji se koristi za izracunavanje serijskog broja.

```
0044FB13  |> /8B45 EC        /MOV EAX,DWORD PTR SS:[EBP-14]
0044FB16  |. |0FB64408 FF    |MOVZX EAX,BYTE PTR DS:[EAX+ECX-1]
0044FB1B  |. |0FAF45 DC      |IMUL EAX,DWORD PTR SS:[EBP-24]
0044FB1F  |. |0145 E0        |ADD DWORD PTR SS:[EBP-20],EAX
0044FB22  |. |41             |INC ECX
0044FB23  |. |4A             |DEC EDX
0044FB24  |.^\75 ED          \JNZ SHORT crackme#.0044FB13
```

Posle izvrsenja ovog loopa i par komandi ispod njega u donjem desnom delu CPU prozora cemo videti:

```
0012F3E8  008A2090   ASCII "71757"
```

tacan serijski broj za ovu putanju. Ispod ovoga sledi poredjenje dve vrednosti

```
0044FB47  |.  8B45 E4         MOV EAX,DWORD PTR SS:[EBP-1C]
0044FB4A  |.  8B55 E8         MOV EDX,DWORD PTR SS:[EBP-18]
0044FB4D  |.  E8 2A4BFBFF     CALL crackme#.0040467C
```

preostalog dela putanje "ccccc" i "71757". Ovo znaci da krajnji folder koji smo mi nazvali "ccccc" treba da se zove "71757". Zatvoricemo Olly i reimenovacemo ovaj direktorijum u "71757" posle cega cemo videti sledece:


Ovo znaci da smo uspeli, uspesno smo pronasli resenje ovog crackmea. Interesantan je, zar ne? Sada nam samo preostaje da napravimo keygenerator za ovaj crackme.

**Keygen:**

Vec smo pronasli gde se to racuna serijski broj a sada samo treba da razumemo taj isti algoritam i da ga prepisemo u nekom drugom programskom jeziku. Za ovaj primer ja cu koristiti samo Delphi. Pre nego sto pocnemo da se bavimo pravljenjem keygeneratora moracemo prvo da se

pozabavimo samim algoritmom da bismo saznali sto je vise moguce o samom algoritmu. Evo kako izgleda deo koda koji sluzi za generisanje tacnog serijskog broja:

```
0044FB13  |> /8B45 EC         /MOV EAX,DWORD PTR SS:[EBP-14]
0044FB16  |. |0FB64408 FF     |MOVZX EAX,BYTE PTR DS:[EAX+ECX-1]
0044FB1B  |. |0FAF45 DC       |IMUL EAX,DWORD PTR SS:[EBP-24]
0044FB1F  |. |0145 E0         |ADD DWORD PTR SS:[EBP-20],EAX
0044FB22  |. |41              |INC ECX
0044FB23  |. |4A              |DEC EDX
0044FB24  |.^\75 ED           \JNZ SHORT crackme#.0044FB13
```

Objasnicemo sta se ovde desava red po red.

- Prvo se na adresi 0044FB13 u EAX smestaju imena prva dva direktorijuma dodata jedno na drugo.
- Na adresi 0044FB16 se u EAX smesta jedno po jedno slovo stringa dobijenog sabiranjem imena foldera. EAX sadrzi hex vrednost ASCIIja tog slova a ne sam ASCII !!!
- Potom se EAX mnozi sa sadrzajem adrese EBP-24. Da bismo saznali sa cime se to mnozi EAX selektovacemo sledeci red sa ekrana:

  ```
  Stack SS:[0012F3DC]=00000043
  EAX=0000004C
  ```

  i pritisnucemo desno dugme -> Follow in dump... sto ce nas odvesti ovde:

  ```
  0012F3DC  43 00 00 00 00 00 00 00  C.......
  ```

  kao sto vidimo EAX se mnozi sa ASCIIjem slova uredjaja na kojem se nalazi crackme (e.g. C:\, D:\,...)
- Na adresi 0044FB1F se na sadrzaj adrese EBP-20 dodaje vrednost EAXa. Posto je u prvom prolazu EBP-20 jednak nuli zakljucujemo da ce se ovde naci rezultat svih sabiranja EAXa + EBP-20.
- Ostali redovi nam samo govore koliko ce se puta izvrsiti ovaj loop. Taj broj je jednak duzini stringa koji se dobije dodavanjem imena dva foldera jednog na drugo.

Mislim da je sada svima jasno kako treba napraviti keygen. Prvo treba uneti dva bilo koja imena foldera tako da je duzina prvog imena foldera sest a drugog pet. Posle ovoga koristeci formulu iz loopa treba odrediti ime treceg podfoldera koji ce se sastojati samo od brojeva. Ovaj algoritam ce izgledati bas ovako:

```
var
 drv,nrd:string;
 i,tmp,eax:integer;
begin
  Drv := ComboBox1.Text;
  nrd := '';
 for i := 1 to 11 do begin
  nrd := nrd + Chr(65 + Random(25));
 end;
  tmp := 0;
  eax := 1;
 for i := 1 to length(nrd) do begin
  eax := Ord(nrd[i]);
  eax := eax * ord(Drv[1]);
  tmp := tmp + eax;
 end;
  nrd := Drv + nrd[1] + nrd[2] + nrd[3] + nrd[4] + nrd[5] + nrd[6] + '\' + nrd[7] + nrd[8] +
nrd[9] + nrd[10] + nrd[11] + '\' + IntToStr(tmp);
  Edit1.Text := nrd;
```

Ceo source i kompajlovani .exe fajl se nalaze u folderu ..\Cas12\KeyGen\

# Patching aC 0.1


Vec sam vam objasnio kako da otpakujete aC 0.1 u poglavlju koje se bavilo otpakivanjem. Sada pred vas stavljam sledeci zadatak:

**Zadatak:**

Fizicki (*bez loadera*) patchovati NAG u fajlu ...\Cas12\unpackme#1.aC.exe bez otpakivanja na disk i bez modifikacije skoka posle provere CRCa sekcije.

**Resenje:**

Otpakujmo prvo crackme u memoriju. Posto je ovo vec objasnjeno reci cu samo da se trenutno nalazimo na OEPu 0040121E. Ako pogledamo sada adresu:

```
0040122C  . 68 F5104000          PUSH unpackme.004010F5
```

videcemo da se glavna procedura za obradu WM poruka nalazi bas na adresi 004010F5. Sada cemo pogledati sta se tamo nalazi:

```
...
004011B4  . 6A 40                PUSH 40
004011B6  . 68 41114000          PUSH unpackme.00401141
004011BB  . 68 23114000          PUSH unpackme.00401123
004011C0  . FF75 08              PUSH DWORD PTR SS:[EBP+8]
004011C3  . E8 9A000000          CALL <JMP.&user32.MessageBoxA>
004011C8  . 33C0                 XOR EAX,EAX
```

Kao sto vidimo ovde se nalazi nas NAG. Posto program ne smemo otpakivati restartovacemo ga sa CTRL + F2 i otici cemo na adresu 004011B4 da bismo pogledali sta se to nalazi na toj adresi kada je program kriptovan. Vidimo:

```
004011B4    3F                   DB 3F                ; CHAR '?'
004011B5    15                   DB 15
004011B6    3D                   DB 3D                ; CHAR '='
004011B7    14                   DB 14
004011B8    44                   DB 44                ; CHAR 'D'
004011B9    15                   DB 15
004011BA    55                   DB 55                ; CHAR 'U'
004011BB    3D                   DB 3D                ; CHAR '='
004011BC    76                   DB 76                ; CHAR 'v'
004011BD    44                   DB 44                ; CHAR 'D'
004011BE    15                   DB 15
004011BF    55                   DB 55                ; CHAR 'U'
004011C0    AA                   DB AA
004011C1    20                   DB 20                ; CHAR ' '
004011C2    5D                   DB 5D                ; CHAR ']'
004011C3    BD                   DB BD
004011C4    CF                   DB CF
004011C5    55                   DB 55                ; CHAR 'U'
004011C6    55                   DB 55                ; CHAR 'U'
004011C7    55                   DB 55                ; CHAR 'U'
```

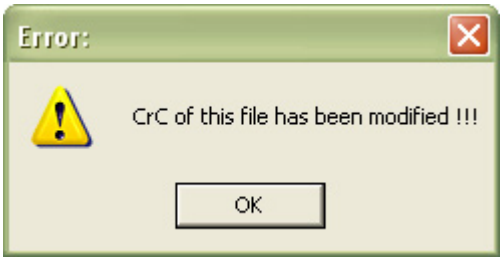
veoma kriptovan kod. Posto sve ovo moramo da promenimo u NOP moramo da znamo kako se enkriptuje 90 da bismo znali u sta da promenimo ove bajtove. Da bismo ovo saznali moramo da pogledamo kod kriptera i da vidimo kako se kriptuju adrese od 004010F5. Za to su zaduzeni sledeci loopovi:

```
004010A3  |> 8033 44             /XOR BYTE PTR DS:[EBX],44
004010A6  |. 83E9 01             |SUB ECX,1
004010A9  |. 43                  |INC EBX
004010AA  |. 83F9 00             |CMP ECX,0
004010AD  |.^ 75 F4              \JNZ SHORT unpackme.004010A3
```

i sledeci loop:

```
004010DB  |> /8033 11          XOR BYTE PTR DS:[EBX],11
004010DE  |. |83E9 01          SUB ECX,1
004010E1  |. |43               INC EBX
004010E2  |. |83F9 00          CMP ECX,0
004010E5  |.^\75 F4            JNZ SHORT unpackme.004010DB
```

Sada imamo dovoljno informacija da bismo mogli da napravimo patch. Iz ova dva loopa smo saznali da se adrese od 004010F5 xoruju prvo sa 44h a onda sa 11h. Posto posle dekripcije bajtovi od 004011B4 do 004011C7 moraju da budu 90 uradicemo reversni xor **90h xor 11h xor 44h = C5h** a kao rezultat cemo dobiti bajt koji treba da se nalazi na svim adresama izmedju 004011B4 i 004011C7. Ove izmene mozemo da uradimo pomocu Ollyja i da ih snimimo. Kada ovo uradimo startovacemo novosnimljeni fajl i videcemo da smo




zaboravili na cinjenicu da ovaj kripter ima mogucnost provere modifikacije koda ispod adrese 004010F5. Ali ono sto ovaj kripter nema je mogucnost provere CRCa iznad adrese 004010F5 a bas ovde se nalazi kod za proveru modifikacije sekcije koda. Posto bi bilo previse lako samo modifikovati skok JE u JNE ja sam kao uslov zadatka postavio da se ovaj skok ne sme modifikovati. Iako ovo zvuci komplikovano uopste nije. Otvoricemo patchovani program pomocu Ollyja i doci cemo do CMP komande koja se nalazi odmah iznad ovog skoka. Ovde smo:

```
00401062   . 81FA B08DEB31        CMP EDX,31EB8DB0
```

Primeticemo da se sadrzaj EAXa (*F9B35572*) poredi sa 31EB8DB0. Stoga cemo samo logicno promeniti CMP komandu u

```
00401062     81FA 7255B3F9         CMP EDX,F9B35572
```

Pre nego sto ovo uradimo moramo da nadjemo kako se adresa na kojoj se nalazi CMP komanda dekriptuje. Odgovor lezi ovde:

```
004010BE  |.  BB 07104000          MOV EBX,unpackme.00401007
004010C3  |.  B9 7F000000          MOV ECX,7F
004010C8  |>  8033 07              /XOR BYTE PTR DS:[EBX],7
004010CB  |.  83E9 01              |SUB ECX,1
004010CE  |.  43                   |INC EBX
004010CF  |.  83F9 00              |CMP ECX,0
004010D2  |.^ 75 F4                \JNZ SHORT unpackme.004010C8
```

Odnosno dekriptcija bajtova izgleda ovako:

```
86 xor 7 = 81          81 xor 7 = 86
FD xor 7 = FA          FA xor 7 = FD
B7 xor 7 = B0          72 xor 7 = 75
8A xor 7 = 8D          55 xor 7 = 52
EC xor 7 = EB          B3 xor 7 = B4
36 xor 7 = 31          F9 xor 7 = FE
```

gde je prva kolona originalna dekripcija a druga patchovana enkripcija. U prevodu treba samo patchovati bajtove na adresi 00401062 u 86FD7552B4FE da bismo prilikom dekripcije na istoj adresi dobili:

```
00401062     81FA 7255B3F9         CMP EDX,F9B35572
```

Posle ovog patchovanja mozemo pokrenuti patchovani fajl i videti da se na ekranu sada ne pojavljuje NAG. Dakle, uspeli smo !!!

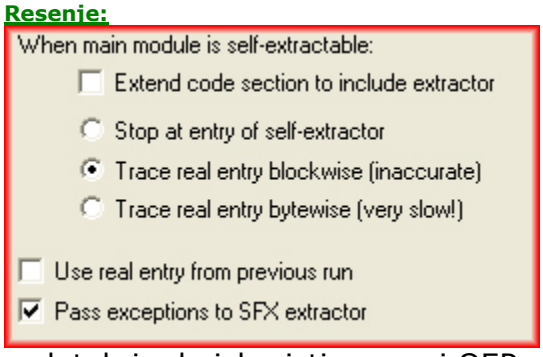
# Unpacking Obsidium 1.2


Iako je u knjizi objasnjeno kako se otpakuje dosta velik broj pakera u ovom nightmare delu knjige cete morati da otpakujete paker koji do sada niste imali prilike da sretnete u ovoj knjizi.

**Zadatak:**

Otpakovati fajl ...\Cas12\crackme.Obsidium12.exe tako da se u fajl PE (*PEid i slicnima*) identifikatorima prikaze pravo stanje a ne identifikacija pakera. Fajl posle otpakivanja ne sme prikazivati Obsidium NAG poruku i mora raditi bez izvrsavanja koda pakera Obsidium.

**Resenje:**




Iako sam ovom pakeru dao visu ocenu, tacnije dao sam mu level 4, paker je lak za otpakivanje kada se Olly lepo podesi. Podesite Olly kao na slici. I prvo cete se sresti sa NAGom Obsidiuma a posle toga cete zavrsiti na OEPu. Kao sto se vidi u samom Ollyju Obsidium je "progutao" par bajtova sa OEPa to jest oni su izvrseni negde drugde. Nas zadatak je da iskoristimo ovaj OEP sto bolje znamo i umemo:

```
004012C0  /.  55              PUSH EBP
004012C1  |.  8BEC            MOV EBP,ESP
004012C3  |.  6A FF           PUSH -1
004012C5  |.  68 F8404000     PUSH crackme_.004040F8
004012CA  |.  68 F41D4000     PUSH crackme_.00401DF4          ;  SE handler installation
004012CF  |.  64:A1 0000000>  MOV EAX,DWORD PTR FS:[0]        ;  Real entry point of SFX
```

Iako ste sigurno pomislili da je Olly prilikom traceovanja progutao ove bajtove i da se zbog toga nalazimo na 004012CF umesto na 004012C0. E pa Olly nije nista progutao jer Obsidium bas ovako radi kradju bajtova: Negde se izvrsi sve sa adrese 004012C0 do 004012CF a to se ne radi ovde. Nama samo predstavlja problem to sto ako sada uradimo dump imacemo iskvarene adrese jer se deo koda vec izvrsio a dump moramo da radimo kada se ni jedna komanda pravog nepakovanog ExE fajla ne izvrsi. Ali mozda mozemo da uradimo dump ovde? Pogledajmo malo bolje adrese od 004012C0 do 004012CF da vidimo sta se tu desava. Nista specijalno nema nikakvih CALLova ili operacija sa registrima, imamo samo par PUSH komandi, dakle tipican VC++ OEP. Ono sto sigurno niste znali je da ako se izvrse PUSH komande njihovo stanje se snima u memoriji a to znaci da nema veze da li cemo uraditi dump ovde ili na 004012C0. Dakle uradicemo dump ovde i pokusacemo da popravimo importe sa ImpRecom. Kao sto cete sigurno primetiti ImpRec je nasao OEP i importe ali ne moze da ih identifikuje. Mozete probati i sa obsidiumIAT pluginom ali nista necete postici. Dakle Importe moramo popraviti rucno. Ovo necu ovde objasniti jer zahteva potpuno znanje API referenci i potrajalo bi dosta dugo, stoga cu vam samo dati fajl iat.txt koji sadrzi ceo IAT nepakovanog fajla. Vi samo treba da ga ucitate u ImpRec pomocu opcije Load Tree i uradite Fix dump.

# Cracking & BruteForceing


I kao poslednji Nightmare test za ovo izdanje knjige The Art Of Reversing je posebno napisan jedan jako tezak crackme.

**Zadatak:**

Mete se nalaze u fajlu ...\Cas12\NAG-Crypto.rar. Ove mete treba reversovati na sledeci nacin:

1) Crackovati NAG-RAR.exe da biste dobili .rar password za drugi deo
2) Crackovati NAG-Crypto.exe tako da on ne pokazuje NAG (*bez otpakivanja*)
3) Napisati KeyGenerator za NAG-Crypto.exe fajl!

Srecno!

**Resenje: - Korak 1**

Nasu prvu metu cemo otvoriti pomocu Ollyja i potrazicemo "Bad Cracker" string. Njega cemo naci ovde:

```
Text strings referenced in NAG-RAR:CODE, item 195
 Address=00407FA4
Text string=ASCII "Bad Cracker"
```

Posle dvoklika na ovaj string zavrsavamo ovde:

```
00407F71  |.  81FA 24030000   CMP EDX,324
00407F77  |.  75 2B           JNZ SHORT NAG-RAR.00407FA4
...
00407FA4  |>  68 24804000     PUSH NAG-RAR.00408024              ; /Text = "Bad Cracker"
```

Kao sto vidimo ovde cemo zavrsiti ako se izvrsi JNZ skok. Postavicemo jedan break-point na sam pocetak ovog CALLa. Unecemo kao password 123456789 i pritisnucemo dugme Check. Sa F8 cemo proci kroz kod sve dok ne dodjemo dovde:

```
00407D82  |.  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]       /*Deo 1
00407D85  |.  8A00            MOV AL,BYTE PTR DS:[EAX]
00407D87  |.  8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00407D8A  |.  8A52 03         MOV DL,BYTE PTR DS:[EDX+3]
00407D8D  |.  32C2            XOR AL,DL
00407D8F  |.  8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00407D92  |.  8A52 06         MOV DL,BYTE PTR DS:[EDX+6]
00407D95  |.  32C2            XOR AL,DL
00407D97  |.  25 FF000000     AND EAX,0FF
00407D9C  |.  8945 F4         MOV DWORD PTR SS:[EBP-C],EAX       */
00407D9F  |.  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]       /* Deo 2
00407DA2  |.  8A40 01         MOV AL,BYTE PTR DS:[EAX+1]
00407DA5  |.  8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00407DA8  |.  8A52 04         MOV DL,BYTE PTR DS:[EDX+4]
00407DAB  |.  32C2            XOR AL,DL
00407DAD  |.  8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00407DB0  |.  8A52 07         MOV DL,BYTE PTR DS:[EDX+7]
00407DB3  |.  32C2            XOR AL,DL
00407DB5  |.  25 FF000000     AND EAX,0FF
00407DBA  |.  8945 EC         MOV DWORD PTR SS:[EBP-14],EAX      */
00407DBD  |.  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]       /* Deo 3
00407DC0  |.  8A40 02         MOV AL,BYTE PTR DS:[EAX+2]
00407DC3  |.  8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00407DC6  |.  8A52 05         MOV DL,BYTE PTR DS:[EDX+5]
00407DC9  |.  32C2            XOR AL,DL
00407DCB  |.  8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00407DCE  |.  8A52 08         MOV DL,BYTE PTR DS:[EDX+8]
00407DD1  |.  32C2            XOR AL,DL
00407DD3  |.  25 FF000000     AND EAX,0FF
00407DD8  |.  8945 F0         MOV DWORD PTR SS:[EBP-10],EAX      */
```

Kao sto vidite ja sam podelio ovaj deo koda na vise celina. Zasto? Zato sto ako analizirate kod za dekripciju dole uvidecete da se koristi par adresa koje se racunaju bas ovde. Ove adrese su: [EBP-10],[EBP-C],[EBP-14]. Ove adrese dobijaju svoje vrednosti na sledecim adresama: 00407DD8, 00407DBA, 00407D9C (*MOV DWORD komanda*). Pre nego sto pocnemo predstavicemo uneti serijski broj kao:

| s[1] | s[2] | s[3] | s[4] | s[5] | s[6] | s[7] | s[8] | s[9] |
|---|---|---|---|---|---|---|---|---|
| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 |

**[ Deo 1 ]**
Ovde se racuna adresa [EBP-C] na sledeci nacin:
- AL dobija vrednost prvog slova iz serijskog broja
- DL dobija vrednost cetvrtog slova iz serijskog broja
- AL dobija novu vrednost; AL = AL xor DL
- DL dobija vrednost sedmog slova iz serijskog broja
- AL dobija novu vrednost; AL = AL xor DL
- Rezultat ovog XORovanja za uneti serijski broj 123456789 je 32h.

t[1] = s[1] xor s[4] xor s[7] = 32h

**[ Deo 2 ]**
Ovde se racuna adresa [EBP-14] na sledeci nacin:
- AL dobija vrednost drugog slova iz serijskog broja
- DL dobija vrednost petog slova iz serijskog broja
- AL dobija novu vrednost; AL = AL xor DL
- DL dobija vrednost osmog slova iz serijskog broja
- AL dobija novu vrednost; AL = AL xor DL
- Rezultat ovog XORovanja za uneti serijski broj 123456789 je 3Fh.

t[2] = s[2] xor s[5] xor s[8] = 3Fh

**[ Deo 3 ]**
Ovde se racuna adresa [EBP-10] na sledeci nacin:
- AL dobija vrednost treceg slova iz serijskog broja
- DL dobija vrednost sestog slova iz serijskog broja
- AL dobija novu vrednost; AL = AL xor DL
- DL dobija vrednost devetog slova iz serijskog broja
- AL dobija novu vrednost; AL = AL xor DL
- Rezultat ovog XORovanja za uneti serijski broj 123456789 je 3Fh.

t[3] = s[3] xor s[6] xor s[9] = 3Ch

```
00407DD8  |.  8945 F0         MOV DWORD PTR SS:[EBP-10],EAX          /* Deo 4
00407DDB  |.  8B75 F4         MOV ESI,DWORD PTR SS:[EBP-C]
00407DDE  |.  3375 EC         XOR ESI,DWORD PTR SS:[EBP-14]
00407DE1  |.  3375 F0         XOR ESI,DWORD PTR SS:[EBP-10]
00407DE4  |.  83C6 02         ADD ESI,2                              */
```

**[ Deo 4 ]**
Ovde se racuna poseban registar ESI koji ce nam takodje trebati:

- ESI dobija vrednost [EBP-C]
- ESI se XORuje sa [EBP-14] i sa [EBP-10]
- ESIju se dodaje 2
- Rezultat dela cetiri za uneti serijski 123456789 je 33.

$$ESI = (t[1] \text{ xor } t[2] \text{ xor } [t3]) + 2 = 33h$$

Posto smo rastumacili prvi deo koda preci cemo na drugi deo u kome se pojavljuje string *a55ayUg>sY9d19x* koji se dekriptuje. Pre nego sto pocnemo napisacemo ovaj string ovako:

| d[1] | d[2] | d[3] | d[4] | d[5] | d[6] | d[7] | d[8] | d[9] | d[10] | d[11] | d[12] | d[13] | d[14] | d[15] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 5 | 5 | a | y | U | g | > | s | Y | 9 | d | 1 | 9 | x |

A sada cemo preci na analizu koda celinu po celinu:

```
00407DF4  |.  8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]              /*Sub 1-1
00407DF7  |.  33DB            XOR EBX,EBX
00407DF9  |.  8A18            MOV BL,BYTE PTR DS:[EAX]
00407DFB  |.  33DE            XOR EBX,ESI
00407DFD  |.  335D F4         XOR EBX,DWORD PTR SS:[EBP-C]
00407E00  |.  8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
00407E03  |.  E8 B0BBFFFF     CALL NAG-RAR.004039B8
00407E08  |.  8818            MOV BYTE PTR DS:[EAX],BL                  */
```

Ovo je poddeo 1-1 i predstavlja samo trecinu poddela 1. Ukratko poddeo jedan se odnosi na prva tri karaktera enkriptovanog stringa. Zasto tri? Zato sti se sva tri XORuju po istom principu. Ako analizirate kod primeticete ovo:

$$e[1] = d[1] \text{ xor } ESI \text{ xor } t[1]$$
$$e[2] = d[2] \text{ xor } ESI \text{ xor } t[1]$$
$$e[3] = d[3] \text{ xor } ESI \text{ xor } t[1]$$

gde je e[x] vrednost koju ce dobiti enkriptovani string. Primetite da se enkriptovani string menja posle XORovanja!

```
00407E3A  |.  8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]              /*Sub 2-1
00407E3D  |.  33DB            XOR EBX,EBX
00407E3F  |.  8A58 03         MOV BL,BYTE PTR DS:[EAX+3]
00407E42  |.  33DE            XOR EBX,ESI
00407E44  |.  335D EC         XOR EBX,DWORD PTR SS:[EBP-14]
00407E47  |.  8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
00407E4A  |.  E8 69BBFFFF     CALL NAG-RAR.004039B8
00407E4F  |.  8858 03         MOV BYTE PTR DS:[EAX+3],BL                */
```

Ovo je druga podcelina i odnosi se na druga tri karaktera enkriptovanog stringa. Vaze sledece jednakosti:

$$e[4] = d[4] \text{ xor } ESI \text{ xor } t[2]$$
$$e[5] = d[5] \text{ xor } ESI \text{ xor } t[2]$$
$$e[6] = d[6] \text{ xor } ESI \text{ xor } t[2]$$

```
00407E7F  |.  8858 05         MOV BYTE PTR DS:[EAX+5],BL                /*Sub 3-1
00407E82  |.  8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
00407E85  |.  33DB            XOR EBX,EBX
00407E87  |.  8A58 06         MOV BL,BYTE PTR DS:[EAX+6]
00407E8A  |.  33DE            XOR EBX,ESI
00407E8C  |.  335D F0         XOR EBX,DWORD PTR SS:[EBP-10]
00407E8F  |.  8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]
00407E92  |.  E8 21BBFFFF     CALL NAG-RAR.004039B8
00407E97  |.  8858 06         MOV BYTE PTR DS:[EAX+6],BL                */
```

Kao i drugi deo treci se svodi na ovo:

e[7] = d[7] xor ESI xor t[3]
e[8] = d[8] xor ESI xor t[3]
e[9] = d[9] xor ESI xor t[3]

Sledeca dva poddela se malo razlikuju u odnosu na dosadasnja tri.

```
00407EC7  |.  8858 08          MOV BYTE PTR DS:[EAX+8],BL               /*Sub 4-1
00407ECA  |.  8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
00407ECD  |.  33DB             XOR EBX,EBX
00407ECF  |.  8A58 09          MOV BL,BYTE PTR DS:[EAX+9]
00407ED2  |.  33DE             XOR EBX,ESI
00407ED4  |.  335D F4          XOR EBX,DWORD PTR SS:[EBP-C]
00407ED7  |.  8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
00407EDA  |.  E8 D9BAFFFF      CALL NAG-RAR.004039B8
00407EDF  |.  8858 09          MOV BYTE PTR DS:[EAX+9],BL               */
00407EE2  |.  8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]             /* Sub 4-2
00407EE5  |.  33DB             XOR EBX,EBX
00407EE7  |.  8A58 0A          MOV BL,BYTE PTR DS:[EAX+A]
00407EEA  |.  33DE             XOR EBX,ESI
00407EEC  |.  335D EC          XOR EBX,DWORD PTR SS:[EBP-14]
00407EEF  |.  8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
00407EF2  |.  E8 C1BAFFFF      CALL NAG-RAR.004039B8
00407EF7  |.  8858 0A          MOV BYTE PTR DS:[EAX+A],BL               */
00407EFA  |.  8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]             /* Sub 4-3
00407EFD  |.  33DB             XOR EBX,EBX
00407EFF  |.  8A58 0B          MOV BL,BYTE PTR DS:[EAX+B]
00407F02  |.  33DE             XOR EBX,ESI
00407F04  |.  335D F0          XOR EBX,DWORD PTR SS:[EBP-10]
00407F07  |.  8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
00407F0A  |.  E8 A9BAFFFF      CALL NAG-RAR.004039B8
00407F0F  |.  8858 0B          MOV BYTE PTR DS:[EAX+B],BL               */
```

Vidimo da je princip isti samo se malo razlikuju koraci 2 i 3 u odnosu na prethodne pododeljke. Za sledeca dva puta vaze sledece formule:

e[10] = d[10] xor ESI xor t[1]
e[11] = d[11] xor ESI xor t[2]
e[12] = d[12] xor ESI xor t[3]
e[13] = d[13] xor ESI xor t[1]
e[14] = d[14] xor ESI xor t[2]
e[15] = d[15] xor ESI xor t[3]

I konacno imamo formulu po kojoj se dekriptuje enkriptovani d[x] string. Sledi deo koji se odnosi na proveru a nalazi se ovde:

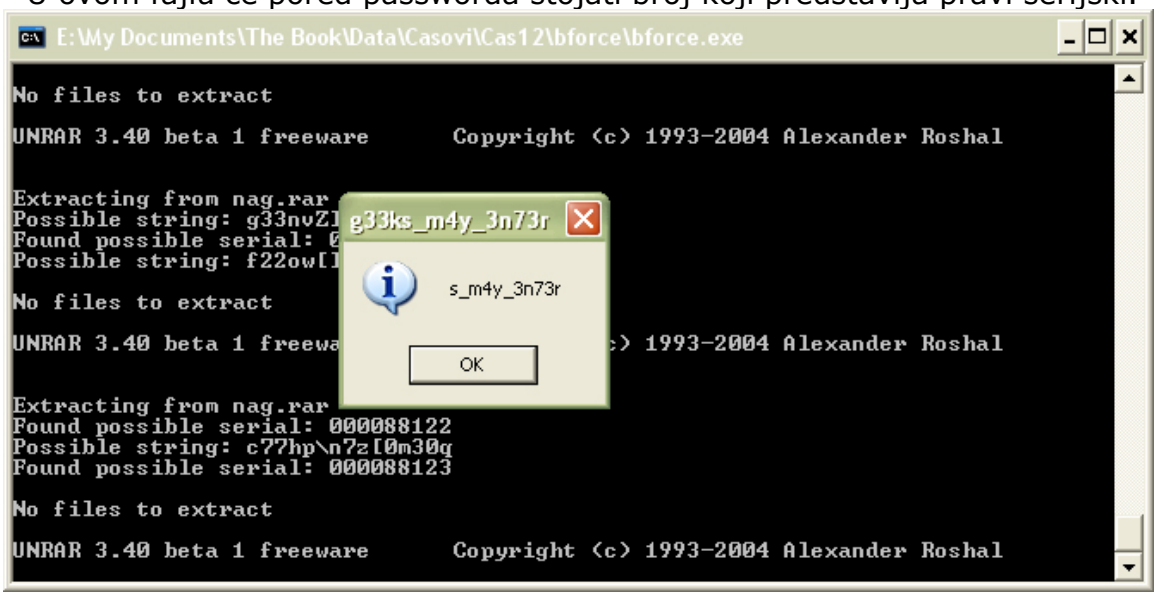
```
00407F5A  |.  33D2             XOR EDX,EDX                              /* Check
00407F5C  |.  B8 01000000      MOV EAX,1
00407F61  |>  8B4D FC          /MOV ECX,DWORD PTR SS:[EBP-4]
00407F64  |.  0FB64C01 FF      |MOVZX ECX,BYTE PTR DS:[ECX+EAX-1]
00407F69  |.  03D1             |ADD EDX,ECX
00407F6B  |.  40               |INC EAX
00407F6C  |.  83F8 0A          |CMP EAX,0A
00407F6F  |.^ 75 F0            \JNZ SHORT NAG-RAR.00407F61
00407F71  |.  81FA 24030000    CMP EDX,324
00407F77  |.  75 2B            JNZ SHORT NAG-RAR.00407FA4               */
```

Kao sto se vidi ovaj loop se ponavlja 9 puta i unutar tog loopa se sabiraju sva slova iz “dekriptovanog” stringa i njihov zbir se poredi sa 0x324. Ovde se namece jedan problem... Naime sabiraju se samo 9 slova iz imena a string je dugacak 15 karaktera. Ovo znaci da ce postojati veliki broj tacnih serijskih brojeva ali da ce samo jedan dekriptovati string na bas pravi nacin. Ovo takodje znaci da ce program za neke serijske govoriti da su tacni iako to

nisu... Na primer probajte ovaj serijski broj 000000092 i program ce vam dati jedan password ali ovaj password nije tacan!
Posto se pred nama nalazi ne mali problem moracemo da ga resimo... Jasno je da cemo morati da koristimo jednu vrstu bruteforceinga da bismo dosli do tacnog serijskog broja... Ali kakvu brute treba uraditi? Posmatracemo ovo kao niz problema:
1) Napraviti algoritam koji ce pronaci sve validne serijske brojeve
   + Ovo je moguce uraditi ali tacnih serijskih brojeva ima jako mnogo
2) Napisati program koji ce za pronadjene serijske brojeve pokusati da otpakuje .rar arhivu
   + I ovo je lako ali je malo teze pronaci tacan password za koji je .rar arhiva otpakovana.

Iz ovih razloga ja sam se odlucio na sledece korake:
- Napisati bruter u C++ koji ce pronaci sve serijske i zabeleziti ih u fajl
- Ovaj bruter ce preko konzolnog fajla UnRAR.exe pokusati da otpakuje fajl
- Modifikovati UnRAR.exe tako da on kao MessageBox prikaze tacan serijski za slucaj kada se fajl NAG-Crypto.exe otpakuje kako treba.

Nazalost ovi koraci su preveliki da bi se opisivali u ovoj knjizi pa cu ja samo ostaviti source i modifikovane fajlove u podfolderu koji se nalazi ovde ...\Casovi\Cas12\bforce\Data.
Posto se ovi koraci nece objasnjavati reci cu vam samo kako da koristite ovaj bruteforcer.
- Startujte fajl brofce.exe i pritisnite <ENTER>
- Na ekranu ce se vrteti brojevi od 000000000 do 999999999
- Kada program "provali" password on ce se prikazati na ekranu kao MsgBox
- Kada se ovo desi zapisite ga, pritisnite <OK> i zatvorite dos prozor
- NAPOMENA: Serijski moze biti ili sadrzaj ili naslov MsgBox, probajte oba!
- Kada dobijete password, potrazite serijski broj u fajlu dict.txt
- U ovom fajlu ce pored passworda stojati broj koji predstavlja pravi serijski.



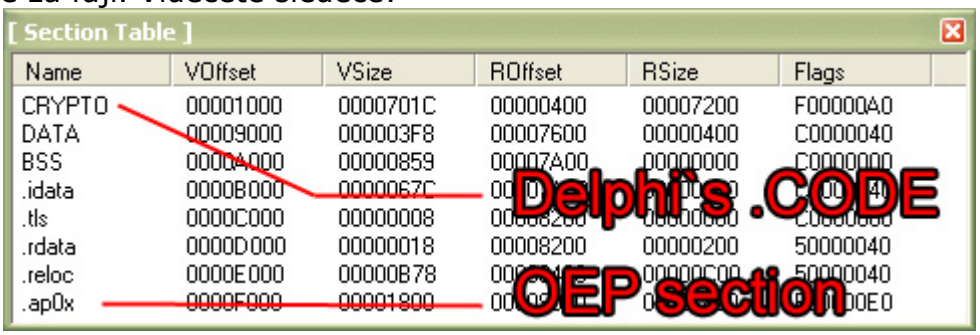

Kao sto se vidi sa slike password za ovu arhivu je g33ks_m4y_3n73r i njega cemo potraziti u fajlu dict.txt. Pronasli smo trazeni password ovde:


b66muYj3~Z5i25u :: 000066189
m99ayUi0}U9j=9v :: 000066286
g33ks_m4y_3n73r :: 000066288
m99ayUi0}U9j=9v :: 000066397

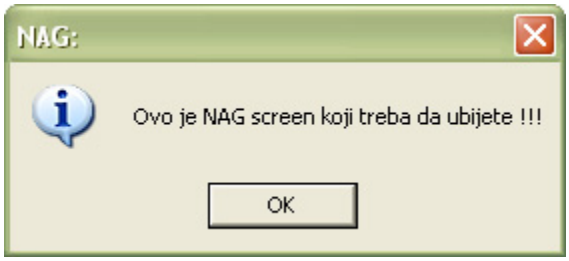
I kao sto se vidi trazeni serijski broj za NAG-RAR.exe je 000066288. Ako unesemo njega u metu program ce prikazati sliku sa strane. Ovo znaci da smo uspeli i da je password za .rar arhivu g33ks_m4y_3n73r. Iako se autor potrudio da nas zaustavi sa velikom kolicinom tacnih serijskih brojeva, u stvari sa nepotpunim informacijama, mi smo uspeli da resimo i ovaj mali problem reversingom i da nadjemo pravi password za arhivu. Stoga cemo u skladu sa passwordom uci. *"137z 3n73r 7h3 4rch1v3 :)"*

**Resenje: - Korak 2**

Pre nego sto pocnete sa reversingom drugog dela potrebno je da iskljucite vas anti-virus jer se NAG-Crypto.exe detektuje kao genericki virus. Naravno ovo nije istina jer da jeste ovakav primer se sigurno ne bi nasao u mojoj knjizi. Dakle kada iskljucite Anti-virus startujte LordPE i pogledajte detalje vezane za fajl. Videcete sledece:




Kao sto vidimo program predstavlja modifikovanu verziju standardnog Delphi exe fajla. Ovo mozete proveriti ako skenirate metu PeIDom. Ono sto se u ovom slucaju razlikuje od Delphija je dodata .ap0x sekcija i reimenovanje prve sekcija koja se u Delphiju oznacava sa .CODE u CRYPTO. Pre nego sto pocnemo sa reversingom ove mete promenicemo Flagove CRYPTO sekcije na E0000020. Posle ovoga mozemo da startujemo ovu metu i vidimo koji je nas zadatak. Kao sto se vidi na sledecoj slici treba da ubijemo NAG.




Ovo bi trebalo da bude lak posao iz razloga sto se svi NAGovi ubijaju jako prosto i iz razloga sto smo ovo uradili jos u drugom poglavlju pa znamo da se ubijanje MessageBox NAGova svodi na NOPovanje svih

PUSH komandi koje prethode CALLu koji poziva Windowsovu MessageBoxA API funkciju. Otvoricemo metu pomocu Ollyja i videcemo ovo:

```
00410556 > $  B8 30054100    MOV EAX,NAG-Cryp.00410530
0041055B   .  B9 25000000    MOV ECX,25
00410560   >  8000 25        ADD BYTE PTR DS:[EAX],25
00410563   .  8028 26        SUB BYTE PTR DS:[EAX],26
00410566   .  8030 27        XOR BYTE PTR DS:[EAX],27
00410569   .  83C0 01        ADD EAX,1
0041056C   .^ E2 F2          LOOPD SHORT NAG-Cryp.00410560
0041056E   .  B8 00044100    MOV EAX,NAG-Cryp.00410400
00410573   .  BA 00000000    MOV EDX,0
00410578   >  0310           ADD EDX,DWORD PTR DS:[EAX]
0041057A   .  3D 54054100    CMP EAX,NAG-Cryp.00410554
0041057F   .  74 03          JE SHORT NAG-Cryp.00410584
00410581   .  40             INC EAX
00410582   .^ EB F4          JMP SHORT NAG-Cryp.00410578
00410584   >  81FA D82F58B4  CMP EDX,B4582FD8
0041058A   .^ 74 A4          JE SHORT NAG-Cryp.00410530
0041058C   .  6A 30          PUSH 30
0041058E   .  68 30064100    PUSH NAG-Cryp.00410630
00410593   .  68 37064100    PUSH NAG-Cryp.00410637
00410598   .  6A 00          PUSH 0
0041059A   .  E8 4940FFFF    CALL <JMP.&user32.MessageBoxA>
0041059F   .  E9 8A000000    JMP NAG-Cryp.0041062E
```

Kao sto vidimo ovde postoji jedan jako dugacak LOOPD koji se nalazi na adresi 0041056C. Da ne bismo izvrsavali ovaj stvarno dugacak LOOP postavicemo jedan break-point ispod nje i pritisnucemo F9. Ovo ce nas odvesti do adrese 0041056E. Dalje cemo nastaviti da izvrsavamo kod sa F8 sve dok ne dodjemo do 00410582. Kao sto vidimo ovde se nalazi jedan JMP skok koji ce nas vratiti gore u kod. Preko ovog skoka vodi samo JE SHORT 00410584 skok, koji ce se izvrsiti samo ako je CMP EAX,00410554, posle cega cemo se naci na adresi 00410584, na kojoj se nalazi komanda CMP EDX,B4582FD8. Sta ovo znaci? Ovo znaci da je program koristio gornji deo koda da izracuna checksum dela koda koji pocinje na adresi 00410400 a zavrsava se ovde 00410554. Ovih poredjenja kroz kod ce biti mnogo i one su tu da provere da li smo, kako i kada patchovali NAG. Posto ne smemo da patchujemo checksum provere onda cemo morati da smislimo gde cemo patchovati usput! Konacno cemo doci do skoka na adresi 0041058A koji ce nas odvesti na deo koda koji se skoro dekriptovao (*u onom prvom LOOPu*):

```
00410530  > /B8 01044100    MOV EAX,NAG-Cryp.00410401
00410533    41              DB 41                    ;  CHAR 'A'
00410534    00              DB 00
00410535    BA              DB BA
00410536    00              DB 00
```

Posto ovaj deo koda nije analiziran samo cemo ga dodatno analizirati pritiskom na CTRL + A posle cega ovaj postaje malo citljiviji.

```
00410530  > /B8 01044100    MOV EAX,NAG-Cryp.00410401
00410535  . |BA 00000000    MOV EDX,0
0041053A  . |B9 2A010000    MOV ECX,12A
0041053F  > |0310           ADD EDX,DWORD PTR DS:[EAX]
00410541  . |8030 29        XOR BYTE PTR DS:[EAX],29
00410544  . |83C0 01        ADD EAX,1
00410547  .^|E2 F6          LOOPD SHORT NAG-Cryp.0041053F
00410549  . |81FA 180C39E2  CMP EDX,E2390C18
0041054F  .^|0F84 D4FEFFFF  JE NAG-Cryp.00410429
00410555  . |C3             RET
```

I kao sto se posle analize vidi i ovaj novi deo koda se koristi za dekripciju i to adresa od 00410401 pa sledecih 12A bajtova. Kao sto vidimo i ovde se vrsi i

dekripcija i provera checksuma (*registar EDX i CMP komanda na adresi 00410549*). Ako se ovaj checksum slaze onda ce se skociti na adresu 00410429, na kojoj se nalazi sledece:

```
00410429  > /55             PUSH EBP
0041042A  ? |8BEC           MOV EBP,ESP
0041042C  ? |83C4 F0        ADD ESP,-10
0041042F  ? |B8 9C7F4000    MOV EAX,NAG-Cryp.00407F9C
00410434  ? |E8 4B40FFFF    CALL NAG-Cryp.00404484
00410439  .^|EB C6          JMP SHORT NAG-Cryp.00410401
0041043B  ? |0000           ADD BYTE PTR DS:[EAX],AL
```

Ovo predstavlja samo jedan deo standardnog Delphi OEPa stoga cemo izvrsiti sav ovaj kod sa F8. Posle izvrsenja JMP komande naci cemo se ovde:

```
00410401  > /B8 FD7F4000    MOV EAX,NAG-Cryp.00407FFD
00410406  . |B9 1B000000    MOV ECX,1B
0041040B  > |83C0 01        ADD EAX,1
0041040E  . |8030 33        XOR BYTE PTR DS:[EAX],33
00410411  . |8030 44        XOR BYTE PTR DS:[EAX],44
00410414  .^|E2 F5          LOOPD SHORT NAG-Cryp.0041040B
00410416  . |A1 10A74000    MOV EAX,DWORD PTR DS:[40A710]
0041041B  . |A3 4CA84000    MOV DWORD PTR DS:[40A84C],EAX
00410420  . |EB 1F          JMP SHORT NAG-Cryp.00410441
00410422    |90             NOP
```

I ovo je vec vise puta vidjeni kod u ovoj meti. Upotreba mu je ista kao i u prethodna dva slucaja, samo sto su ovaj put na meti sledece adrese 00407FFD i 1B sledecih. Posle izvrsenja skoka na adresi 00410439 doci cemo:

```
00410441  > \B8 407F4000    MOV EAX,NAG-Cryp.00407F40     /* Prva dekripcija
00410446  .  B9 04000000    MOV ECX,4
0041044B  >  8030 44        XOR BYTE PTR DS:[EAX],44
0041044E  .  83C0 01        ADD EAX,1
00410451  .^ E2 F8          LOOPD SHORT NAG-Cryp.0041044B  */
00410453  .  B8 487F4000    MOV EAX,NAG-Cryp.00407F48     /* Druga dekripcija
00410458  .  B9 2B000000    MOV ECX,2B
0041045D  >  8030 44        XOR BYTE PTR DS:[EAX],44
00410460  .  83C0 01        ADD EAX,1
00410463  .^ E2 F8          LOOPD SHORT NAG-Cryp.0041045D  */
00410465  .  B8 107F4000    MOV EAX,NAG-Cryp.00407F10     /* Treca dekripcija
0041046A  .  B9 12000000    MOV ECX,12
0041046F  >  8030 44        XOR BYTE PTR DS:[EAX],44
00410472  .  83C0 01        ADD EAX,1
00410475  .^ E2 F8          LOOPD SHORT NAG-Cryp.0041046F  */
00410477  .  B8 107F4000    MOV EAX,NAG-Cryp.00407F10     /* Cetvrta dekripcija
0041047C  .  B9 12000000    MOV ECX,12
00410481  .  BA 00000000    MOV EDX,0
00410486  >  0310           ADD EDX,DWORD PTR DS:[EAX]
00410488  .  83C0 01        ADD EAX,1
0041048B  .^ E2 F9          LOOPD SHORT NAG-Cryp.00410486  */
0041048D  .  81FA 45CE9313  CMP EDX,1393CE45              /* checksum provera
00410493  .  74 0F          JE SHORT NAG-Cryp.004104A4    */
```

Analizirao sam ovaj deo koda i ovde je sve jasno, stoga posle izvrsavanja JE skoka na adresi 00410493 zavrsicemo ovde:

```
004104A4  > \8B15 94044100 MOV EDX,DWORD PTR DS:[410494]  /* Provera 410494
004104AA  .  81EA 00909090 SUB EDX,90909000
004104B0  .  83C2 1A       ADD EDX,1A
004104B3  .  83FA 29       CMP EDX,29
004104B6  .^ 75 E1         JNZ SHORT NAG-Cryp.00410499    */ ExitProcess
004104B8  .  B8 807D4000   MOV EAX,NAG-Cryp.00407D80      /* Dekripcija jedan
004104BD  .  B9 19000000   MOV ECX,19
004104C2  >  8030 29       XOR BYTE PTR DS:[EAX],29
004104C5  .  83C0 01       ADD EAX,1
004104C8  .^ E2 F8         LOOPD SHORT NAG-Cryp.004104C2  */
004104CA  .  8B15 807D4000 MOV EDX,DWORD PTR DS:[407D80]  /* Provera 407D80
```

```
004104D0  . 81EA 004DFC8A  SUB EDX,8AFC4D00
004104D6  . 83EA 1A        SUB EDX,1A
004104D9  . 83E2 1C        AND EDX,1C
004104DC  . 83FA 10        CMP EDX,10
004104DF  .^ 75 B8         JNZ SHORT NAG-Cryp.00410499        */ ExitProcess
004104E1  . B8 D47D4000    MOV EAX,NAG-Cryp.00407DD4          /* Dekripcija dva
004104E6  . B9 2C000000    MOV ECX,2C
004104EB  > 8030 10        XOR BYTE PTR DS:[EAX],10
004104EE  . 83C0 01        ADD EAX,1
004104F1  .^ E2 F8         LOOPD SHORT NAG-Cryp.004104EB      */
004104F3  . B8 307E4000    MOV EAX,NAG-Cryp.00407E30          /* Dekripcija tri
004104F8  . B9 17000000    MOV ECX,17
004104FD  > 8000 02        ADD BYTE PTR DS:[EAX],2
00410500  . 83C0 01        ADD EAX,1
00410503  .^ E2 F8         LOOPD SHORT NAG-Cryp.004104FD      */
00410505  . B8 00044100    MOV EAX,NAG-Cryp.00410400          /* Dekripcija cetiri
0041050A  . B9 03010000    MOV ECX,103
0041050F  . BA 00000000    MOV EDX,0
00410514  > 0310           ADD EDX,DWORD PTR DS:[EAX]
00410516  . 83C0 01        ADD EAX,1
00410519  .^ E2 F9         LOOPD SHORT NAG-Cryp.00410514      */
0041051B  . 81FA 3A80C17C  CMP EDX,7CC1803A                   /* checksum
00410521  .- 0F84 D77AFFFF JE NAG-Cryp.00407FFE               */ JMP to OEP
00410527  .^ E9 6DFFFFFF   JMP NAG-Cryp.00410499              */ ExitProcess
```

Kao sto vidite i gornji deo koda je analiziran sa strane. Ovde se nalazi veci deo provera i dekripcija gornjeg dela koda iz sekcije CRYPTO. Ono sto je bitno da uradite da biste konacno stigli do OEPa je da postavite break-point na adresu 00410521 i pritiskom na F9 dodjete do nje. Kao sto vidimo ovde se nalazi jos jedna checksum provera, ispod koje se nalazi jedan JE skok koji uvek mora da se izvrsi ako zelimo da dodjemo na OEP. Iz ovog razloga cemo izmeniti ovaj JE skok u JNE. Ovo cemo posle izvrsavanja skoka vratiti na JE jer ni ovaj deo koda ne sme biti patchovan zbog raznih checksum provera koje proveravaju ovaj deo koda. I evo kako izgleda sam OEP:

```
00407FFE   6A 00           PUSH 0
00408000   68 5C7E4000     PUSH NAG-Cryp.00407E5C
00408005   6A 00           PUSH 0
00408007   6A 64           PUSH 64
00408009   FF35 4CA84000    PUSH DWORD PTR DS:[40A84C]          ; NAG-Cryp.00400000
0040800F   E8 9CC5FFFF     CALL <JMP.&user32.DialogBoxParamA>
00408014   E8 ABB4FFFF     CALL NAG-Cryp.004034C4
```

Posto je dekriptor zavrsio svoj posao ostaje nam samo da pronadjemo gde se to nalazi NAG gore u kodu.

```
00407F10   6A 40           PUSH 40
00407F12   68 407F4000     PUSH  00407F40  "NAG:"
00407F17   68 487F4000     PUSH 00407F48   "Ovo je NAG screen koji treba da ubijete !!!"
00407F1C   53              PUSH EBX
00407F1D   E8 C6C6FFFF     CALL <JMP.&user32.MessageBoxA>
```

I pronasli smo ga ovde. Sada treba nekako da patchujemo ovaj kod pre nego sto se NAG prikaze na ekran. Posto ovo ne smemo da radimo pomocu otpakivanja mete, a posto je lokacija na kojoj se nalazi NAG visestruko zasticena enkripcijom i checksum proverama moracemo da patchovanje izvrsimo ovako.

Presrescemo DialogBoxParamA funkciju jer se ona izvrsava odmah na OEPu. Ovo se radi veoma lako ako samo analizirate parametre koje API DialogBoxParamA uzima. Kao sto se vidi iz podataka gore WM_MESSAGE loop se nalazi na adresi 00407E5C. Ok, otici cemo na tu adresu i pocecemo sa patchovanjem.

Prvo cemo backupovati stare podatke koji se nalaze na samom pocetku ovog CALLa. Trebaju nam sledeci redovi:

```
00407E5C   55              PUSH EBP
00407E5D   8BEC            MOV EBP,ESP
00407E5F   53              PUSH EBX
00407E60   56              PUSH ESI
00407E61   8B5D 08         MOV EBX,DWORD PTR SS:[EBP+8]
00407E64   33F6            XOR ESI,ESI
```

Sada treba da ubacimo novi ASM kod koji ce patchovati NAG. Za ovo cemo iskoristiti sekciju .ap0x i prazan prostor na njenom kraju.
Prvo radimo preusmeravanje WM_MESSAGE loopa na .ap0x sekciju. Ovo cemo uraditi tako sto cemo izmeniti PUSH EBP komandu u sledeci ASM kod:

```
00407E5C  - E9 02880000    JMP NAG-Cryp.00410663
```

posle cega cemo otici na adresu 00410663 gde cemo uneti sledece:

```
00410663   ?  A3 8A064100    MOV DWORD PTR DS:[41068A],EAX     /* Backup EAXa
00410668   .  B8 107F4000    MOV EAX,NAG-Cryp.00407F10         EAX = prvom bajtu NAGa
0041066D   >  3D 227F4000    CMP EAX,NAG-Cryp.00407F22         Da li patchovan ceo NAG
00410672   .  74 06          JE SHORT NAG-Cryp.0041067A        Ako jeste skoci na kraj
00410674   .  C600 90        MOV BYTE PTR DS:[EAX],90          Patchuj NAG
00410677   .  40            INC EAX                           Povecaj EAX za jedan
00410678   .^ EB F3          JMP SHORT NAG-Cryp.0041066D       Skoci na pocetak patcha
0041067A   >  A1 8A064100    MOV EAX,DWORD PTR DS:[41068A]     Vrati vrednost EAXu */
0041067F   .  55            PUSH EBP                           /* Backupovan CALL
00410680   .  8BEC          MOV EBP,ESP
00410682   .  53            PUSH EBX
00410683   .  56            PUSH ESI
00410684   .- E9 D877FFFF    JMP NAG-Cryp.00407E61              */ Vrati se u CALL
```

Ovaj kod je detaljno komentarisan sa strane ali cu vam objasniti o cemu se radi:
1) Posto EAX sadrzi broj vitalan za sam CALL prvo cemo ga backupovati negde
2) U EAX stavljamo prvu adresu koju patchujemo
3) Preko MOV BYTE PTR komande memorijski patchujemo NAG
4) Kada su svi NAG bajtovi patchovani vracemo EAXu vrednost
5) Unosimo bajtove koji su zamenlji u samom CALLu kada smo uneli JMP
Ostaje jos samo da kazem da se ovaj patch aktivira prilikom svakog prolaza kroz WM_MESSAGE loop stoga nema protrebe za dodatnim proveravanjem koda. Ako posle patchovanja startujemo metu videcemo da smo uspeli i da NAGa vise nema.

**Resenje: - Korak 3**

I kao finalni korak ovog dugackog reverserskog problema ostalo nam je samo da napravimo jedan keygenerator. Da bismo ovo uradili kada dodjemo do OEPa sa izvrsavanjem programa potrazicemo stringove skrolovanjem na gore i pronaci cemo ovo:

```
00407DD4   68 307E4000     PUSH NAG-Cryp.00407E30            ; ASCII "Cracked ok"
```

A ako odemo malo gore videcemo i kod koji se koristi za generisanje serijskog broja:

```
00407D79   85C0            TEST EAX,EAX
00407D7B   7C 1E           JL SHORT NAG-Cryp.00407D9B                /* Deo 1
00407D7D   40              INC EAX
00407D7E   33D2            XOR EDX,EDX
00407D80   8B4D FC         MOV ECX,DWORD PTR SS:[EBP-4]
00407D83   8A4C11 FF       MOV CL,BYTE PTR DS:[ECX+EDX-1]
00407D87   80F1 2C         XOR CL,2C
00407D8A   81E1 FF000000   AND ECX,0FF
```

```
00407D90   03C9              ADD ECX,ECX
00407D92   8D0C89            LEA ECX,DWORD PTR DS:[ECX+ECX*4]
00407D95   03F1              ADD ESI,ECX
00407D97   42                INC EDX
00407D98   48                DEC EAX
00407D99  ^ 75 E5            JNZ SHORT NAG-Cryp.00407D80                */
00407D9B  - E9 37880000      JMP NAG-Cryp.004105D7                      JMP deo 2
```

Prvi deo ne predstavlja nikakav problem za keygening i evo kako bi on izgledao u C++:

```
unsigned int i,esi,cl,edi;
 esi = 0x5B8;
for(i=0;i<strlen(name);i++){
 cl = name[i];
 cl = cl ^ 0x2C;
 cl = cl + cl;
 cl = cl + (cl * 4);
 esi = esi + cl;
}
```

Drugi deo je malo interesantniji ali i ni on nije preterano komplikovan. Do njega cemo doci kada izvrsimo JMP skok.

```
004105D7   .  A1 1B054100    MOV EAX,DWORD PTR DS:[41051B]
004105DC   .  40             INC EAX
004105DD   .  8BF8           MOV EDI,EAX
004105DF   .  A1 21054100    MOV EAX,DWORD PTR DS:[410521]
004105E4   .  40             INC EAX
004105E5   .  8BF8           MOV EDI,EAX
004105E7   .  A1 84054100    MOV EAX,DWORD PTR DS:[410584]
004105EC   .  40             INC EAX
004105ED   .  8BF8           MOV EDI,EAX
004105EF   .  A1 8A054100    MOV EAX,DWORD PTR DS:[41058A]
004105F4   .  40             INC EAX
004105F5   .  8BF8           MOV EDI,EAX
004105F7   .  A1 49054100    MOV EAX,DWORD PTR DS:[410549]
004105FC   .  40             INC EAX
004105FD   .  8BF8           MOV EDI,EAX
004105FF   .  A1 4F054100    MOV EAX,DWORD PTR DS:[41054F]              /* Bitno
00410604   .  40             INC EAX
00410605   .  8BF8           MOV EDI,EAX
00410607   .  81C7 85000000  ADD EDI,85
0041060D   .^ EB 95          JMP SHORT NAG-Cryp.004105A4                */ Deo tri
```

Ovo je rutinska provera nekih adresa, samo sto je pogresno napisana! Kao sto vidite u EDI registar se stavlja vrednost adresa, ali se ove vrednosti ne dodaju jedna na drugu nego se prilikom svake EDI provere EDI registar brise. Zbog ovoga je bitan samo poslednji deo koji se odnosi na adresu 41054F. Ovaj deo bi u C++ izgledao ovako:

```
 edi = 0xFED48410;
 edi = edi + 0x85;
```

JMP skok na adresi 0041060D vodi do dela tri koji izgleda ovako:

```
004105A4   > /81EF 99090000 SUB EDI,999                                /* Bitno
004105AA   . |81EF F90F0900  SUB EDI,90FF9
004105B0   . |2BFE           SUB EDI,ESI
004105B2   . |B8 00000000    MOV EAX,0
004105B7   . |81C6 25100000  ADD ESI,1025
004105BD   . |6BF6 04        IMUL ESI,ESI,4
004105C0   . |81C6 55050000  ADD ESI,555
004105C6   . |81EF 0000CBFE  SUB EDI,FECB0000
004105CC   . |03F7           ADD ESI,EDI
004105CE   . |6BF6 04        IMUL ESI,ESI,4                            */
004105D1   . |EB 40          JMP SHORT NAG-Cryp.00410613               /* Deo cetiri
```

I ovo je dosta trivijano i u C-u izgleda ovako:

```
 edi = edi - 0x999;
 edi = edi - 0x90FF9;
```

```
 edi = edi - esi;
 esi = esi + 0x1025;
 esi = esi * 4;
 esi = esi + 0x555;
 edi = edi - 0xFECB0000;
 esi = esi + edi;
 esi = esi * 4;
```

I na kraju deo cetiri:

```
00410613   > \68 00040000    PUSH 400
00410618   .  53             PUSH EBX
00410619   .  68 B90B0000    PUSH 0BB9
0041061E   .  A1 50A84000    MOV EAX,DWORD PTR DS:[40A850]
00410623   .  50             PUSH EAX
00410624   .  E8 973FFFFF    CALL <JMP.&user32.GetDlgItemTextA>
00410629   .- E9 8377FFFF    JMP NAG-Cryp.00407DB1
```

koji se koristi za uzimanje podataka iz polja u koje se unosi serijski broj. Posle ovoga program se vraca u deo koda koji poredi dve vrednosti: ESI i uneti serijski broj. Dakle uspeli smo keygenerator izgleda ovako:

```
int Keygen(HWND hWnd)
{
        char name[100]="";
        char serial[64]="";
        char buffer[64]="";

        GetDlgItemText(hWnd,IDC_NAME,name,100);

        if (strlen(name)==0)
        {
         SetDlgItemText(hWnd,IDC_SERIAL,"Please type your name !");
         return 1;
        }

   if (strlen(name)<4)
        {
         SetDlgItemText(hWnd,IDC_SERIAL,"Name must be longer than 3 characters !");
          return 1;
        }
 unsigned int i,esi,cl,edi;
  esi = 0x5B8;
 for(i=0;i<strlen(name);i++){
  cl = name[i];
  cl = cl ^ 0x2C;
  cl = cl + cl;
  cl = cl + (cl * 4);
  esi = esi + cl;
 }
  edi = 0xFED48410;
  edi = edi + 0x85;
  edi = edi - 0x999;
  edi = edi - 0x90FF9;
  edi = edi - esi;
  esi = esi + 0x1025;
  esi = esi * 4;
  esi = esi + 0x555;
  edi = edi - 0xFECB0000;
  esi = esi + edi;
  esi = esi * 4;
  wsprintf(buffer,"%d",esi);
  SetDlgItemText(hWnd,IDC_SERIAL,buffer);
        return 0;
}
```

Konacno u folderu ...\Cas12\bforce\Data\ se nalaze svi podaci bitni za ovaj crackme. I konacno smo uspeli da uradimo sve sto je ovaj crackme trazio od nas i kao sto vidite iako je objasnjenje dugo crackme nije tezak :)

# 13 Tricks of Trade

Ovo je poslednje poglavlje knjige posvecene reversnom inzenjeringu. U njemu ce biti obradjene teme vezane za zastitu softwarea od "lakog" reversnog inzenjeringa. U ovom poglavlju cu se potruditi da vam predocim veoma ceste programerske, ali i crackerske zablude...

# Coding Tricks

Shvatite ovaj deo poglavlja kao deo za programere koji zele da pisu sigurnije aplikacije, koje ce nama, reverserima, zadati dosta glavobolje kako bismo ih reversovali.

## Mark's famous protector's commandments

- Nikada ne koristite imena fajlova ili imena procedura tako da ona sama po sebi imaju smisla za reversere, npr. IsValidSerialNum. Ako vec ne zelite da se odreknete ove dobre programerske navike onda bar iskodirajte vas program tako da on zavisi od iste funkcije, tako da se njenim neizvrsavanjem srusi.
- Ne upozoravajte korisnika da je uneti serijski broj netacan odmah nakon provere. Obavestite korisnika posle dan ili dva, crackeri mrze to :)
- Koristite provere checksumove u EXEu i DLLu. Napravite sistem tako da EXE i DLL proveravaju checksumove jedan drugom.
- Napravite pauzu od jedne ili dve sekunde pre provere validnosti serijskog broja. Ovako cete povecati vreme koje potrebno da se pronadje validan serijski broj putem bruteforceinga (veoma lako ali se ne koristi cesto).
- Popravljajte sami, automatski, svoj software.
- Patchujte sami svoj software. Naterajte svoj program da se modifikuje i tako svaki put poziva drugu proceduru za proveru validnosti serijskog broja.
- Cuvajte serijske brojeve na neocekivanim mestima, npr. u propertyju nekog polja baze podataka.
- Cuvajte serijske brojeve na vise razlicitih mesta.
- Ne uzdajte se u sistemsko vreme. Umesto toga proveravajte vreme kreacije fajlova SYSTEM.DAT i BOOTLOG.TXT i uporedjujte ih sa sistemskim vremenom. Zahtevajte da je vreme vece od prethodnog vremena startovanja programa.
- Ne koristite stringove koji ce obavestiti korisnika o isteku probnog vremena softwarea. Umesto ovoga kriptujte stringove ili ih pravite dinamicki.
- Zavarajte crackerima trag pozivajuci veliki broj CALLova, koriscenjem laznih mamaca, enkriptovanih stringova...
- Koristite enkripcije i sisteme koji nisu bazirani na suvom poredjenju validnog i unetog broja.
- Nikada se ne oslanjajte na klasicne pakere / protektore da ce oni zastititi vasu aplikaciju.
- Mi vam nikada necemo otkriti nase najbolje zastite :)

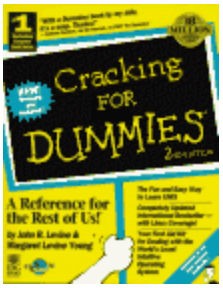
# Cracking Tricks

Shvatite ovaj deo poglavlja kao deo za crackere, to jest kao neku vrstu koraka koje treba primenjivati u reversingu.

## ApOx`s deprotector's commandments

- Uvek prvo skenirajte vasu metu sa PeIDom ili nekim drugim dobrim PE identifikatorom
- Pre pristupa reversingu skupite sto je vise moguce informcija o samoj meti. Ovo radite startovanjem i unosenjem pogresnih podataka u polja za unos serijskog broja.
- Uvek prilikom prvog startovanja programa nadgledajte kojim fajlovima meta pristupa i koje registry kljuceve otvara, cita, ili modifikuje.
- Ako meta ima antidebugging trikove prvo ih sve eliminisite pre nego sto pocnete sa reversingom.
- Ako meta koristi NAGove iskoristite ReSHacker ili W32dasm da pronadjete IDove dijaloga koji se koriste kao NAG. Kada saznate ovu informaciju bice vam lakse da patchujete kod.
- Nikada ne koristite ResHacker u cilju uklanjanja dijaloga. Ovo ce vam samo ukinuti mogucnost pravljenja patcha ili ce ga povecati za veci broj bajtova.
- Ako ste se odlucili da patchujete program uvek pronadjite najjednostavnije resenje koje ce modifikovati najmanji broj bajtova.
- Pri trazenju serijskog broja uvek trazite karakteristicne stringove, tipa "serial", "register", "valid",...
- Prilikom trazenja serijskih brojeva ili drugih provera uvek prvo postavite break-pointe na adekvatnim API pozivima pre nego sto pocnete sa trazenjem serijskog broja.
- Ako je meta zapakovana uvek je rucno raspakujte. Postoji veliki broj pakera koji se moze reversovati i direktno u memoriji ali je uvek lakse da se prilikom restarta programa krene od otpakovanog dela, da ne biste uvek trazili OEP iz pocetka, plus posle popravke importa moci cete da postavljate breakpointe na njih.
- Uvek traceujte u CALLove koji se nalaze blizu provere seriskog broja ili poruke o pogresnom serijskom broju.
- Detaljno analizirajte kod koji se nalazi u istom CALLu kao i kod za prikazivanje poruke o pogresnom seriskom broj.
- Uvek testirajte da li vasi crackovi rade uvek, na svim kompjuterima, da li rade u neogranicenom vremenskom periodu, da li imaju online proveru tacnosti serijskog broja...
- Nikada javno ne publikujte vase patcheve. Ovo podilazi pod Zakon o krsenju intelektualne svojine i autorskih prava, Zakon o pirateriji zbog cega se krivicno odgovara. Upozoreni ste....

# Only Fools and Horses




Ova strana sadrzi veliki broj zabuna vezanih za reversni inzenjering i njegovo polje delovanja. Vecina ovih gluposti je dosla iz usta samih programera "dobrih" zastita. Jedno je sigurno a to je da sam bio sarkastican sa svakim od ovih citata :)

- Password se ne moze vratiti ako se nalazi iza zvezdica *******
- Serijske brojeve je najbolje cuvati u registryju pod sistemskim granama.
- Nije lose koristiti jedan kljuc ili promenljivu koja ce govoriti programu da li je registrovan ili ne.
- Source kod programa se ne moze vratiti iz kompajlovanog exe fajla.
- Nije moguce otkriti stringove koje sam koristio/la unutar samog exe pa se provera passworda moze raditi ovako: if pwd = "mypwd" then ...
- Nema potrebe traciti vreme na kodiranje zastite.
- Ok je distribuirati software sa iskljucenim opcijama, tako ce korisnici ustedeti vreme za download cele verzije.
- Crackeri plate za originalni serijski broj a onda ga puste na internet
- Crackeri ne mogu da nadju serijski broj ako ga cuvam u nekom .dll fajlu u sistemskom direktorijumu....
- Ko ce da crackuje bas moju aplikaciju...

Glupost je neunistiva....

# Cracker`s guide

Ovo je samo mali podsetnik svima kako bi pravi crackeri trebali da se ponasaju.

- ✓ Dobro poznajte "svoje" alate. Svaki cracker je dobar koliko i alati koje koristi i pravi!
- ✓ Nikada ne ripujte (kradite) tude patcheve
- ✓ Nikada ne koristite tudje keygeneratore da biste poturili serijski broj kao vas fishing
- ✓ Nikada ne reversujte program posle koriscenja tudjeg patcha
- ✓ Nikada ne pisite keygeneratore i patcheve na osnovu tudjih tutorijala
- ✓ Uvek prvo pokusajte sami da reversujete aplikaciju. Ne odustajte ni posle vise sati mucenja :) ovo je draz reversinga, verujte mi na rec
- ✓ Uvek se informisite o najnovijim zastitama
- ✓ Svoje znanje produbljujte na crackmeima [http://www.crackemes.de]
- ✓ Nikada ne objavljujte svoje crackove, pravite ih samo za sebe...

# F.A.Q.

Ovo su odgovori na cesto postavljana pitanja vezana za reversni inzenjering, u vezi sa ovom knjigom ali i odgovori na cesto postavljana pitanja koja zavrsavaju u mom mailboxu.

**Q:** Da li je ova knjiga legalna?
**A:** 100% odgovor je DA, iz razloga sto se vrsi reversovanje crackmea (*meta specijalno pravljenih za reversing*) pomocu freeware i trial alata.

**Q:** Da li knjigu mogu citati preko reda?
**A:** Ako ste apsolutni pocetnik moj odgovor je jednostavno NE. Knjiga je napisana tako da gradacijski vodi od laksih stvari ka veoma komplikovanim. Stoga citanje knjige od "kraja" nije moguce i imace samo kontra efekat jer je jako malo reverserskih tehnika objasnjeno vise puta!

**Q:** Zasto kada patchujem program preko HIEWa pisem tacku ispred adrese ?
**A:** Zato sto se adrese koje se patchuju nalaze na virtualnim lokacijama a ne na stvarnim fizickim. Da biste videli stvarne fizicke adrese morate uci u edit mode sa F3.

**Q:** Kako da pocnem "pecanje" serijskog broja?
**A:** Prvo sto uvek treba uraditi je potraziti poruke koje se pojavljuju kao poruke o pogresno unetom serijskom broju u string referencama fajla. Ako ovo nema rezultate koje ste ocekivali predjite na postavljanje breakpointa na standardne API pozive.

**Q:** Kako da otpakujem XX.xx paker?
**A:** Ako se navedeni paker ne nalazi u knjizi to znaci da nije standardan stoga ga nisam opisao. Postoji mnogo tutorijala na internetu.

**Q:** Nisam razumeo/la deo knjige koji se odnosi na xxx?
**A:** Siguran sam da sam sve detaljno i potanko objasnio. Ako vam nesto na prvi pogled izgleda tesko potrudite se da se udubite u materiju i da iscitate odredjeno poglavlje vise puta, konsultujte Time table.

**Q:** Pronasao/la sam gresku u pravopisu ili u semantici na strani xx?
**A:** Super, ja sam se trudio da broj gresaka bude sveden na minimum, ali neke sitne greske su mi sigurno promakle. Posaljite mi email na adresu: ap0x.rce@gmail.com (*0 je nula a ne slovo O*)

**Q:** Mislim da je knjiga stvarno super i zeleo/la bih da pomognem u nekom buducem izdanju ove i sledecih knjiga?
**A:** Drago mi je sto tako mislite, ako imate konkretne ideje kako da mi pomognete u pisanju napisite mi email.

**Q:** Mislim da kniga nije nista posebno i da je nisi ni trebao pisati !!!
**A:** A jel te to neko pitao za misljenje... Pritisni ALT+F4 odmah....

**Q:** Da li bi mogao da mi pomognes oko crackovanja programa xx.xx?
**A:** NE, nemam ni vremena ni zelje da crackujem shareware programe stoga ustedite sebi vreme pisuci mi email ove sadrzine.

**Q:** Gde mogu da postavim dodatna pitanja u vezi poglavlja X?
**A:** Sva dodatna pitanja mozete postavljati na forumu predvidjenom za to na sajtu http://ap0x.headcoders.net/forum/

**Q:** Odakle mogu da narucim CD koji sadrzi ovu knjigu?
**A:** CD sa dodatnim materijalom kao i velikim brojem primera na kojim se moze vezbati sadrzaj knjige se moze poruciti sa mog web-sajta http://ap0x.headcoders.net/

**Q:** Moj CD je ostecen i ne mogu da iskopiram fajl xx.xxx sa njega?
**A:** Svi CDovi su verifikovani pre posiljke, u rootu CDa se nalazi file checksum.svf koji sadrzi CRCove svih fajlova na disku i moze se proveriti pomocu Total Commandera. Ako i posle testiranja CD na drugom kompjuteru budete imali problema sa CDom, posaljite mi email i CD ce biti zamenjen.

**Q:** Da li mi mozete pomoci u vezi zastite mog softwarea?
**A:** Iako idealna (*nesalomiva*) zastita ne postoji ja cu se potruditi da uradim sto je moguce bolji posao revizije i ispravke vase zastite. Posaljite mi email da se dogovorimo oko moguce saradnje.

**Q:** Da li ce biti jos izdanja ove knjige?
**A:** Bez ikakve sumlje odgovaram sa DA. Knjiga je tek pocela da opisuje reverserske tehnike i nema nikakve sumnlje da ce doziveti jos par izdanja!

**Q:** Kada ce izaci drugo izdanje knjige?
**A:** Evo izaslo je....

***Pogovor***

Dosli ste do samog kraja ove knige, cestitam vam na strpljenju i volji da je procitate u celosti. Ja sam se trudio da vas kroz ovu knjigu uvedem u svet reversnog inzenjeringa pocevsi od fundamentalnih pojmova, preko najcesce sretanih problema pa do filozofije reversinga. Priznajem da je ovaj projekat iziskivao dosta veoma pozrtvovanog rada i vremena provedenog za kompjuterom ali kada ovako pogledam do sada napisano delo mogu slobodno da kazem da predstavlja jednu zaokruzenu celinu koja ce vas na mala vrata uvesti u "nas" svet.

Svako poglavlje ove knjige predstavlja jednu celinu koja se samo na prvi pogled cini odvojenom od drugih, ali to svakako nije tako. Svako od ovih poglavlja je odabrano tako da se gradacijski nadovezuje na prethodna, uvlaceci vas tako sve dublje i dublje u samu materiju reversinga. Ona najbitnija misao koja se provlaci kroz celu knjigu je krajnje nedefinisana i neizrecena, ali ona predstavlja samu nit reversinga kao nacina razmisljanja. Ovaj stepen svesti i nacina razmisljanja se ne stice lako ali tokom vremena ce vam se podvuci pod kozu i shvaticete da se RCE primenu nalazi svuda oko nas.

Iako ova knjiga predstavlja jednu kompletnu celinu, trenutno nemam osecaj da je ona u potpunosti zavrsena. Mislim,... ne, siguran sam, da ce ova knjiga doziveti jos nekoliko dopunjenih izdanja ali ce se to zasnivati na dve jako bitne osnove: interesovanje stalne publike i zivot stalnog pisca. Hoce li biti treceg izdanja? …

NASTAVICE SE...