PE and ASM for Crackers

The Art of Reversing

by ApOx

PE and ASM for Crackers

by Ap0x

Predgovor drugom izdanju:

Iako se prvo na internetu pojavila knjiga "The Art of Cracking" hronoloski bi mozda bolje bilo da procitate prvo ovu knjigu a tek onda se posvetite citanju knjige "The Art of Cracking". Naravno ovakav redosled citanja i nije obavezan i prvobitno i nije namenjeno da se cita ovako, ali posle prelistavanja svoje prve knjige o RCEu uvideo sam da je jedna ovakva knjiga neophodna kako bi se ispunila praznina koja se moze osetiti prilikom citanja prve knjige.

Naravno ova knjiga je namenjena prvenstveno crackerima ali i onima koje ne zanimaju detalji vezani za tacne i nepotrebne delove samog PE formata. Ova knjiga je sebi stavila za cilj da upozna crackere pocetnike sa osnovnim i cesto sretanim problemima prilikom modifikovanja standardnih PE fajlova. Podaci koji se nalaze u ovoj knjizi ce vam pomoci da razumete adrese koje srecete prilikom reversinga, da se upoznate sa sekcijama standardnih PE fajlova i da konacno modifikujete bilo koji PE fajl prema svojim potrebama...

Ali ova knjiga se ne bavi samo PE stukturom, drugi deo ove knjige ce vas dosta detaljno uvesti u svet ASM komandi, kako bi ste sa sto vecom lakocom resavali sve reverserske probleme. Ovaj deo knjige je logicki nastavak mog teksta ASM 4 Crackers ali predstavlja mnogo vise od onoga sto se nalazi u tekstu ASM 4 Crackers. Ovo izdanje nije samo dopunjeno i prepravljeno nego je napisano tako da svi mogu da razumeju i sa lakocom koriste osnovne ASM komande. Ova knjiga vas nece nauciti kako da pisete ASM programe ali ce vas nauciti kako da ih reversujete...

Index of PE and ASM for Crackers

01 What is PE?

Kao prvo poglavlje u knjizi ovo poglavlje ima za cilj da vam objasni same osnove PE fajla i da vam razjasni cesto postavljana pitanja vezana za PE.

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.

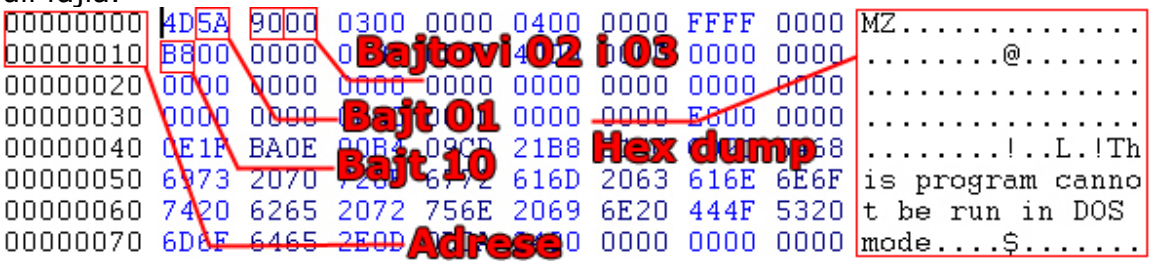
PE ExE files - Intro

Svaki klasican PE fajl pocinje uvek sa dva ista bajta, sa MZ ili sa 4D5A heksadecimanlo. 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....$.......
```

(slika 01.00 - Pe fajl u Hex editoru)

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 hexadecimalnih brojeva u ASCII, naravno pod uslovom da postoji ASCII slovo sa kodom koji koresponduje hexadecimalnoj vrednosti. Osnovna ASCII tabela se sastoji od 127 karaktera ili 7F ako broj karaktera predstavljamo hexadecimalno. 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.



(slika 01.01 - Pe fajl u Hex editoru)

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 kojim se nalaze bajtovi. Posmatrajte ovo kao da su upitanju 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 hexadecimalni a ne decimalni brojevi. Posto se hexadecimalni brojevi znatno razlikuju od decimalnih potrebno ih je pretvarati, a najbolji program za to je obican Windowsov calculator.

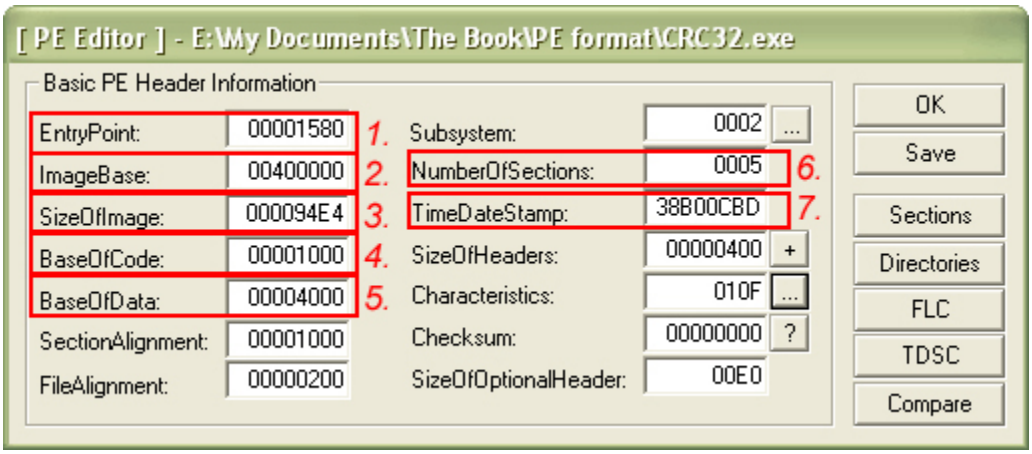
(slika 01.02 - Windows Calculator)

Kao sto se vidi na slici potrebno je samo selektovati dugme Hex umesto dugmeta Dec i uneti zeljeni hexadecimalni broj. Probajte sa brojem 90h, posle cega samo treba ponovo izabrati dugme Dec posle cega broj 90h postaje 144. Razumevanje hexadecimalnih brojeva je kljucno za reversere jer se svi problemi zasnivaju na hexadecimalnim 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 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 korespodiraju 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 slici 01.01 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 bi ste 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.



(slika 01.03 - LordPE edit)

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.

EnteryPoint je prva a ujedno i jako vazna opcija za reversere. Da bi ste 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 linuju koda koja ce se izvrsiti odmah nakon startovanja programa. Ova prva linija se nazima EnteryPoint 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 dissasemblera 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.

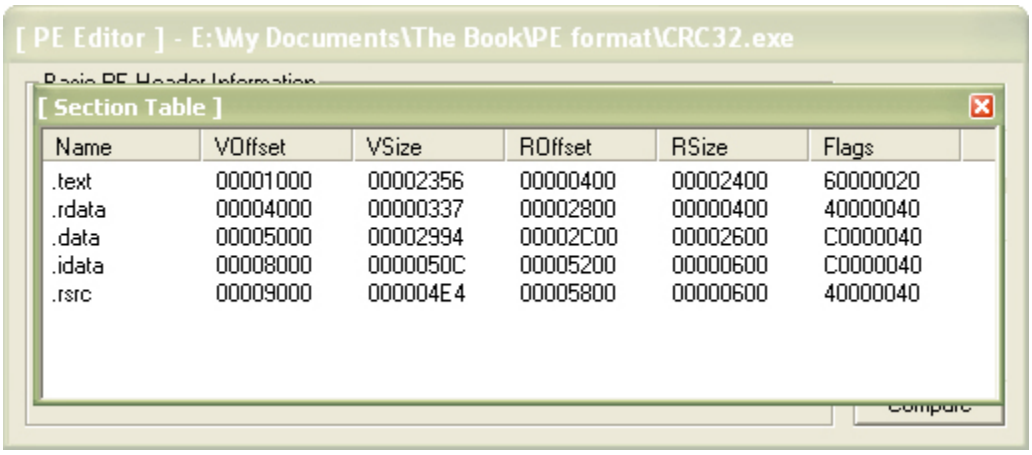
BaseOfCode je cetvrta opcija u nizu i predstavlja Virtualni offset glavne code sekcije koja sadrzi OEP i kod koji se izvrsava. Ovo je glavna sekcija na kojoj se u 99% slucajeva izvrsava reversing.

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 bi ste me pratili pritisnite dumge Sections u LordPEu.



(slika 01.04 - Sekcije PE fajla)

Sekcije - Ono sto ce se pojaviti na ekranu vidi se na slici 01.04. 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 prvu kolonu koja sadrzi imena sekcija. Primeticete da imena svih sekcija pocinju sa tackom. Ovo je definisano standardom kompajlera koji je napravio exe fajl ali ime sekcije ne mora pocinjati sa 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 sta se i kako se modifikuje.

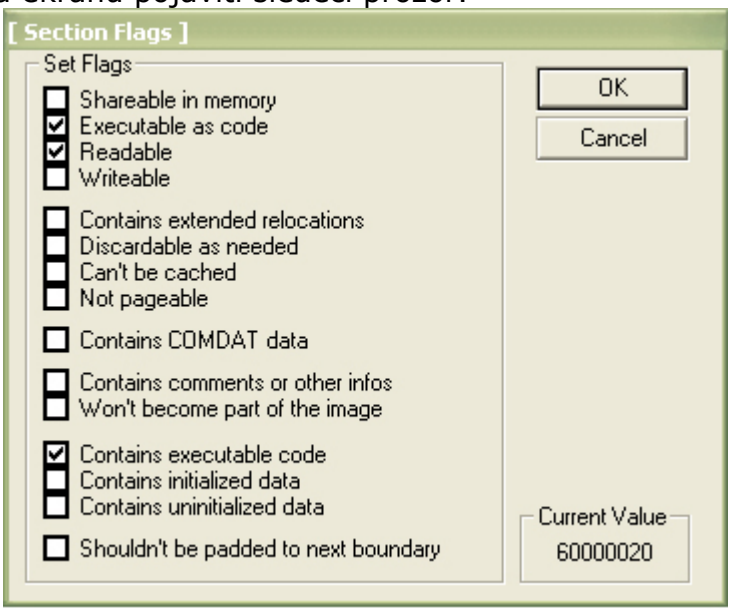
Virtualni Offset - Ovo je druga kolona i izuzetno je bitna za razumevanje virtualnih adresa i njihovo pretvaranje u prave fizicke. Ova kolona sadrzi hexadecimalne brojeve koji predstavljaju prvu adresu koja se nalazi u sekciji. Ovo znaci da ako je upitanju prva sekcija, izvrsna, adrese u njoj pocinju 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 nemora 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 bi 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.

Virtualni 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 hexadecimalni broj ali u zavisnosti od kombinacije ovih brojeva sekcija ima razlicite osobine. Da bi ste 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.



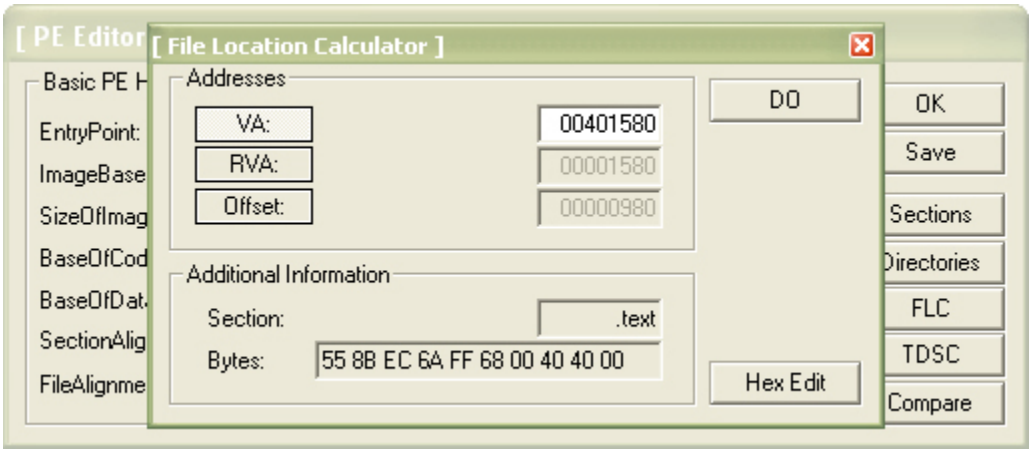
(slika 01.05 - Modifikovanje flagova)

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 upitanju sekcije koje sadrze kod najbolje da karakteristike odmah postavite na E0000020 da bi ste 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 virutalne adrese u fizicke kako bi ste uspesno patchovali vase mete. Da bi ste ovo razumeli otvorite nasu metu pomocu Ollya. 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 EnteryPoint 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 bi smo 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 hexadecimalni 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 hexadecimalnih 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 bi smo 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.



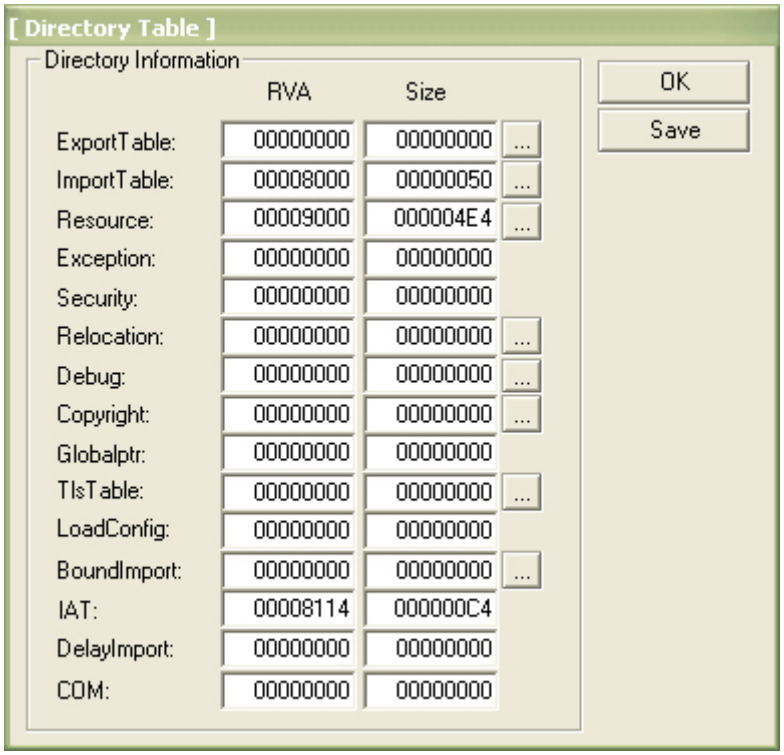
(slika 01.06 - File Offset / RVA calculator)

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 bi smo 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 bi ste videli koje tabele postoje u jednom exe fajlu u LordPeu izaberite opciju Directories sto ce vam prikazati sledece:



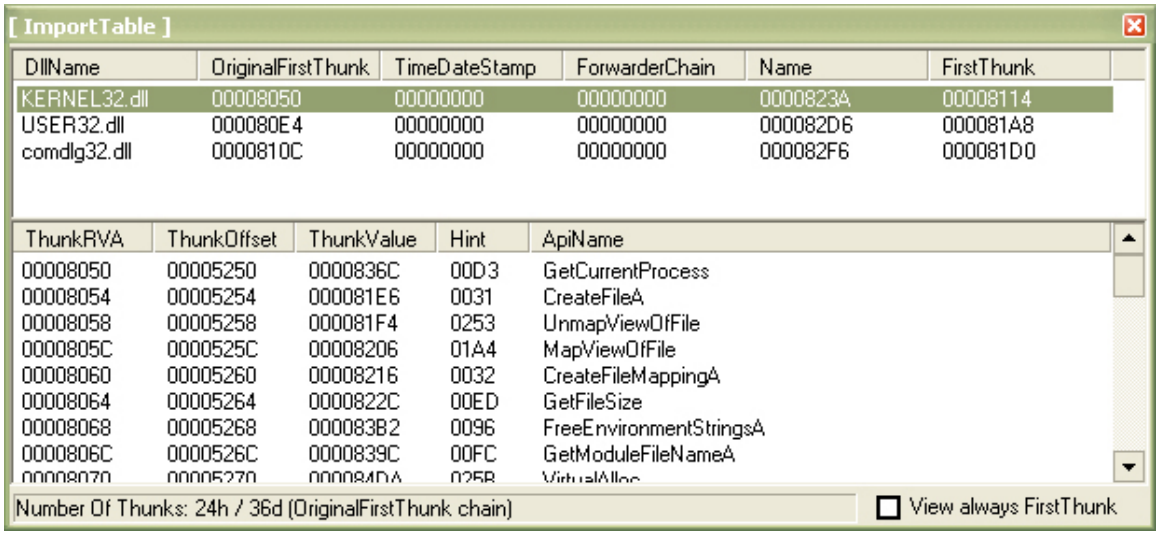
(slika 01.07 - Tablice koje se nalaze u jednom .exe fajlu)

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

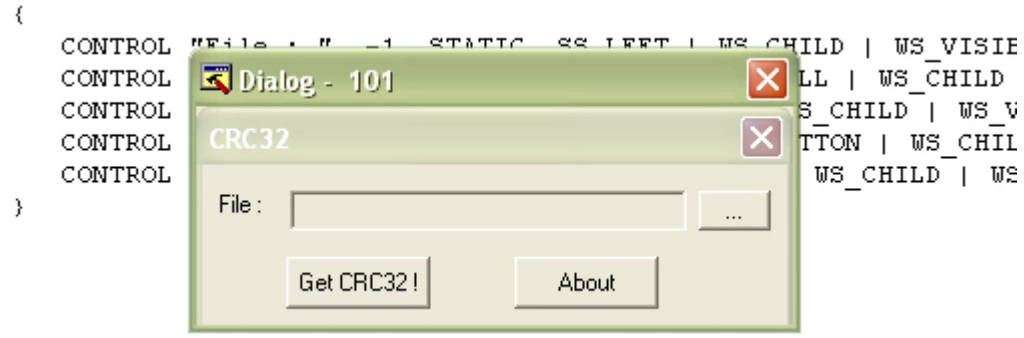
svi .dll fajlovi koji se nalaze u import listi .exe fajla. Importi se mogu citati iz prakticno neogranicenog broja .dll fajlova.
IAT - predstavlja tablicu funkcija koja sluzi za mapiranje svih API funkcija koje se nalaze u jednom PE fajlu.
Import/export tablice se mogu pregledati direktno iz LordPea, samo treba da pritisnemo dugme tri tackice i videcemo sledeci prozor.



(slika 01.08 - Pregled jedne IAT tablice)

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.

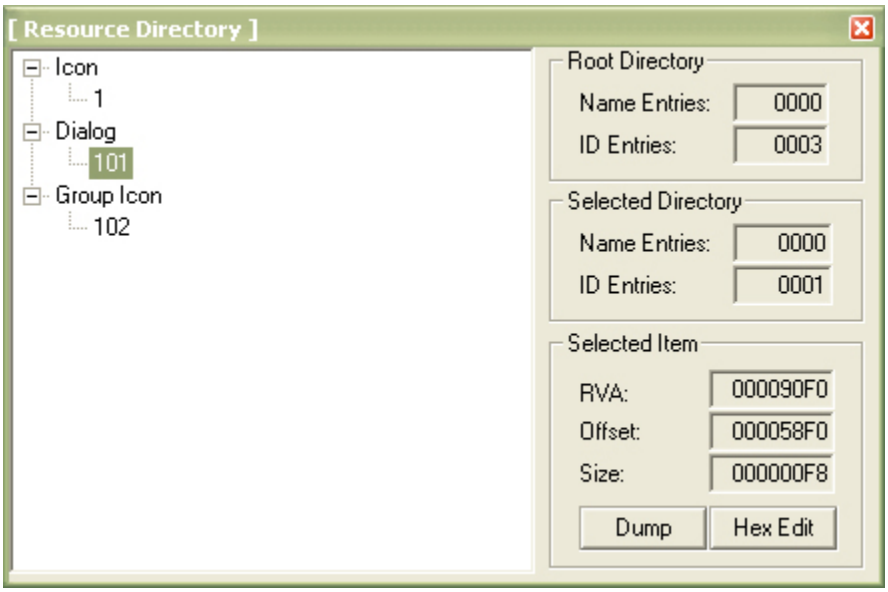
Cetvrta tablica koju smo pomenuli je Resource tablica. Ona se koristi za cuvanje specificnih podataka u jednom PE fajlu. Ovi podaci mogu biti multimedijalnog karaktera (slika,tekst,zvuk...) ili programski podaci kao sto su ikone, dialozi i slicni. Primer jednog klasicnog dialoga se nalazi na slici 01.09. Ovi dialozi predstavljaju standardan Windows interface, ili GUI.



(slika 01.09 - Primer klasicnog dialoga)

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 upitanju import/export tablice potrebno je pritisnuti samo dugme tri tacke pored polja Resources u Directories prozoru. Posle ovoga ce se pojaviti sledeci prozor.



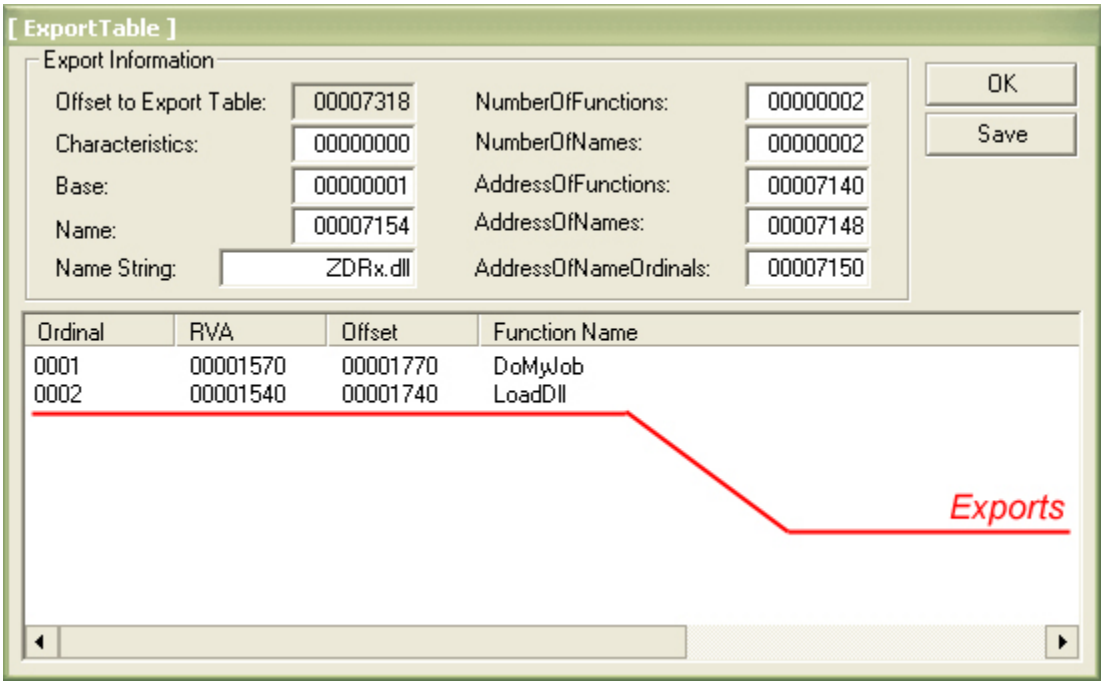
(slika 01.10 - Pregled Resursa jednog fajla)

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 specialnih 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 LordPe-ove opcije Directory moze se videti kako izgleda Export tabela jednog PE fajla:



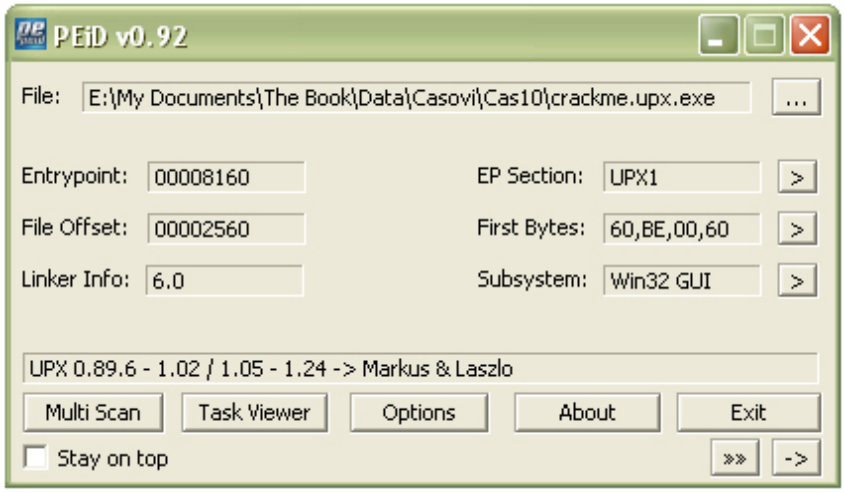
(slika 01.11 - Pregled Exporta jednog dll fajla)

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 LordPe-a 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 bi smo reversovali fajl na pravom mestu.

Unpacking UPX and Fixing IAT

Kao klasican primer packera, ali nikako i protektora, se navodi 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 odpakovati pakovani sadrzaj negde na disk. Isto se desava i sa exe pakerima kao sto je UPX samo sa razlikom sto se pakovani kod odpakuje direktno u memoriju a ne na hard disk.

Krenucemo sa odpakivanjem programa koji se nalazi u folderu Examples a zove se crackme.upx.exe. Prvo sto moramo uraditi je identifikacija pakera sa 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 odpakujemo. 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 odpakivanja nije promenio od verzije 0.89 i da bez ikakvih problema mozemo odpakovati bilo koju veziju UPXa primenjujuci isti metod. Otvoricemo metu pomocu Ollya i videcemo upozorenje da je upitanju zapakovani PE fajl pa da se zbog toga mora paziti gde i kako postavljamo break-pointe. 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              USH 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. Odpakivanje UPXa je veoma lako, postoji veoma mnogo nacina da se odpakuje program pakovan program UPXom ali ja cu vas nauciti najlaksi i najbrzi nacin kako da dodjete do samog OEPa i tu uradite memory dump. Odpakivanje UPXovanih programa se svodi na jednostavno

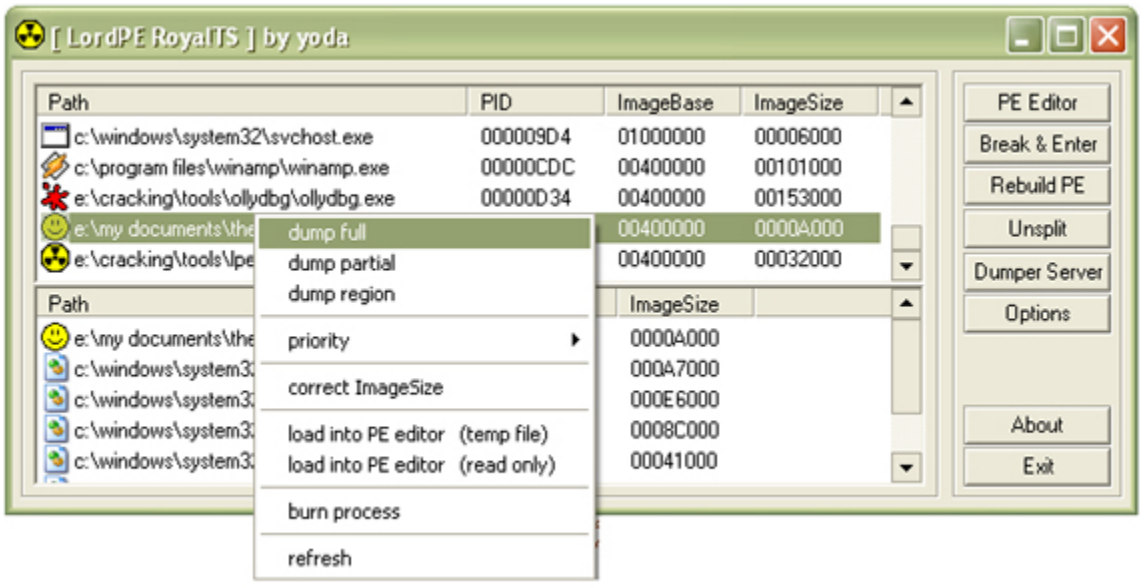
skrolovanje nanize dok ne dodjete do zadnje komande iz koje se nalaze samo neiskosceni 0x00 bajtovi. Ti par poslednjih redova izgledaju ovako:

```
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 upitanju bezuslovni skok (JMP) sama adresa ka kojoj on vodi je adresa do 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 se algoritam za odpakivanje nije izvrsio odpakivanje zapakovanog dela fajla u memoriju i stoga originalni OEP jos ne postoji. Ono sto moramo da uradimo je da nateramo algoritam za odpakivanje da se izvrsi u podpunosti da bi smo onda mogli da uradimo dump memorije. Da bi smo ovo uradili postavicemo break-point 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 break-point. Potrebno je jos samo izvrsiti ovaj skok da bi smo se nasli na OEPu. Pritisnucemo F8 i nacicemo se na OEPu.

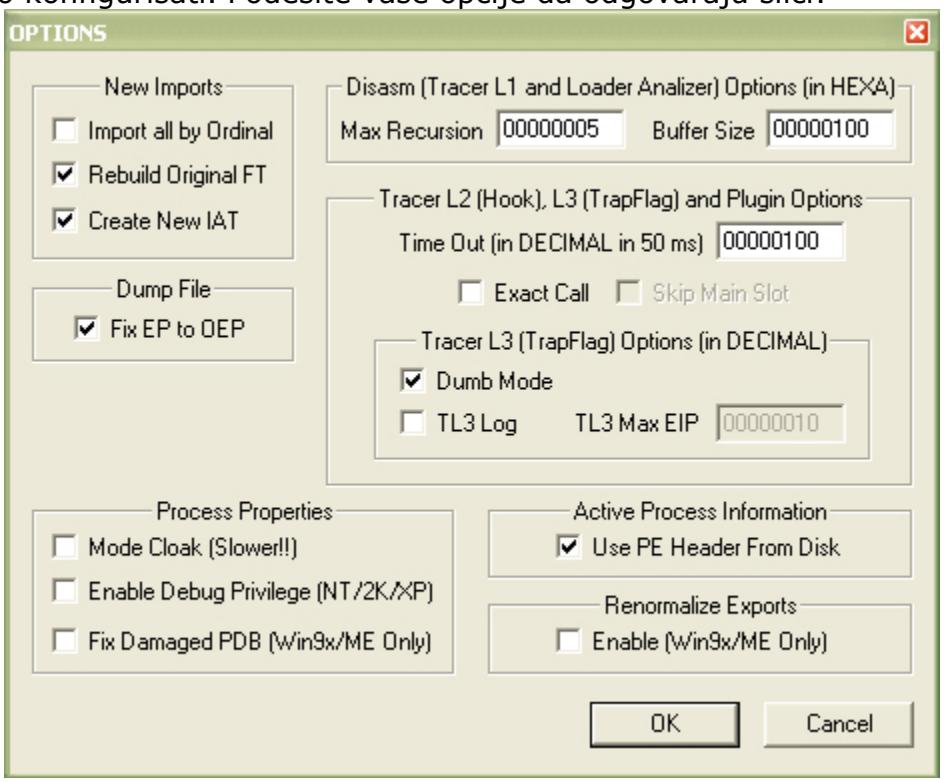
```
004012C0   55                    PUSH EBP
004012C1   8BEC                  MOV EBP,ESP
004012C3   6A FF                 PUSH -1
004012C5   68 F8404000           PUSH crackme_.004040F8
004012CA   68 F41D4000           PUSH crackme_.00401DF4
004012CF   64:A1 00000000        MOV EAX,DWORD PTR FS:[0]
004012D5   50                    PUSH EAX
```

Sada kada se nalazimo na OEPu treba da uradimo memory dump kako bi sacuvali memorisku sliku odpakovanog 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.



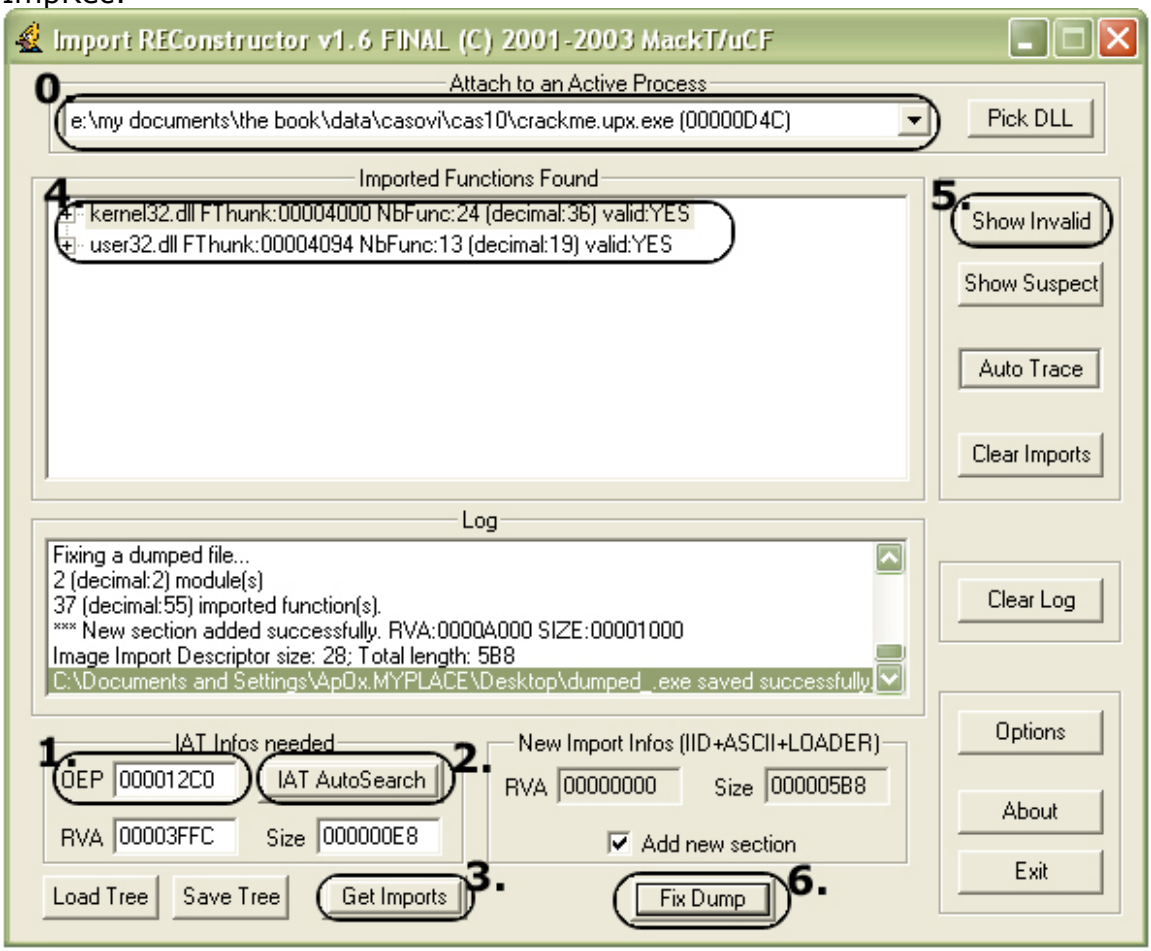
Kada smo snimili memorisku sliku nase mete bilo gde na disk mozemo da zatvorimo LordPE posto nam on nece vise trebati. Pitate zasto nisam koristio

Ollyev dump plugin??? Odgovor je jednostavan: Sve vezije koje sam ikada testirao su imale 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 odpakivanjem UPXa, prevarili ste se. Ostaje nam jos dosta posla kako bi smo naterali metu da se startuje. Zapamtili smo adresu na kojoj se nalazi pravi OEP: 004012C0, ona ce nam trebati kako bi smo 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 odpakovao 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 linije OEPa startujete ImportREconstructer 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 bi smo 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 RVA 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 predhodno. 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.

Ovaj tekst predstavlja deo knjige *The Art of Cracking* i predstavlja glavnu korist od poznavanja PE formata. Ovo znanje je potreba ako zelite da se bavite odpakivanjem zapakovanih aplikacija.

02 ASM BASICS...

Da bi ste se bavili reversingom od izuzetne je vaznosti da dobro poznajete osnovne ASM komande i manipulaciju memorijom. Drugo poglavlje ce pokriti sve osnovne ASM pojmove koji su vam potrebni da bi ste se bavili reversingom.

ASM Basics

ASM je osnova svakog reverserskog problema, zbog cega je potrebno dobro znati makar osnovne ASM komande kako bi ste 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 sreseti tokom reversovanja meta. Pocecemo sa malo matematike...

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

Dodeljivanje vrednosti - je osnovna ASM komanda koja se koristi kako bi nekim promenjivim (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 neznate kako se vrsi sabiranje brojeva u asembleru. Primer sabiranja dve promenjive 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 promenjive 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 promenjivim EAX i ECX kako bi smo 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 bice: 3 * 4 + 4 - 8 * 4 = 32. Naravno treba imati na umu da ako pisemo ASM programe svi brojevi moraju biti u heaxadecimalnom 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 promenjive.
Kao i u matematici i u asembleru mozemo definisati promenjive kojima mozemo dodeljivati bilo koju aritmeticku vrednost. Jedino ogranicenje kada se radi sa asemblerom postoji vec definisani broj promenjivih koje mozemo koristiti. Ove promenjive 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,
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 zavisniti od situacije do situacije. Videli smo da kao i promenjive 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 registi u sebi sadrze 16bitne i 8bitne registre. Ovo znaci da ako na primer EAX sadrzi heaxdecimani broj FF443322 on predstavlja njegovu 32bitnu vrednost. Ali postoje drugi registi 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	--

(tablica 02.00 - Registri)

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 registu 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 regista 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. Ona u ASM izgleda ovako:

```
INC EAX
```

Ova jednostavna ASM komanda je ekvivalent matematickoj EAX = EAX + 1. Sada se sigurno pitate zasto ovo jednostavno ne bi smo 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 registara. 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 bi ste 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 koplikovano ustvari 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 potredno 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 vrenost 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 bi ste 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

(tablica 02.01 - Operacija AND)

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 broj po broj i na osnovu toga formira rezultat. Ako su brojevi 0 i 0, 0 i 1 ili 1 i 0 rezultat ce uvek biti 0, a jedino ako su dva broja koja se porede jedanka 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

(tablica 02.02 - Operacija OR)

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 broj po broj, 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

(tablica 02.03 - Operacija NOT)

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 kranje jednostavna ASM komanda.

XOR - je jako bitna ASM komanda koja se zbog svoje revezibilnosti koristi u svrshe 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

(tablica 02.04 - Operacija XOR)

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 razultat 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 revezibilnosti. 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 bi smo 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) = 5. Naravno primetili ste da je upitanju 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 operacije 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 operacije 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 alplikacija, 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 jednobajtna memoriska 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 jedank 1 onda ce zero flag biti 0. Ovo je bitno jer se vecina provera seriskih brojava zasniva na poredjenu 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 beuzslovne 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 beuzslovni jer nije bitno koja je vrednost zero flaga da bio 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 or 0F85	jne	skoci ako nije jednako
74 or 0F84	je	skoci ako je jednako
EB	jmp	bezuslovni skok
77 or 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 bi smo 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 promenjivu
  int ecx;                    // Definisi celobrojnu promenjivu
  int edx;                    // Definisi celobrojnu promenjivu
  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 pojaljuju na mestima gde se u C-u ali i u drugim programskim jezicima nalaze IF uslovne klauze. Nezamislivi su programi bez poredjenja ili programi bez uslovnih skokova. Na ovakve mete necete naici prilikom reverserske prakse, sto ga 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 predhode 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 predhoditi PUSH komande, a ako funkcija nema ulazne parametare onda CALLu nece predhoditi ni jedna PUSH komanda. Da bi ste 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 zavrsne 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 parametarima:

```
...
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 predhode 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 (*PUSH komande se nalaze i su samom CALLu jer se tako pravi lokalni STACK koji cuva ulazne vrednosti iz registara EAX i ECX, da bi se one na kraju CALLa vratile na prvobitne vrednosti*). 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

Pretposlednje 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 stingovi 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 nacesce 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 bi ste 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 nega 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 bi smo 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 funckije 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 seriskih brojeva zasnivaju na pristupanju delova ili celom stringu nekog unesenog podatka. Ovaj podatak moze biti ime, seriski 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 hexadecimalnom 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 memoriske lokacije koje se mogu ispunjavati brojevima, nad kojima se kasnije mogu izvrsavati matematicke operacije. Kod 32bitnih registara ove memoriske lokacije su se zvale EAX,EBX,… a kod FPU registara te lokacije se zovu ST0,ST1,ST2,…ST7. Njih mozete vidite 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 sopstio 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 nekim promenjivim (ST0,ST1,...) cija su imena definisana konstanta dodelila neka vrednost. Ova komanda izgleda ovako:

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

gde je FLD komanda a source destinacija sa koje se cita niz bajtova koji se pretvara u real promenjivu i smesta po defaultu u prvi raspolozivi memoriski 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 promenjive. Primera radi mogu se sabrati ST0 i ST1 tako da ST0 drzi rezultat sabiranja. Za ovo se koristi komanda:

FADD *destination, source – Dodavanje REAL promenjive*
FIADD *destination, source – Dodavanje INTEGER promenjive [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 promenjive. Primera radi moze se od ST0 oduzeti ST1 tako da ST0 drzi rezultat oduzimanja. Za ovo se koristi komanda:

FSUB *destination, source – Oduzimanje REAL promenjive*
FISUB *destination, source – Oduzimanje INTEGER promenjive [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 promenjive. 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 ume sto FMUL koristi FIMUL onda se drugi parametar prvo pretvara u ExtendedReal a tek onda mnozi sa ST0.

Deljenje - je osnovna FPU matematicka komanda koja se koristi kako bi se podelile dve promenjive. 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 ume sto FDIV koristi FIDIV onda se drugi parametar prvo pretvara u ExtendedReal a tek onda deli sa ST0.

Kvadratni koren – je matetaticka 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 trigonometriske komande koje u ASMu izgleda ovako:

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

Dakle ove komande imju samo jedan patametar koji ujedno predstavlja i source i destination izvrsenja komande. Dakle sinus/kosinus se racuna na registu 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 koristu 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 sva 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;
```

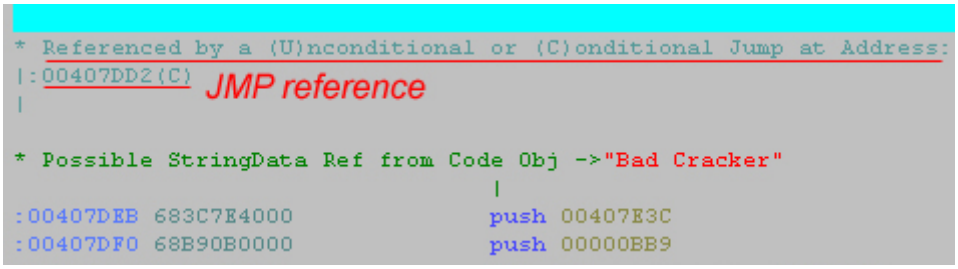
Poredjenje – je poznata ASM komanda koja se koristi za poredjenje dve vrednosti ili dva registra. Ona ima svoj FPU ekvivalent kada se radi o poredjenju integer ili real vrednosti. Ako se porede integer vrednosti onda se koristi FICOM komanda a ako se porede real vrednosti onda se koristi FCOM vrednost.

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 ce se sresti sa nekim od ovih komandi prilikom reversovanja nekih crackmea i/ili nekih enkripcija.

W32Dasm and ASM grouping

Kao poslednje poglavlje za ovu knjigu ostavljena je veza izmedju W32Dasm tumacenja dissasemblovanog fajla i grupisanja ASM funkcija u CALLove i delove koda.

Kao sto je vec napomenuto prilikom dissasemblovanja W32Dasm ce napraviti reference ka pojedinim delovima koda. Sta su to reference? Reference predstavljaju neku vrstu linkovanja delova koda. Ovo znaci da postoje pojedini skokovi u samom PE fajlu koji ce kada se izvrse dovesti izvrsavanje koda do neke adrese. Ako postoji bar jedan ovakav uslovno / bezuslovni skok ili CALL koji vodi ka nekoj adresi, onda on sam pravi jednu referencu. Ovakvih referenci moze biti prakticno neograniceno. Primer jedne takve JMP reference se nalazi na sledecoj slici.



(Slika 02.05 - JMP reference)

Sta je to JMP referenca? Kao sto sam rec rekao postoje dva tipa referenci, CALL i JMP reference.

JMP reference jednostavno predstaljaju adrese na koje ce doci program posle izvrsavanja nekog uslovnog ili bezuslovnog skoka. Spisak ovih skokova se nalazi odmah pored poruke o postojanju referencnih skokova. Primeticete da se pored adrese samog skoka (lokacije gde se skok nalazi) nalazi i odredjeno slovo u zagradi. Ovo slovo moze biti ili (C) ili (U). Prvo slovo, (C), oznacava da je skok koji se nalazi na nekoj adresi uslovni skok i stoga moze biti u obliku JE,JNE i sl. Dok nasuprot slovu (C), slovo (U) oznacava da je skok koji vodi do sledece adrese (00407DEB) bezuslovan, to jest da je najcesce oblika JMP. Ono sto je bitno je da se sve reference odnose na prvu adresu koja se nalazi odmah ispod spiska referenci, sto je u ovom slucaju 00407DEB. Ovo znaci da postoji skok na adresi 00407DD2 koji je uslovni i koji kada se izvrsi vodi tacno na adresu 00407DEB.

Analogno ovome postoje i CALL reference koje imaju prakticno isti nacin zapisivanja i tumacenja.

Za reversere je bitno da provere sve skokove i referencne CALLove koji se nalaze veoma blizu stringa o tacnom ili netacnom unetom seriskom broju. Pogotovo je bitno ako veci broj CALL ili JMP komandi vodi ka istom mestu. Ovo vec samo po sebi govori da smo na pravom mestu i da bas tu treba reversovati program. Ovo nemora uvek biti ispunjeno ali se kod vecine programa ova pravilnost uocava. Reference su veoma vazne za reversing stoga na povezanost delova koda treba posebno obratiti paznju. Da bi ste sebi olaksali reversovanje predlazem da umesto W32Dasma koristite Olly koji radi slikovito grupisanje po referencama.

Pogovor:

Dosli ste do samog kraja ove knige, cestitam vam na strpljenju i volji da je procitate u celosti. Iako se ova knjiga pojavila posle moje prve knjige o RCEu *The Art Of Cracking*, ova druga knjiga ustvari predstavlja uvod u sve ono sto bi svaki cracker pocetnik trebalo da zna pre nego sto pocne da reversuje svoje prve mete. Ovo moze ali i nemora biti uvek prvo i obavezno stivo za sve pocetnike, ali ce njegovo citanje sigurno olaksati razumevanje osnovnih reverserskih problema. Posle citanja ove knjige preporucujem citanje knjige *The Art Of Cracking*. Ona ce vam dati kompletan uvid u resavanje RCE problema.

NASTAVICE SE...