HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Race condition propusti

CCERT-PUBDOC-2004-10-92

CARNet CERT u suradnji s LS&S

Sigurnosni problemi u računalnim programima i operativnim sustavima područje je na kojem CARNet CERT kontinuirano radi.
Rezultat toga rada ovaj je dokument koji je nastao suradnjom CARNet CERT-a i LS&S-a, a za koji se nadamo se da će Vam koristiti u poboljšanju sigurnosti Vašeg sustava.

CARNet CERT, www.cert.hr - nacionalno središte za **sigurnost računalnih mreža** i sustava.

LS&S, **www.lss.hr**- laboratorij za sustave i signale pri Zavodu za elektroničke sustave i obradbu informacija Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

Sadržaj

1. Uvod

Razvoj računalnih programa od programera sve više zahtjeva da osim same funkcionalnosti vode računa i o sigurnosti, odnosno pouzdanosti programskog koda. Računalna sigurnost vrlo je kompleksno i široko područje, i ponekad je vrlo teško razviti programski kod koji će biti otporan na sve moguće ranjivosti koje potencijalnom napadaču mogu omogućiti pristup sustavu.
Višekorisnički i višezadaćni sustavi programerima predstavljaju dodatni izazov zato što u obzir treba uzeti i sve moguće situacije u kojima resursi koje program koristi mogu biti dostupni i drugim procesima koji se istovremeno izvršavaju na sustavu. Problem do kojeg dolazi kada dva procesa nesinkronizirano, ali istovremeno pristupaju nekom resursu (memoriji, datoteci, procesorskom vremenu, itd.) naziva se *race condition* propust. *Race condition* propusti iz perspektive računalne sigurnosti mogu se iskoristiti za onesposobljavanje sustava, eskaliranje privilegija ili neku drugu neovlaštenu aktivnost. *Race condition* propusti poznati su stručnjacima za računalnu sigurnost već oko 20 godina (otprilike isto kao i propusti preljeva spremnika), no i dalje se mogu pronaći u velikom broju računalnih programa. Propusti ovog tipa otkriveni su u brojnim dobro poznatim programima što jasno ukazuje na raširenost problema. Nedavno otkrivene ranjivosti uključuju *race condition* propuste u programima kao što su `Tripwire`, `Sendmail`, `Wu-ftpd`, `screen`, itd. Svi ti propusti kategorizirani su kao propusti visokog sigurnosnog rizika, te su omogućavali eskaliranje korisničkih privilegija. Treba napomenuti da r*ace condition* propusti nisu ograničeni samo na korisničke programe. Postoje slučajevi gdje su isti propusti otkriveni i u samim jezgrama operacijskih sustava kao što su npr. Linux, Windows, FreeBSD, itd.
U dokumentu su opisani *race condition* propusti koji su karakteristični za Unix platforme. U osnovi postoje dvije vrste *race condition* propusta koji se odnose na datotečni sustav (*engl. file System*) i na rukovatelje signalima (*engl. signal handlers*). Oba dva tipa propusta podjednako su opasni, no *race condition* propuste uzrokovane signalima puno je teže spriječiti. Ponekad je ovakve propuste moguće iskoristiti i preko TCP/IP mrežnih sjednica, što predstavlja dodatni sigurnosni rizik.

2. Vrste propusta i njihovi uzroci

Race condition propusti uglavnom se događaju zato što korisnik ima mogućnost na neki način prekinuti ili preusmjeriti izvršavanje programa dok on obavlja neku kritičnu operaciju. Višezadačni (*engl. multithread)* računalni sustavi (u koje se danas ubrajaju svi popularni operacijski sustavi) imaju mogućnost izvršavanja više procesa istovremeno, odnosno paralelno. Stvarno paralelno izvršavanje procesa odnosno instrukcija na CISC (*engl. **C**ommon **I**nstruction **S**et **C**omputing*) arhitekturi (kao što je x86) nije moguće, pa operativni sustav simulira paralelizam tako što svakom procesu dodijeli određeni dio procesorskog vremena. Ovu zadaću obavlja dio jezgre operativnog sustava koji se naziva *scheduler*. Ukoliko neki aktivni proces sustava pristupa nekom resursu, a nakon isteka procesorskog vremena koje je predviđeno za taj proces tu mogućnost istovremeno ima i neki drugi aktivni proces, rezultat može biti *race condition* propust. Prije spomenuti *race condition* propusti na bazi datotečnog sustava događaju se uglavnom zato što neovlašteni korisnik može pogoditi ime datoteke koju će neki privilegirani proces kreirati prije nego što ju on stigne kreirati, pa tako neovlašteni korisnik može preduhitriti privilegirani proces. Ovakva situacija može rezultirati prepisivanjem neke važne sistemske datoteke sadržajem koji kontrolira neovlašteni korisnik.
Race condition propusti na bazi sistemskih signala mogu se uzrokovati tako što neovlašteni korisnik privilegiranom procesu pošalje određeni signal u trenutku dok on obavlja određenu kritičnu operaciju koja ne smije biti prekinuta. Za *race condition* propuste vrlo je bitno vrijeme, odnosno stanje privilegiranog procesa u trenutku u kojem neovlašteni korisnik utječe na njegovo izvođenje.

3. Race condition propusti na razini datotečnog sustava

Race condition propusti koji se temelje na datotečnom sustavu odnosno na pristupu datotekama uglavnom su rezultat nemara ili neznanja programera koji razvija određeno programsko rješenje. Svaki operativni sustav posjeduje određene API pozive koji služe za rad sa datotekama. API pozivi koji su uzrok *race condition* propusta su uglavnom `open()`, `mktemp()`, `tmpnam()` i njihovi derivati. Za API funkcije koje služe za rad sa datotekama važno je napomenuti da postoje *atomic* i *non-atomic*

funkcije. *Atomic* funkcije su one koje se izvršavaju u cijelosti, odnosno njihovo izvršavanje nije moguće prekinuti. *Non-atomic* funkcije su one koje je moguće prekinuti i one su glavni uzrok *race condition* sigurnosnih propusta. Primjer takve funkcije je funkcija `mktemp()`.
Race condition propusti često su uzrokovani zato što programer nije implementirao provjeru da li neka datoteka već postoji prije nego ju otvara ili zato što se ta provjera vrši na pogrešan način, pa neovlašteni korisnik može kreirati datoteku nakon provjere. *Race condition* propusti se uglavnom odnose na one datoteke koje su kreirane u dijeljenim direktorijima, kao što je npr. `/tmp`, kojima pristup imaju svi korisnici i procesi sustava.

3.1. Korištenje mktemp() API poziva

API poziv `mktemp()` omogućuje generiranje jedinstvenog imena neke privremene datoteke najčešće u `/tmp` direktoriju. Ta funkcija interno provjerava da li datoteka čije ime je generirano već postoji, a povratna vrijednost funkcije je pokazivač na znakovni niz imena datoteke. Nakon toga datoteku treba otvoriti sa pozivom `open()`. Iako funkcija `mktemp()` provjerava da li generirano ime datoteke već postoji, neovlašteni korisnik može kreirati tu datoteku nakon provjere, a prije nego što je ona otvorena funkcijom `open()`, što uzrokuje *race condition* propust. U nastavku je priložen program koji sadrži upravo spomenuti propust.

Race1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main (int argc, char **argv)
{
        char datoteka[256], unos[256], gen[256]="/tmp/raceconditionXXXXXX", *ime;
        int fd;

        ime = mktemp (gen); // ranjiva funkcija

        if ((fd = open (ime, O_CREAT|O_WRONLY,00755)) == -1)
        {
                perror ("open:");
                exit (-1);
        }

        do {
                fgets (unos,sizeof(unos),stdin);
                write (fd, unos, strlen(unos));
        } while (strncmp (unos,"izlaz",5) != 0);

        close (fd);
}
```

Pretpostavimo da priloženi program mogu pokretati svi korisnici na sustavu i da se izvršava pod ovlastima root korisničkog računa (`suid root`). Priloženi program funkcijom `mktemp()` generira jedinstveno ime datoteke, te ju kasnije otvara funkcijom `open()`. Ime generirane datoteke je `/tmp/raceconditionXXXXXX`, s time da funkcija `mktemp()` znakove 'X' mijenja i na njihova mjesta ubacuje alfanumeričke znakove koji osiguravaju jedinstvenost datoteke. Nakon zamjene znakova, ime datoteke izgleda kao `/tmp/raceconditionPgAs51`. Iako ime datoteke (zadnjih šest znakova) izgleda teško za predvidjeti, neovlašteni korisnik ima neograničeni broj pokušaja, pa se sigurnost provjere smanjuje sa svakim novim pokušajem. Neovlašteni korisnik može kreirati simboličke veze (*engl. symbolic links*) koje pokazuju na datoteku `/etc/passwd`, što mu omogućava prepisivanje te datoteke. Ukoliko neovlašteni korisnik uspije predvidjeti ime datoteke i kreira simboličku vezu tog imena točno nakon `mktemp()` funkcije, a prije poziva `open()` funkcije, datoteka `/etc/passwd` je prepisana sadržajem koji kontrolira neovlašteni korisnik. Neovlašteni korisnik također može dodatno usporiti izvršavanje programa, primjenom napada uskraćivanjem računalnih resursa (*engl. **D**enial **o**f **S**ervice*), tako da poveća vjerojatnost kreiranja simboličke veze točno između pozivanja `mktemp()` i `open()` funkcija. Nakon prevođenja programa i sam prevoditelj (`gcc`) javlja upozorenje da se koristi `mktemp()` funkcija koja je opasna.

```
[root@laptop RACECONDITION]# gcc race3.c
race3.c: In function `main':
race3.c:16: warning: assignment makes pointer from integer without a cast
/tmp/ccDlxc4J.o: In function `main':
/tmp/ccDlxc4J.o(.text+0x39): the use of `mktemp' is dangerous, better use
`mkstemp'
```

Kao što i `gcc` prevoditelj upozorava, umjesto funkcije `mktemp()` preporučljivo je koristiti *atomic* funkciju `mkstemp()` koja umjesto imena datoteke vraća otvorenu datoteku odnosno njezin opisnik datoteke (engl. *file descriptor*), što uklanja mogućnost provođenja *race condition* napada.

3.2. Generiranje privremene datoteke vlastitim postupkom

Programeri često koriste vlastite postupke za generiranje slučajnog imena datoteke. Ovakva metoda je posebno opasna, zato jer se imena datoteka najčešće generiraju na temelju vrijednosti koje je lako pogoditi. Uglavnom se radi o uobičajenim imenima datoteka kojima se dodaje PID oznaka trenutnog procesa, pa ime datoteke izgleda kao `/tmp/racecondition.25012`. U nastavku je priložen program kao primjer upravo spomenutog propusta.

Race2.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main (int argc, char **argv)
{
        char datoteka[256], unos[256];
        int fd,pid;

        pid = getpid();
        sprintf (datoteka,"/tmp/racecondition.%d",pid);

        if ((fd = open (datoteka, O_CREAT|O_WRONLY,00755)) == -1)
        {
                perror ("open:");
                exit (-1);
        }

        do {
                fgets (unos,sizeof(unos),stdin);
                write (fd, unos, strlen(unos));
        } while (strncmp (unos,"izlaz",5) != 0);

        close (fd);
}
```

Kao što je vidljivo iz primjera, datoteka se kreira tako da se kao ekstenzija dodaje PID procesa. Neovlašteni korisnik može vrlo lako predvidjeti PID vrijednosti za nove procese koji će se pokrenuti, pa može kreirati simboličke veze na `/etc/passwd` datoteku što mu opet omogućuje prepisivanje te datoteke proizvoljnim sadržajem. Ovakvi *race condition* propusti mnogo su opasniji i od korištenja `mktemp()` funkcije zato što u imenu datoteke nema nikakve slučajne vrijednosti, s obzirom da se PID vrijednost uvećava linearno za svaki novi proces. Ovakav i slične druge postupke generiranja "slučajnog" imena datoteke treba izbjegavati, no ukoliko se i koristi, pri pozivu `open()` funkcije potrebno je dodati i O_EXCL zastavicu koja označava da ukoliko datoteka koja se otvara već postoji, funkcija `open()` prekida izvršavanje i vraća negativnu vrijednost (grešku). Za zaštitu od *race condition* propusta, može se koristiti i `lstat()` sistemski poziv koji može utvrditi da li je neka datoteka simbolička veza. Ovakvi propusti često se mogu pronaći u skriptama ljuske (*engl. shell script*). U nastavku je priložena skripta koja sadrži spomenuti propust.

```
#!/bin/sh
ls -al /tmp
echo "CERT&LSS Race condition primjer" > /tmp/racecondition.$$
ls -al /tmp
```

Takve ranjive skripte se na žalost i danas mogu vrlo često pronaći u nekim popularnim distribucijama Linux operacijskog sustava.

4. Race condition propusti vezani uz signale

Signali su poruke koje se šalju procesima u određenim situacijama. Kad proces primi signal, prekida se izvršavanje programa i "obrađuje" se signal, a ukoliko program ima postavljene rukovatelje signala (funkcije koje se izvršavaju po primitku signala) oni se u tom trenutku izvršavaju. Svaki signal ima jedinstveni broj koji ga predstavlja. Neki signali mogu biti ignorirani ili uhvaćeni (pomoću rukovatelja signala), dok neki (npr. KILL i STOP) bezuvjetno prekidaju izvođenje programa. Signale procesima može slati jezgra operativnog sustava (npr. SEGV) i sam korisnik programima `kill` i `killall`. Signali se šalju pomoću API poziva `kill()`. Primjer za slanje signala je kada korisnik pritisne kombinaciju tipki CTRL+C za prekid programa, nakon čega se procesu šalje prekidni (INT) signal i prekida se izvršavanje programa.
Važno je napomenuti da se nakon završetka izvršavanja funkcije za rukovanje signalima program nastavlja dalje normalno izvršavati. Za hvatanje signala potrebno je definirati funkciju koja će po primitku signala obraditi signal. Funkcija koja hvata određeni signal definira se pomoću API poziva `signal()`. Kada je procesu poslan signal (npr. INT za prekid) on se obrađuje i za vrijeme obrade signala ako proces opet primi isti signal, novi signal će biti zanemaren, no ukoliko proces primi neki drugi signal on će također biti obrađen. *Race condition* propusti vezani uz signale događaju se kada program primi više signala (barem dva), a za sve njih ima samo jednu funkciju, ili kad primitak signala u određenom trenutku može utjecati na daljnje izvršavanje programa.

4.1. Propusti izazvani primanjem dva različita signala

Ukoliko program za dva različita signala ima definiranu istu funkciju, postoji mogućnost da je program ranjiv na *race condition* napad. Do problema dolazi ukoliko proces primi određeni signal i počne ga obrađivati odgovarajućom funkcijom, a onda primi drugi signal koji prekida izvršavanje rukovateljske funkcije u nezgodnom trenutku, te opet poziva istu funkciju. U nastavku je priložen primjer takvog programa.

Race3.c

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

void handler (int a)
{
        static int flag=0;
        printf ("flag:%d - sig: %d\n",flag,a);

        if (flag == 1) {
                printf ("Kolizija signala!!!\n");
                exit(0);
        }

        flag = 1;
         if (flag == 1)
        printf ("HANDLER JE POKRENUT!!!\n");
        sleep(1);
        flag = 0;
}

main (int argc, char **argv)
{
        signal (SIGINT, handler); // 2
        signal (SIGQUIT, handler); // 3

        printf ("CERT&LSS Signal race condition primjer\n");
        getchar();
}
```

U programu je za rukovanje signalima INT i QUIT postavljena rukovateljska funkcija `handler()` koja obrađuje te signale ukoliko ih proces primi. U normalnim okolnostima, žuto označen `if()` uvijet nikad nije ispunjen, zato jer je varijabla `flag` postavljena na vrijednost 0. Varijabla `flag` se unutar `handler ()` funkcije postavlja na vrijednost 1, te se ispisuje tekst "HANDLER JE POKRENUT!!!" i vrijednost `flag` zastavice se opet postavlja na vrijednost 0. Ukoliko proces primi INT signal, počinje

se izvršavati funkcija `handler()` i varijabla `flag` se postavlja na vrijednost 1. Ako u tom trenutku proces primi i signal QUIT, ponovo se poziva funkcija `handler()`, a s obzirom da je `flag` zastavica postavljena na vrijednost 1, izvršava se žuto označen `if()` uvjet koji u normalnim okolnostima nije dostupan. U nastavku je prikazano korištenje programa `Race3.c`.

```
[root@laptop RACECONDITION]# ./race4
CERT&LSS Signal race condition primjer
flag:0 - sig: 3
HANDLER JE POKRENUT!!!
flag:0 - sig: 2
HANDLER JE POKRENUT!!!
flag:0 - sig: 2
flag:1 - sig: 3
Kolizija signala!!!
[root@laptop RACECONDITION]#
```

Signali koje je program `race4` primio poslani su sljedećim naredbama:

```
[root@laptop RACECONDITION]# kill -3 `pidof race4`
[root@laptop RACECONDITION]# kill -2 `pidof race4`
[root@laptop RACECONDITION]# kill -2 `pidof race4` ; kill -3 `pidof race4`
```

Proces `race4` prvo prima signal QUIT (3), a zatim nekoliko sekundi kasnije INT (2) signal. Nakon primitka signala pokreću se rukovatelji signala i za svaki signal ispisuje se stanje `flag` varijable i broj signala, te tekst "HANDLER JE POKRENUT!!!". Na kraju proces u kratkom vremenu prima i INT i QUIT signal što uzrokuje *race condition* propust odnosno koliziju signala, jer se izvršava prvi `if()` uvjet koji je u normalnim okolnostima nedostupan.

Ovo je bio samo jednostavan primjer kako rukovatelji signalima mogu utjecati na izvršavanje programa. Programi koji se koriste svakodnevno puno su kompliciraniji i njihovi rukovatelji signala moraju biti sigurni. Kod pisanja rukovatelja signala potrebno je paziti koje API funkcije se koriste, jer samo se neke API funkcije mogu pozivati unutar rukovatelja signala bez da potencijalno utječu na sigurnost programa. U nastavku su nabrojene funkcije koje se mogu pozivati iz rukovatelja signala bez da predstavljaju potencijalni sigurnosni propust za sustav. Njihovo izvršavanje se ne može prekinuti ponovnim pozivanjem rukovateljske funkcije (kao što je prikazano u prethodnom primjeru). Sve funkcije koje nisu navedene predstavljaju opasnost za sigurnost izvršavanja programa ukoliko ih se poziva iz rukovatelja signala.

```
_exit(2), access(2), alarm(3), cfgetispeed(3), cfgetospeed(3),
cfsetispeed(3), cfsetospeed(3), chdir(2), chmod(2), chown(2),
close(2), creat(2), dup(2), dup2(2), execle(2), execve(2),
fcntl(2), fork(2), fpathconf(2), fstat(2), fsync(2), getegid(2),
geteuid(2), getgid(2), getgroups(2), getpgrp(2), getpid(2),
getppid(2), getuid(2), kill(2), link(2), lseek(2), mkdir(2),
mkfifo(2), open(2), pathconf(2), pause(2), pipe(2), raise(3),
read(2), rename(2), rmdir(2), setgid(2), setpgid(2), setsid(2),
setuid(2), sigaction(2), sigaddset(3), sigdelset(3),
sigemptyset(3), sigfillset(3), sigismember(3), signal(3),
sigpending(2), sigprocmask(2), sigsuspend(2), sleep(3), stat(2),
sysconf(3), tcdrain(3), tcflow(3), tcflush(3), tcgetattr(3),
tcgetpgrp(3), tcsendbreak(3), tcsetattr(3), tcsetpgrp(3), time(3),
times(3), umask(2), uname(3), unlink(2), utime(3), wait(2),
waitpid(2), write(2). sigpause(3), sigset(3).
```

Umjesto pisanja sigurnih rukovatelja signala tako da se koriste samo sigurne funkcije, lakše je prilikom pozivanja rukovatelja signala koristiti globalnu varijablu koja označava stanje u kojem se nalazi rukovatelj signala te tako dopušta, ili ne dopušta pozivanje određenog dijela programskog koda.

4.2. Primanje signala preko TCP/IP sjednice

Ponekad program može imati samo jedan rukovatelj signala, no primitak tog signala u pogrešnom trenutku odnosno kada program obavlja neku osjetljivu operaciju može izazvati sigurnosni propust. Mrežni programi često imaju postavljen rukovatelj signala za signal URG. Signal URG može biti poslan preko mreže tako da se funkciji `send()` koja se koristi za slanje podataka preko TCP/IP sjednice doda MSG_OOB (*engl. Out-Of-Band*) zastavica. TCP paket koji se šalje preko mreže u tom slučaju ima u zaglavlju postavljenu zastavicu URG. Rukovatelji signala za URG signal neovlaštenom korisniku omogućavaju prekid odnosno preusmjeravanje izvršavanja poslužiteljskog programa preko mreže bez

posebne potrebe za autentikacijom, što samo po sebi predstavlja određeni sigurnosni problem. U nastavku je priložen primjer poslužiteljskog programa koji je ranjiv na *race condition* propust uzrokovan URG signalom.

Race4.c

```
#include <stdio.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <signal.h>
#include <fcntl.h>

char *unos;
int fdcli;

void handle (int sig)
{
        int n;
         n = recv (fdcli, unos, 1024, MSG_OOB);
         printf ("Signal handler za MSG_OOB!!!: %s\n",unos);
         free(unos);
}

main (int argc, char **argv)
{
  struct sockaddr_in serv, cli;
  int fdserv;
  int i=1,n,x;

  signal (SIGURG, handle);

  fdserv = socket (AF_INET, SOCK_STREAM, 0);
  setsockopt (fdserv, SOL_SOCKET, SO_REUSEADDR, (void*)&i, sizeof(i));
  serv.sin_family = AF_INET;
  serv.sin_port = htons (31337);
  serv.sin_addr.s_addr = INADDR_ANY;
  bzero (serv.sin_zero, 8);

  if ((bind(fdserv, (struct sockaddr*)&serv, sizeof (struct sockaddr))) == -1)
  {
       perror ("bind:");
       exit(-1);
  }
  listen (fdserv, 5);
  n = sizeof(struct sockaddr);
  while ((fdcli = accept (fdserv, (struct sockaddr*)&cli, &n)) > 0) {
         unos = (char*)malloc(1024);
         fcntl(fdcli, F_SETOWN, getpid());
         n = recv (fdcli, unos, 1024, 0);
         printf ("KLIJENT: %s\n",unos);
        sleep(2);
         free (unos);
         exit(0);
  }
  close (fdserv);
  close (fdcli);
}
```

Priloženi poslužiteljski program nakon pokretanja sluša na TCP portu 31337. Nakon spajanja klijenta, program od njega očekuje podatke koji su zatim ispisani na standardni izlaz. Program također ima postavljen rukovatelj signala `handle()` za signal URG. Za pohranu podataka primljenih od strane klijenta program koristi znakovni niz dinamički alocirane memorije na hrpi (*engl. heap*). Nakon ispisa znakovnog niza koji je dobiven od klijenta, alocirana memorija se oslobađa funkcijom `free()`. Ukoliko poslužiteljski program primi URG signal, rukovatelj signala ispisuje sadržaj znakovnog niza dobivenog od klijenta i opet oslobađa memoriju funkcijom `free()`.

Race condition sigurnosni propust ovdje je moguće izazvati tako da neovlašteni korisnik pošalje poslužitelju podatke preko TCP sjednice i onda nakon kratkog vremena preko te iste sjednice pošalje podatke sa postavljenom MSG_OOB zastavicom. Prvi paket će biti normalno obrađen od strane

programa, a drugi paket će pozvati rukovatelj signala za URG signal. S obzirom da rukovatelj URG signala oslobađa zauzetu memoriju za podatke primljene od strane klijenta, a ta ista memorija je bez provjere oslobođena i u samom programu, ova situacija rezultira klasičnim *double-free* sigurnosnim propustom na korisničkom unosu, što omogućuje izvršavanje nekog *shellcode* programskog koda. U nastavku je priložen program koji iskorištava spomenutu ranjivost, pri čemu prekida izvođenje poslužiteljskog programa.

Racexsploit.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/time.h>

main (int argc, char **argv)
{
        int sock,n,y;
        struct sockaddr_in sin;

        sock = socket (AF_INET, SOCK_STREAM, 0);
        sin.sin_family = AF_INET;
        sin.sin_port = htons (31337);
        sin.sin_addr.s_addr = inet_addr (argv[1]);
        bzero (sin.sin_zero,8);

        connect (sock, (struct sockaddr*)&sin, sizeof(struct sockaddr));

        send (sock, "[ERT&LSS",8,0);
        send (sock, "C",1, MSG_OOB);
        sleep(2);
        close (sock);
}
```

Žuto označena linija predstavlja postavljanje MSG_OOB zastavice na paket koji će biti poslan. Kao argument programu potrebno je putem naredbenog retka proslijediti IP adresu na kojoj se nalazi ranjivi poslužitelj. U nastavku je prikazan `ltrace` ispis funkcija koje se pozivaju iz ranjivog poslužiteljskog programa kada se na njega spojimo programom `racexploit.c`.

```
recv(4, 0x08049c38, 1024, 0, 0x4200aef8)          = 8
sleep(2 <unfinished ...>
--- SIGURG (Urgent I/O condition) ---
recv(4, 0x08049c38, 1024, 1, 6)                   = 1
printf("Signal handler za MSG_OOB!!!: %s...,  "CERT&LSS"Signal handler za
MSG_OOB!!!: CERT&LSS
) = 39
free(0x08049c38)                                  = <void>
breakpointed at 0x420b4b30 (?)
printf("KLIJENT: %s\n", "CERT&LSS"KLIJENT: CERT&LSS
)                 = 18
free(0x08049c38)                                  = <void>
--- SIGSEGV (Segmentation fault) ---
```

Kao što je vidljivo iz priloženog primjera, poslužitelj dobiva URG signal i na prvoj žuto označenoj liniji rukovatelj signala oslobađa memoriju na kojoj se nalaze podaci dobiveni od strane klijenta. Proces se vraća iz obrade signala nakon čega se oslobađa već oslobođena memorijska adresa što dovodi do rušenja programa, odnosno program dobiva SEGV signal od jezgre operacijskog sustava.

5. Zaključak

Race condition propuste koji se temelje na datotečnom sustavu relativno je lako otkriti i odstraniti, dok se propusti vezani uz rukovanje signalima puno teže otkrivaju i sprječavaju. Oba tipa propusta podjednako su opasni i predstavljaju prijetnju za sam računalni sustav. Ponekad je pisanje sigurnog programskog koda koji će se izvršavati na višezadaćnim sustavima težak zadatak, no ukoliko programer slijedi određene norme i standarde sigurnog programiranja, to je svakako moguće.

6. Reference

[1] Michal Zalewski, "*Delivering Signals for Fun and Profit*", http://www.bindview.com
[2] David Wheeler, *Secure Programming for Linux and Unix HOWTO*
[3] Simson Garfinel, Gene Spafford & Alan Schwartz, *"Practical Unix & Internet Security"*
[4] Richard Stevenson, *"TCP/IP Illustrated, Volume 1"*
[5] man signal(7)
[6] man mktemp(3)