HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Napredne metode otkrivanja sigurnosnih propusta

CCERT-PUBDOC-2003-12-100

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

Sigurnost računalnog sustava uvelike ovisi o stupnju njegove transparentnosti. Ukoliko je sustav jednostavan i pristupačan, otkrivanje njegovih sigurnosnih propusta i nedostataka može biti vrlo jednostavan i kratak proces. Kod složenijih sustava proces otkrivanja sigurnosnih propusta mnogo je kompliciraniji i zahtijeva znatno više vremena i truda.

Osnovni cilj postupka analize sigurnosti računalnog sustava je prikupiti što više upotrebljivih informacija, a da se pri tome utroši što manje vremena i resursa. Ukoliko se radi o kompleksnom računalnom sustavu to nije uvijek moguće, pa su se iz tog razloga počeli razvijati različiti alati koji automatiziraju postupak ispitivanja sigurnosti. Takvi alati se obično nazivaju *fuzzeri* i oni predstavljaju značajan pomak u području otkrivanja sigurnosnih propusta. Njihovom upotrebom moguće je pravovremeno uočiti nove i nepoznate ranjivosti, koje potencijalni neovlašteni korisnici mogu iskoristiti za nedopuštene, odnosno maliciozne radnje.

U ovom dokumentu opisani su postupci otkrivanja sigurnosnih propusta uz pomoć jednostavnih, ali učinkovitih alata, koji automatiziraju proces analize sigurnosti programa i servisa na Linux operacijskim sustavima. Iako se alati fokusiraju na otkrivanje danas najopasnijih sigurnosnih propusta kao što su preljevi spremnika i *format string* propusti, rezultat analize može biti i otkrivanje ranjivosti na napad uskraćivanjem računalnih resursa (*engl. Denial of Service*). *Fuzzer* alati omogućavaju analizu sigurnosti računalnog programa bez izravne intervencije osobe ili tima koji su uključeni u proces, što svakako predstavlja određenu prednost.

Tehnike i metode koje koriste *fuzzer alati* uglavnom su vrlo jednostavne. *Fuzzer* alat razumije protokol ili način na koji funkcionira određena aplikacija, te na temelju toga aplikaciji prosljeđuje one vrijednosti koje bi mogle uzrokovati nepredvidljivo ponašanje koje upućuje na potencijalni sigurnosni problem. Na Internetu je danas moguće pronaći besplatne i komercijalne *fuzzer* alate, čija kvaliteta varira od proizvoda do proizvoda. Trenutno najpoznatiji mrežni *fuzzer* je alat pod nazivom `SPIKE`, koji se odlikuje mnoštvom opcija i funkcionalnosti, te predstavlja trenutno najbolji besplatni *fuzzer* alat. Poznat je i alat pod nazivom `CHAM` (*engl. **C**ommon **H**acking **A**ttack **M**ethods*), komercijalni *fuzzer* alat kojeg je razvila tvrtka eEye koja trenutno zauzima vodeće mjesto u području otkrivanja sigurnosnih propusta na Windows operacijskim sustavima. `CHAM` *fuzzer* dolazi integriran u alat `Retina` koji služi za provjeru ranjivosti (*engl. vulnerability scanning*) računalnih sustava i mreža.

Za potrebe dokumenta razvijeni su programi koji na jednostavan način prikazuju tehnike i metode na kojima se baziraju *fuzzer* alati. Također treba napomenuti da je u sklopu provedenih testiranja otkriveno nekoliko sigurnosnih propusta od kojih će neki biti pobliže objašnjeni u nastavku dokumentu.

2. Fuzzing metode i tehnike

Da bi *fuzzer* alat mogao analizirati sigurnost određenog računalnog sustava, odnosno aplikacije, on na neki način mora s njome komunicirati. Ukoliko se radi o korisničkom programu, *fuzzer* mora biti u mogućnosti predavanja unosa aplikaciji, a ukoliko se radi o mrežnoj aplikaciji odnosno Internet servisu, *fuzzer* mora poznavati i protokol na kojem se servis bazira.

Metode za otkrivanje sigurnosnih propusta koje koriste *fuzzer* alati uglavnom su vrlo jednostavne. S ciljem otkrivanja sigurnosnog propusta, *fuzzer* pokušava "napasti" aplikaciju korisničkim unosom za koji se pretpostavlja da bi mogao uzrokovati probleme koji upućuju na potencijalni sigurnosni problem. Osnovna tehnika na kojoj se baziraju *fuzzer* alati je modifikacija korisničkog unosa na način da se umjesto standardnih upita i naredbi koje se predaju aplikaciji, šalju podaci koji se razlikuju od onih koje aplikacija očekuje. Najčešće se radi o predugim upitima ili o specijalnoj kombinaciji naredbi, parametara i znakova koji bi mogli uzrokovati nepredviđeno ponašanje aplikacije koja se testira.

Iako su *fuzzeri* iznimno moćan alat u rukama sigurnosnih stručnjaka, oni ipak imaju određene nedostatke. Nedostatak *fuzzera* je činjenica da se njihov rad i uspješnost zasniva na metodi pokušaja i promašaja, te na znanju i iskustvu osobe koja je izradila određeni *fuzzer* alat. Neke sigurnosne propuste koji su na prvi pogled očiti u slučaju analize izvornog koda aplikacije, teško je otkriti *fuzzing* tehnikama. Može se npr. raditi o sigurnosnom propustu koji za svoju realizaciju zahtijeva određene

akcije koje moraju biti provedene prije nego se pozove naredba, instrukcija ili parametar koji izaziva sigurnosni propust.
Fuzzer programi dijele se na lokalne i mrežne verzije. Lokalni *fuzzer* alati služe za analizu korisničkih programa koji se nalaze na lokalnom računalu. Uglavnom se radi o korisničkim programima koji nakon pokretanja korisniku daju veću razinu ovlasti od onih koje već ima. Mrežni *fuzzer* alati služe za analizu sigurnosti mrežnih aplikacija odnosno servisa (npr. FTP, HTTP, SMTP, itd.). Da bi mrežni *fuzzer* alati mogli uspješno provjeriti "otpornost" aplikacije na različite napade oni moraju dobro poznavati protokol na kojem se aplikacija zasniva, pa o tome na kraju i ovisi uspješnost samog *fuzzer* alata. Prilikom provedbe *fuzzing* testa na aplikaciji, poželjno je nadgledati ponašanje aplikacije u odnosu na količinu memorije koju alocira, operacije na datotečnom sustavu, prekide i signale koje aplikacija uzrokuje i dobiva, te eventualno nasilno prekidanje rada aplikacije (engl. *crash*) koje može upućivati na potencijalni sigurnosni problem. Ponašanje aplikacije najbolje je nadgledati nekim *real time* alatom za otkrivanje grešaka (engl. *debuger*), pomoću kojeg je moguće analizirati stanje aplikacije u svakom trenutku. Za Linux operacijski sustav preporuča se `gdb`, a za Windows operacijske sustave poznat je alat pod nazivom `OllyDbg` (http://home.t-online.de/home/Ollydbg).

3. Lokalni fuzzer alati

Kao što je već objašnjeno, lokalni *fuzzer* alati služe za analizu sigurnosti korisničkih programa koji mogu biti iskorišteni za eskaliranje privilegija potencijalnog neovlaštenog korisnika. Uglavnom se radi o tzv. `suid` (engl. *set user ID*) aplikacijama koje korisniku privremeno pružaju višu razinu ovlasti na sustavu. Lokalni *fuzzer* alat na Linux operacijskom sustavu može testirati aplikaciju npr. s obzirom na argumente koji se aplikaciji predaju putem naredbenog retka i na podatke koje aplikacija preuzima iz varijabli okruženja (engl. *environment*). Takvi *fuzzer* alati su jednostavni, a uglavnom na brz i efikasan način dokazuju postojanje sigurnosnog propusta.
Varijable okruženja često znaju biti predmet sigurnosnih propusta, jer programeri ne pridodaju dovoljnu pažnju operacijama koje se obavljaju na podacima dobivenim iz njih. Pri analizi sigurnosti aplikacija često se može naići na dijelove koda u kojima se podaci preuzimaju iz varijabli okruženja, a zatim se nad tim podacima obavljaju operacije za koje se zna da mogu uzrokovati sigurnosni propust. Takve funkcije mogu biti npr. `strcpy()`, `sprintf()`, `fprintf()` itd. Program, odnosno *fuzzer* alat koji će testirati ranjivosti vezane uz varijable okruženja jednostavno je napisati, a rezultati mogu biti i više nego zadovoljavajući. U nastavku je priložen jednostavan Linux *fuzzer* koji omogućuje testiranje sigurnosti aplikacije s obzirom na podatke koje ona preuzima iz varijabli okruženja. *Fuzzer* za svaku varijablu okruženja postavlja niz znakova dužine 100000 okteta, što kod nekih programa uzrokuje preljeve spremnika.

Myfuzz.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

char *ptr;

char *getenv(const char *name)
{

        if (strncmp(name,"HOME",4) == 0)
                return "/home/ljuranic";
        if (strcmp(name,"LD_PRELOAD")==0)
                return NULL;
        if (strcmp(name,"LOGNAME")==0)
                return NULL;

        return ptr;
}

uid_t getuid (void)
{
        return 500;
}
```

```
uid_t geteuid (void)
{
       return 500;
}

uid_t getgid (void)
{
       return 500;
}

uid_t getegid (void)
{
       return 500;
}

void _init ()
{

       ptr = (char*) malloc (100000);
       memset (ptr,'A', 100000);
}
```

Priloženi *fuzzer* alat realiziran je u obliku biblioteke koja preučitava (engl. *preload*) određene GLIBC pozive i vraća određene vrijednosti za svakog od njih. Samo testiranje ranjivosti odnosi se na preučitavanje poziva `getenv()` koji služi za dolazak do sadržaja varijable okruženja i postavljanja povratne vrijednosti tog poziva na niz znakova dužina 100000 okteta. Linux biblioteke prevode se drugačije nego obični programi kao što je i prikazano u nastavku.

```
# gcc myfuzz.c -c –fPIC
# ld  -o myfuzz.so myfuzz.o -ldl -shared –lc
```

Nakon prevođenja dobivamo biblioteku `myfuzz.so`. Linux operacijski sustav pri pokretanju programa provjerava varijablu okruženja LD_PRELOAD. Uglavnom je ta varijabla prazna, no ona može sadržavati putanju do biblioteke koja će se naknadno učitati u memoriju aplikacije koja se pokreće. U ovom slučaju funkcije koje će se učitati zamjenjuju GLIBC pozive vlastitim kodom, pa će aplikacija umjesto originalnih poziva koristiti pozive od *fuzzer* programa i na taj način omogućiti analizu sigurnosti. Postavljanje LD_PRELOAD varijable prikazano je u nastavku.

```
# export LD_PRELOAD=/root/PROJEKTI/myfuzz.so
```

Za analiziranje sigurnosti korisničkih programa važno je napomenuti da se testovi moraju provoditi pod ovlastima `root` korisničkog računa, jer običan korisnik nema ovlasti preučitavanja biblioteka za `suid` aplikacije.

Ovom jednostavnom bibliotekom otkrivena su 22 preljeva spremnika u standardnim korisničkim programima na operacijskom sustavu Linux `Red Hat 7.3` u periodu od 10 minuta. Programi u kojima su otkrivene ranjivosti pomoću `myfuzz.so` biblioteke su: `gtk-demo`, `4rdf`, `4xupdate`, `Mail`, `apacheconf`, `artsplay`, `artsshell`, `authconfig-gtk`, `consolehelper`, `dateconfig`, `ddd`, `dia`, `dig`, `disol`, `ethereal`, `fax2tiff`, `fmtest`, `gconftool-1`, `getent`, `gftp`, `gtk-gftp` i `gnome-man2html2`.

U nastavku je pobliže prikazano otkrivanje preljeva spremnika u popularnom programu za nadgledanje mrežnog prometa pod nazivom `Ethereal`.

```
[root@laptop MYFUZZER]# ethereal –v
ethereal 0.9.3, with GTK+ 1.2.10, with GLib 1.2.10, with libpcap 0.6, with
libz 1.1.3, with UCD SNMP 4.2.4
[root@laptop MYFUZZER]# export LD_PRELOAD="./myfuzz.so"
[root@laptop MYFUZZER]# ethereal
Segmentation fault (core dumped)
[root@laptop MYFUZZER]# export LD_PRELOAD=""
[root@laptop MYFUZZER]# gdb -c=core.1737
GNU gdb Red Hat Linux (5.1.90CVS-5)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux".
Core was generated by `AAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0  0x420824ac in ?? ()
```

```
(gdb) bt
#0  0x420824ac in ?? ()
#1  0x42051ca8 in ?? ()
#2  0x4206dbf3 in ?? ()
#3  0x4205a257 in ?? ()
#4  0x4005f5b7 in ?? ()
#5  0x41414141 in ?? ()
Cannot access memory at address 0x41414141
(gdb) frame 5
#5  0x41414141 in ?? ()
$eip(gdb) print $eip
$1 = (void *) 0x41414141
(gdb)
```

Žuto označene linije označavaju učitavanje `myfuzz.so` biblioteke, pokretanje `Ethereal` aplikacije i njezino rušenje. Rušenje aplikacije sa porukom "*Segmentation fault*" označava pogrešku pri upravljanju sa memorijom, što upućuje na sigurnosni propust.
Nakon otvaranja `core.1737` datoteke (pomoću `gdb` programa) koja predstavlja sadržaj memorije `Ethereal` procesa prilikom rušenja, vidljivo je da je stog prepisan znakovima 'A' (heksadecimalno 0x41 na zadnjoj žuto označenoj liniji), što upućuje na klasični propust preljeva spremnika.

4. Mrežni fuzzer alati

Kao što je već navedeno, mrežni *fuzzer* alati omogućuju analizu sigurnosti mrežnih aplikacija, odnosno servisa. Veliki broj sigurnosnih propusta visokog sigurnosnog rizika u brojnim popularnim mrežnim aplikacijama može biti otkriven jednostavnim *fuzzing* testom. Isto tako, velik broj ranjivosti koje su brojni poznati mrežni crvi koristili za svoje širenje, mogle su biti locirane i eliminirane prije nego što su ovi maliciozni programi kompromitirali velik broj poslužitelja na Internetu i time uzrokovali ogromne novčane gubitke.
Za ispitivanje sigurnosti određene mrežne aplikacije potrebno je *fuzzer* alat prilagoditi protokolu na kojem se ona zasniva. Bolji *fuzzer* alati kao što je SPIKE imaju mogućnost modeliranja, odnosno definiranja specifikacija mrežnih protokola, što omogućuje analizu velikog broja mrežnih aplikacija različitih namjena. Metode i tehnike koje koriste mrežni *fuzzer* alati slične su onima koje koriste lokalni *fuzzeri*, samo što je u tom slučaju potrebno zadovoljiti i pravila korištenog mrežnog protokola. U svrhu što kvalitetnije analize, potrebno je napraviti dobru specifikaciju protokola i u okvire testa uključiti što više mogućnosti koje pruža ciljna aplikacija.
Primjer mrežnog *fuzzinga* biti će objašnjen na SMTP (engl. ***S****imple* ***M****ail* ***T****ransfer* ***P****rotocol*) poslužiteljskom softveru. SMTP poslužiteljski softver prema RFC (engl. ***R****equest* ***F****or* ***C****omment*) definiciji protokola mora podržavati nekoliko naredbi kao što su HELO, MAIL, RCPT, DATA, NOOP, VRFY, EXPN i druge. Testiranje sigurnosti SMTP poslužiteljskog paketa podrazumijeva testiranje svih naredbi koje definira RFC specifikacija i njihovih parametara, odnosno argumenata. Samo testiranje provodi se parametrima i podacima koji u određenoj mjeri odstupaju od definiranih standarda, i vjerojatno samim time od očekivanja programera koji je radio na razvoju programa.
U nastavku je prikazano nekoliko jednostavnih testova kakve bi između ostalih trebao provoditi solidan SMTP *fuzzer* alat.

```
HELO
HELO %n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
MAIL FROM: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…
RCPT TO: AAAAAA@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@…
VRFY:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
EXPN <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<@>>>>>>>>>>>>>>>>>>>>>>>>>>>>
MAIL: %nAAA<@>AAAAA;AAA@@@@AAA%n%nAAAAAAA...AAAAAAAAA@lss.hr%n><<<<>…
DATA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
NOOP ;;;;;;AA;;;;;;;;;;;;;;;;;;;;;;;@;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Uspješnost pojedinih testova ovisi o potencijalnim ranjivostima aplikacije koja se analizira. Automatizacija ovakvih testova u konačnici rezultira izradom *fuzzer* alata. U nastavku je priložen jednostavan SMTP *fuzzer* alat napisan u PERL programskom jeziku. *Fuzzer* se spaja na SMTP servis i testira pojedine SMTP naredbe na moguće ranjivosti (preljevi spremnika, *format string* propusti, itd.).

Smtpfuzz.pl

```
#!/usr/bin/perl
#
```

```
# VERY Simple SMTP Fuzzer
# ----------------------
# With this simple, but aggressive fuzzer it is easy to discover some obvious bugs
in
# SMTP server software.
#
# Coded by Leon Juranic <ljuranic@lss.hr> // http://security.lss.hr
#

use IO::Socket;
use Time::localtime;

@var = ("HELO", "MAIL FROM", "RCPT TO", "DATA", "RSET", "NOOP",
        "VRFY", "EXPN", "VERB", "ETRN", "DSN", "AUTH", "STARTTLS", "RCPT", "MAIL",
"QUIT", "EHLO");

@fzzchr = ("\%n", "\%n@\%n.\%n", "\%x", "@", "@.@", ";", ".",
"test\@test.com;",":", ")", "(", "#{",
          "<",">","<@>","\xff","|"," ","\%s",  "/", "A", "&", "!", "_", "=" ,"`");

$SIG{PIPE} = '\&connect_smtpd';  # handle broken pipe
$SIG{ALRM}=  '\&connect_smtpd';

connect_smtpd();

print $in = <$sock>;

for ($inc = 1; $inc < 4000 ; $inc +=200)
{
   foreach $chr (@fzzchr) {
        foreach $cmd (@var)
        {

                print "CMD: $cmd:" . $chr x 4 . "...\n";;
                print $sock "$cmd\r\n";
                print $sock "$cmd" . $chr x $inc . "\r\n";
                print $sock "$cmd: " . $chr x $inc . "\r\n";
                print $sock ".\r\n";
                alarm(5);
                if(sysread ($sock, $in, 10000) == 0) {
                  connect_smtpd();
                 }
                if ($in =~/221/ || $ine =~ /421/) { connect_smtpd() };
        }

   }
}

printf "\n-> Stage 2...random (no output)\n";
sleep 3;

for ($inc = 1; $inc < 8000 ; $inc +=400)
{
        foreach $cmd (@var)
        {
                $a = mix_fzz_chr ($inc);
                 print $sock "$cmd\r\n";
                print $sock $a . "\r\n";
                 print $sock "$cmd: " . $a . "\r\n";
                 print $sock ".\r\n";
                 alarm(5);
                 if(sysread ($sock, $in, 10000) == 0) {
                   connect_smtpd();
                 }
                 if ($in =~/221/ || $ine =~ /421/) { connect_smtpd() };
        }

}


print "\nDONE - FUZZING OVER!\n";
```

```
sub mix_fzz_chr ()  # very simple "random" string generator
{
  my $num = @_;
  $tm = localtime;
  srand ($tm->sec);
  for ($x = 0 ; $x < $num ; $x++)
  {
   $index = int(rand($#fzzchr - 1));
   $ret .= $fzzchr[$index];
  }
  return $ret;
}


sub connect_smtpd ()
{

  close ($sock);
  $sock = IO::Socket::INET->new (PeerAddr => $ARGV[0],
                                 PeerPort => 25,
                                 Proto    => "tcp") || connect_smtpd();
  print ".";
  alarm 0;
}
```

Fuzzer se sastoji od dva testa. U prvom testu *fuzzer* se spaja na SMTP servis i testira SMTP naredbe na određene nizove znakova koji mogu izazvati sigurnosni propust. Provođenje drugog testa se odnosi na pseudo-slučajni odabir nizova znakova koji se šalju SMTP naredbama. Nakon svakog prekida (zatvaranja mrežne sjednice) SMTP servisa, *fuzzer* se ponovo spaja na servis, što rezultira prilično agresivnom provjerom sigurnosti.

Ovakav jednostavan *fuzzer* može otkriti mnogo zanimljivih i opasnih sigurnosnih propusta, što dokazuju mnogi sigurnosni propusti otkriveni u popularnom SMTP programskom paketu. Ovim SMTP *fuzzerom* otkriveni su sigurnosni propusti preljeva spremnika i *format string* propust u SMTP poslužitelju pod nazivom `qwik-smtpd`. *Format string* propust u međuvremenu je objavljen (od treće strane), no `qwik-smtpd` je izvrstan primjer ranjive aplikacije. U nastavku je prikazan sam proces *fuzzing* testa na `qwik-smtpd` SMTP servisu pomoću `smtpfuzz.pl` *fuzzera*.

```
# perl smtpfuzz.pl localhost
CMD: DATA:%s%s%s%s...
CMD: VRFY:%s%s%s%s...
CMD: EXPN:%s%s%s%s...
CMD: QUIT:%s%s%s%s...
.CMD: EHLO:%s%s%s%s...
CMD: HELO:////...
CMD: MAIL FROM:////...
CMD: RCPT TO:////...
CMD: DATA:////...
```

Važno je napomenuti da se `smtpfuzz.pl` *fuzzerom* mogu otkriti mnogi još neotkriveni sigurnosni propusti u SMTP poslužiteljima. U nastavku je prikazano otkrivanje *format string* propusta u `qwik-smtpd` poslužitelju pomoću `smtpfuzz.pl` *fuzzera*.

```
[root@laptop SECRES]# perl smtpfzz.pl localhost
.220 laptop ready
CMD: HELO:%n%n%n%n...
CMD: MAIL FROM:%n%n%n%n...
CMD: RCPT TO:%n%n%n%n...
CMD: DATA:%n%n%n%n...
CMD: RSET:%n%n%n%n...
CMD: NOOP:%n%n%n%n...
CMD: VRFY:%n%n%n%n...
CMD: EXPN:%n%n%n%n...
CMD: VERB:%n%n%n%n...
.CMD: ETRN:%n%n%n%n...
CMD: DSN:%n%n%n%n...
CMD: AUTH:%n%n%n%n...
CMD: STARTTLS:%n%n%n%n...
.CMD: QUIT:%n%n%n%n...
.CMD: EHLO:%n%n%n%n...
CMD: HELO:%n@%n.%n%n@%n.%n%n@%n.%n%n@%n.%n...
```

```
CMD: MAIL FROM:%n@%n.%n%n@%n.%n%n@%n.%n%n@%n.%n...
CMD: RCPT TO:%n@%n.%n%n@%n.%n%n@%n.%n%n@%n.%n...
CMD: DATA:%n@%n.%n%n@%n.%n%n@%n.%n%n@%n.%n...
```

Nakon zadnje naredbe DATA, SMTP poslužitelj više ne odgovara, a nakon analize *debuger* alatom, otkriven je *format string* propust kao što je prikazano u nastavku.

```
Segmentation fault (core dumped)
GNU gdb Red Hat Linux (5.1.90CVS-5)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
Core was generated by `qwik-smtpd'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x42051f75 in vfprintf () from /lib/i686/libc.so.6
(gdb)
```

Označena linija predstavlja nasilno prekidanje rada programa u `vfprintf` funkciji, što upućuje na *format string* ranjivost.

5. Analiza sigurnosti u samom izvršavanju programa

Proces u izvršavanju ponekad je lakše analizirati ukoliko se za analizu odaberu specifične funkcije koje mogu uzrokovati sigurnosni propust. U tu svrhu LSS grupa (http://security.lss.hr) je razvila `LibAudit` biblioteku koja preučitava neke potencijalno ranjive `GLIBC` pozive i prijavljuje ih ukoliko se pozivaju od strane ciljne aplikacije (čija sigurnost se analizira). Ova biblioteka posebno je korisna ukoliko se analizira sigurnost većih aplikacija, a pogotovo za programe za koje nije dostupan izvorni kod. `LibAudit` trenutno preučitava `strcpy`, `strncpy`, `getenv`, `fprintf`, `sprintf`, `snprintf`, `vsprintf` i `vsnprintf` funkcije. Ukoliko ciljna aplikacija pozove neku od funkcija koje `LibAudit` preučitava, ispisuje se poruka koja može izgledati otprilike ovako:

```
[LibAudit] strcpy (0xbfffd2c0[32],"CERT+LSS"[0xbfffd464])
```

Iz primjera je vidljivo da ciljna aplikacija poziva `strcpy` funkciju, sa prvim argumentom (odredišni spremnik) koji se nalazi na adresi `0xbfffd2c0` i ima rezervirana 32 okteta, te sa drugim argumentom koji je `"CERT+LSS"`, a nalazi se na adresi `0xbfffd464`. `LibAudit` u ovoj verziji može saznati veličinu spremnika samo ako se radi o stog dijelu memorijskog prostora. Također je važno napomenuti da se veličina argumenta kod ispisa argumenata funkcije skraćuje na maksimalno 50 znakova.

`LibAudit` biblioteka se prevodi na sljedeći način:

```
gcc libaudit.c -c –fPIC
ld  -o libaudit.so libaudit.o -ldl -shared –lc
```

Učitavanje biblioteke u memoriju procesa koji će se pokrenuti:

```
# export LD_PRELOAD="/root/PROJECTS/LIBAUDIT/libaudit.so
```

`LibAudit` biblioteka priložena je u Dodatku A, a način korištenja biblioteke prikazan je u nastavku na nekoliko jednostavnih primjera.

5.1. Ranjiva strcpy funkcija

Prvi primjer se odnosi na klasičan preljev spremnika uzrokovan pogrešnim korištenjem `strcpy` funkcije. U nastavku je priložen izvorni kod za prvi primjer.

`Vuln1.c`

```
main (int argc, char **argv)
{
        char buf[32];

        printf ("Trazenje propusta primjer #1\n");
        printf ("-----------------------------\n");
        if (argc != 2)
        {
```

```
                printf ("Upotreba: %s <argument>\n\n",argv[0]);
                exit(-1);
        }

        strcpy (buf, argv[1]);
        printf ("Ispis: %s\n", buf);
}
```

Kao što je vidljivo iz izvornog koda, program je ranjiv na klasični preljev spremnika koji je označen žutom bojom. U nastavku je prikazano prevođenje programa i njegova analiza pomoću `LibAudit` biblioteke.

```
[root@laptop MYFUZZER]# gcc vuln1.c –o vuln1
[root@laptopMYFUZZER]# export LD_PRELOAD="/root/PROJECTS/LIBAUDIT/libaudit.so
[root@laptop MYFUZZER]# ./vuln1
Trazenje propusta primjer #1
----------------------------
Upotreba: ./vuln1 <argument>

[root@laptop MYFUZZER]# ./vuln1 CERT+LSS
Trazenje propusta primjer #1
----------------------------
[LibAudit] strcpy (0xbfffd2c0[32],"CERT+LSS"[0xbfffd464])
Ispis: CERT+LSS
[root@laptop MYFUZZER]# ./vuln1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Trazenje propusta primjer #1
----------------------------
[LibAudit] strcpy
(0xbfffd290[32],"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"[0xbfffd4
2f])
Ispis: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
[root@laptop MYFUZZER]#
```

Označena linija predstavlja poruku koju ispisuje `LibAudit` biblioteka. Iz poruke je vidljivo da se unutar `vuln1` programa poziva `strcpy` funkcija koja kopira parametre iz naredbenog retka u spremnik veličine 32 okteta, što upućuje na preljev spremnika. Program na kraju dobiva poruku *Segmentation fault* zbog preljeva spremnika koji je prepisao dio stog memorije.

5.2. Format string propust

Format string propuste jako je lako otkriti pomoću `LibAudit` biblioteke. Umjesto cjelovite analize izvornog ili asemblerskog koda, moguće je pratiti korisnički unos kroz argumente funkcija koje ispisuje `LibAudit` biblioteka. Tako je moguće otkriti *format string* funkcije koje korisniku omogućavaju da sam specificira *format string* argument i tako iskoristi *format string* propust. U nastavku je prikazan program koji je ranjiv na *format string* propust.

`Vuln2.c`

```
#include <stdio.h>

main (int argc, char **argv)
{
        char buf[32];

        printf ("Trazenje propusta primjer #2\n");
        printf ("----------------------------\n");
        if (argc != 2)
        {
                printf ("Upotreba: %s <argument>\n\n",argv[0]);
                exit(-1);
        }

        strncpy (buf, argv[1], sizeof(buf));
        fprintf (stdout, buf);
}
```

Označena linija predstavlja sam *format string* propust. Korisnik je u mogućnosti odrediti *format string* argument `fprintf` funkcije i na taj način može iskoristiti propust. U nastavku je prikazano prevođenje, pokretanje i analiza programa `vuln2`.

```
[root@laptop MYFUZZER]# gcc vuln2.c –o vuln1
[root@laptop MYFUZZER]# export LD_PRELOAD=/root/PROJECTS/LIBAUDIT/libaudit.so
[root@laptop MYFUZZER]# ./vuln2 CERT+LSS
Trazenje propusta primjer #2
----------------------------
[LibAudit] strncpy (0xbfffd2c0[32], "CERT+LSS"[0xbfffd462], 32)
[LibAudit] fprintf (1, "CERT+LSS"[0xbfffd2c0],CERT+LSS)

CERT+LSS[root@laptop MYFUZZER]# ./vuln2 CERT+LSS.%x.%x.%x.%x
Trazenje propusta primjer #2
----------------------------
[LibAudit] strncpy (0xbfffd2b0[32], "CERT+LSS.%x.%x.%x.%x"[0xbfffd456], 32)
[LibAudit] fprintf (1,
"CERT+LSS.%x.%x.%x.%x"[0xbfffd2b0],CERT+LSS.20.bfffd344.54524543.53534c2b)
CERT+LSS.20.bfffd344.54524543.53534c2b
```

Prva označena linija predstavlja `fprintf` funkciju koja ispisuje korisnički unos na standardni izlaz. Kao što je vidljivo, korisnik može sam odrediti *format string* argument te funkcije (u ovom slučaju «CERT+LSS») , što automatski upućuje na *format string* propust. Drugo pokretanje programa sa argumentom «CERT+LSS.%x.%x.%x.%x» otkriva sam *format string* propust, jer korisnik specificira *format string* koji je u ovom slučaju '%x' i čita memoriju sa stoga. Druga žuto označena linija ispisuje argumente `fprintf` funkcije i otkriva *format string* kojeg je odredio korisnik.

5.3. Praćenje korisničkog unosa

Nakon što korisnik na neki način preda podatke određenom programu, podaci "teku" kroz više međuspremnika i sistemskih poziva. Praćenjem tih podataka moguće je otkriti funkcije koje obavljaju operacije nad njima, a rade to na pogrešan način. U nastavku je priložen program koji pomoću `getenv` poziva uzima sadržaj USER varijable okruženja i kopira ga u neki spremnik na stogu ranjivom funkcijom `strcpy`.

```
main (int argc, char **argv)
{
        char buf[32], *lang;

        printf ("Trazenje propusta primjer #3\n");
        printf ("----------------------------\n");
        lang = getenv ("USER");
        strcpy (buf, lang);
        printf ("Ispis: %s\n", buf);
}
```

U nastavku je prikazano prevođenje, pokretanje i analiza `vuln3` programa.

```
[root@laptop MYFUZZER]# gcc vuln3.c –o vuln3
[root@laptop MYFUZZER]# export LD_PRELOAD=/root/PROJECTS/LIBAUDIT/libaudit.so
[root@laptop MYFUZZER]# ./vuln3
Trazenje propusta primjer #3
----------------------------
[LibAudit] bfffd5c2 = getenv ("USER")
[LibAudit] strcpy (0xbfffd350[32],"root"[0xbfffd5c2])
Ispis: root
[root@laptopMYFUZZER]# export USER=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

[root@laptop MYFUZZER]# ./vuln3
Trazenje propusta primjer #3
----------------------------
[LibAudit] bfffd54e = getenv ("USER")
[LibAudit]strcpy(0xbfffd2d0[32],"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA"[0xbfffd54e])
Ispis:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
[root@laptop MYFUZZER]#
```

Iz prve dvije označene linije vidljivo je da program dobiva pokazivač na sadržaj USER varijable pomoću `getenv` poziva. Pokazivač je na adresi `0xbfffd5c2` čiji sadržaj se kasnije kopira pomoću ranjive `strcpy` funkcije u spremnik na stogu koji se nalazi na adresi `0xbfffd350` i veličine je 32 okteta. Prije sljedećeg pokretanja `vuln3` programa, varijabla USER se postavlja na velik niz znakova, što pri kopiranju `strcpy` funkcijom izaziva preljev spremnika (druge dvije označene linije).

6. Zaključak

Analiza i testiranje sigurnosti programske podrške vrlo je važan proces kojim je moguće pravovremeno uočiti i otkloniti sigurnosne propuste u računalnom sustavu. *Fuzzing* tehnike i metode otkrivanja sigurnosnih propusta, analizirane u dokumentu, mogu se primijeniti na bilo kojoj aplikaciji čija je sigurnost ključna za pouzdano i nesmetano izvršavanje određenih zadataka. No, iako jednostavne, i vrlo često iznimno učinkovite, korištenje ovih tehnika ipak zahtijeva određenu razinu znanja i iskustva osobe koja provodi analizu.

7. Reference

[1] Jack Koziol, *"Shellcoders Handbook"*
[2] SPIKE *fuzzer*, http://www.immunitysec.com

8. DODATAK A

LibAudit.c

```
/*
 *
 * Library Audit [LibAudit] v0.1
 * ----------------------------
 * This simple library will trace some potentially dangerous function calls
 * from program that you audit.
 *
 * Coded by Leon Juranic <ljuranic@lss.hr> / http://security.lss.hr
 *
 */

#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <dlfcn.h>
#include <stdarg.h>

typedef struct _IO_FILE FILE;
extern FILE *stdin;             /* Standard input stream.  */
extern FILE *stdout;            /* Standard output stream.  */
extern FILE *stderr;

char *(*getenv_orig)(const char *name);
char *(*strcpy_orig)(char *dest, const char *src);
char *(*strncpy_orig)(char *dest, const char *src, size_t n);
int  (*fprintf_orig)(FILE *stream, const char *format, ...);
int  (*vsprintf_orig)(char *str, const char *format, va_list ap);
int   (*vsnprintf_orig)(char  *str,  size_t  size,  const  char  *format,  va_list
ap);

long get_memsize (char *block) // adopted from "Shellcoder's Handbook" from
win to linux
{
        int num=0, memsize=0;
        int *frame;
        int *prev_frame;
        int *stack_base = (int*)0xbfffffff;
        int *stack_top  = (int*)0xb0000000;

        asm ("movl %%ebp, %0" : "=r"(frame));

        if (block < (char*)stack_base && block > (char*)stack_top)
           for (num=0;num<=5;num++) {
               if (frame < (int*)stack_base && frame > (int*)stack_top)
               {
                   prev_frame = (int*)*frame;
                   if (prev_frame < stack_base && prev_frame > stack_top) {
                       if (frame < (int*)block & (int*)block < prev_frame)
                       {
                               memsize = (int)prev_frame - (int)block - 8;
```

```
                        break;
                    }
                   else frame = prev_frame;
                 }
               }
             }
         return (memsize);
}



int  fprintf (FILE *stream, const char *format, ...)
{
       va_list ap;
       va_start (ap, format);
       printf      ("[LibAudit]      fprintf      (%d,      \"%.50s\"[0x%x],",
fileno(stream),format,format);
       vfprintf (stdout, format, ap);
       printf (")\n");
       va_end (ap);
       return vfprintf (stream, format, ap);
}

int vsprintf(char *str, const char *format, va_list ap)
{
       printf     ("[LibAudit]     vsprintf     (%x[%d],     \"%.50s\"[0x%x],",
str,get_memsize(str),format,format);
       vfprintf (stdout, format, ap);
       printf (")\n");
       return vsprintf_orig (str, format, ap);
}


int vsnprintf(char *str, size_t size, const char *format, va_list ap)
{
       printf   ("[LibAudit]   vsnprintf   (%x[%d],   %d,   \"%.50s\"[0x%x],",
str,get_memsize(str),size, format,format);
       vfprintf (stdout, format, ap);
       printf (")\n");
       return vsnprintf_orig (str, size, format, ap);

}


int sprintf(char *str, const char *format, ...)
{
        va_list ap;
        va_start (ap, format);
        printf     (">     sprintf     (0x%x[%d],     \"%.50s\"[0x%x],",     str,
get_memsize(str), format,format);
        vfprintf (stdout, format, ap);
        printf (")\n");
        va_end (ap);
       return vsprintf_orig (str, format, ap);
}



int  snprintf (char *str, size_t size, const char *format, ...)
{
        va_list ap;
        va_start (ap, format);
        printf ("[LibAudit] snprintf (0x%x[%d], %d, \"%.50s\"[0x%x],", str,
get_memsize(str), size, format, format);
       vfprintf (stdout, format, ap);
       printf (")\n");
       va_end (ap);
       return vsnprintf_orig (str, size, format, ap);
}

char *getenv(const char *name)
{
       long x;
```

```
        char *ret = getenv_orig(name);
        printf ("[LibAudit] %x = getenv (\"%.50s\")\n",ret,name);
        return ret;
}


char *strcpy (char *dest, const char *src)
{
        printf                        ("[LibAudit]                        strcpy
(0x%x[%d],\"%.50s\"[0x%x])\n",dest,get_memsize(dest),src,src);
        return strcpy_orig(dest,src);
}


char *strncpy (char *dest, const char *src, size_t n)
{
        printf     ("[LibAudit]     strncpy     (0x%x[%d],     \"%.50s\"[0x%x],
%d)\n",dest,get_memsize(dest),src,src,n);
        return strncpy_orig(dest,src,n);
}

void _init ()
{
        void *hdl;
        char *error;

        hdl = dlopen ("/lib/libc.so.6",RTLD_LAZY);
        if (!hdl) {
                fputs (dlerror(), stderr);
                exit(1);
        }

        getenv_orig    = dlsym (hdl, "getenv");
        strcpy_orig    = dlsym (hdl, "strcpy");
        strncpy_orig   = dlsym (hdl, "strncpy");
        fprintf_orig   = dlsym (hdl, "fprintf");
        vsprintf_orig  = dlsym (hdl, "vsprintf");
        vsnprintf_orig = dlsym (hdl, "vsnprintf");


        if ((error = dlerror()) != NULL)  {
                fprintf (stderr, "%s\n", error);
                exit(1);
        }

}
```