HRVATSKA AKADEMSKA I ISTRAŽIVAČKA MREŽA
CROATIAN ACADEMIC AND RESEARCH NETWORK

Zaštita binarnih ELF datoteka

CCERT-PUBDOC-2005-09-134

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

Zbog različitih ciljeva bilo privatnih ili poslovnih, u posljednje vrijeme sve se više radi na području zaštite intelektualnog vlasništva nad programskim rješenjima. S obzirom da se softver najčešće distribuira u obliku izvršnih datoteka i njihovih komponenata (izuzevši *s*oftver pod GPL licencom), veliki se napori ulažu u mehanizme njihove zaštite od neovlaštene upotrebe. Zaštita binarnih datoteka odnosi se uglavnom na zaštitu programa od neovlaštenog kopiranja, pokretanja i analize, kao i označavanje programa nekim oblikom žiga (npr. vodenim žigom).
Tehnologija zaštite binarnih datoteka na Windows operacijskim sustavima vrlo je razvijena, dok se o zaštiti binarni datoteka na Linux operacijskim sustavima počelo razmišljati tek prije nekoliko godina. Standardni format izvršne datoteke na Windows sustavima naziva se ***P**ortable **E**xecutable* (PE), i za njega postoji velik broj besplatnih i komercijalnih alata koji omogućuju zaštitu datoteke. Standardni format izvršne datoteke na Linux sustavima naziva se ***E**xecutable and **L**inking **F**ormat* (ELF) i za njega do nedavno nije postojao skoro nikakav oblik zaštite. Ta činjenica inicirala je velik broj istraživačkih projekata koji se bave upravo analizom različitih mogućnosti zaštite ELF binarnih datoteka. Također, treba napomenuti da kod analize zaštićenih izvršnih datoteka, veliku ulogu igra i sučelje sa kojim korisnik može nadgledati, odnosno analizirati tijek izvršavanja zaštićenog programa. Na Linux/Unix operacijskim sustavima tijek izvršavanja programa temelji se na vrlo ograničenom `ptrace()` sistemskom pozivu.
Prilikom zaštite binarnih datoteka postoji nekoliko različitih metoda zaštite od kojih su neke navedene u nastavku:

- **Enkripcija binarne datoteke** – ukoliko je izvršna datoteka kriptirana, korisnik koji ju želi pokrenuti mora znati odgovarajući ključ da bi dekriptirao sadržaj datoteke.
- **Korištenje anti-debuging tehnika** – tehnike koje onemogućavaju ili otežavaju *runtime* analizu zaštićene izvršne datoteke korištenjem propusta u samom *debugeru*, odnosno disasembleru, ili sučelju koje se koristi za analizu (npr. ranije spomenuti `ptrace()`). Također, postoje tehnike u kojima se izvršavanje zaštićenog programa realizira putem procesa roditelja (engl. *parent*) i procesa djeteta (engl. *child*), tako da roditelj nadgleda izvršavanje djeteta i mijenja njegov memorijski prostor po potrebi, što znatno otežava analizu zaštićenog programa.
- **Maskiranje programskog koda (engl. *obfuscation*)** – programski kod izvršne datoteke moguće je prilično zakomplicirati, a da se pritom sama logika programa ne promijeni. Ova metoda bitno otežava forenzičku analizu zaštićene izvršne datoteke zbog količine netočnih i nepotrebnih podataka kojima se osoba koja provodi analizu navodi na krivi put.
- **Korištenje virtualne CPU jedinice** – virtualna CPU jedinica je programski implementiran CPU (sa vlastitim setom programskih instrukcija) u zaštićenoj izvršnoj datoteci. Putem virtualne CPU jedinice izvode se programske instrukcije programa u kojem je sam CPU implementiran. Za analizu izvršne datoteke zaštićene ovom metodom potrebno je prvo analizirati rad virtualne CPU jedinice, što dodatno otežava postupak analize.
- **Osiguravanje integriteta** – neki alati za binarnu enkripciju u zaštićenu izvršnu datoteku implementiraju provjere koje osiguravaju da se sama binarna datoteka nije mijenjala nakon enkripcije. Ova razina zaštite može se vrlo lako implementirati pomoću `checksum` funkcija.

Dokument opisuje različite tehnike zaštite binarnih ELF datoteka, pri čemu je kao testna platforma korišten Linux operacijski sustav. Važno je napomenuti da neke od analiziranih tehnika zajedno sa dobivenim rezultatima mogu varirati ovisno o tipu i inačici operacijskog sustava te pripadajućim alatima. Za analizu primjera u dokumentu korišteni su standardni Linux alati kao što su `objdump`, `strace`, `ltrace`, `readelf` i `gdb`.

2. Struktura ELF binarne datoteke

U nastavku poglavlja opisana je osnovna struktura ELF datoteke. ELF datoteke mogu se pronaći u četiri najčešća oblika:

1. **Izvršna datoteka** (engl. *executable file*) odnosno `ET_EXEC` – označava klasične izvršne datoteke. U takvom formatu su svi Linux alati.

2. **Programska biblioteka** (engl. *shared object file*) odnosno ET_DYN – označava programske biblioteke koje sadrže podatke i programski kod koji *linker* koristi za povezivanje (engl. *linking*). Pri pokretanju neke izvršne datoteke (ET_EXEC), u slučaju da nije prevedena statički, *linker* odnosno *loader* kombiniraju programske biblioteke i izvršnu datoteku kako bi se kreirao proces u radnoj memoriji sustava.
3. ***Relocatable* objektna datoteka** odnosno ET_REL – označava objektne datoteke koje se povezuju sa drugim objektnim datotekama za potrebe kreiranje izvršne datoteke ili programske bilioteke.
4. ***Core* datoteka** odnosno ET_CORE – označava datoteke koje se kreiraju u slučaju da proces nasilno prekine svoje izvršavanje (engl. *crash*). *Core* datoteka sadrži sliku (engl. *image*) procesa u memoriji u trenutku kada je proces prekinuo s izvršavanjem.

2.1. Zaglavlje ELF datoteke

Svaka ELF datoteka mora sadržavati zaglavlje (engl. *header*) koje opisuje osnovnu strukturu datoteke. Zaglavlje se nalazi na samom početku binarne datoteke i uvijek je iste veličine. Zaglavlje je najbitnije za uspješno učitavanje i pokretanje ELF datoteke, budući da ono opisuje gdje se nalaze ostale važne strukture unutar iste datoteke.

U nastavku je prikazana struktura koja predstavlja zaglavlje ELF datoteke.

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

U nastavku su ukratko opisani elementi strukture zaglavlja `Elf32_Ehdr`.

- **e_ident** – označava da se radi o objektnoj datoteci i specificira vrijednosti kao npr. da li se radi o 32 bitnim ili 64 bitnim objektima, *little endian* ili *big endian* arhitekturi, i sl. Prva četiri okteta označavaju da se radi o ELF binarnoj datoteci i sadrže niz okteta '\x7f','E','L','F'.
- **e_type** – označava ranije spomenuti tip ELF datoteke. Može sadržavati vrijednosti ET_EXEC, ET_REL, ET_CORE i ET_DYN.
- **e_machine** – označava tip arhitekture za koju je datoteka namijenjena. Može sadržavati vrijednosti kao što su EM_SPARC, EM_MIPS, EM_386, EM_SPARCV9, EM_68K, EM_IA_64.
- **e_version** – označava inačicu ELF datoteke.
- **e_entry** – definira ulaznu točku (engl. *entry point*) koja označava virtualnu adresu nakon mapiranja programa u memoriju na kojoj će se program početi izvršavati. Ulazna točka se kod ELF datoteka obično specificira simbolom "`_start`".
- **e_phoff** - predstavlja udaljenost (engl. *offset*) od početka ELF datoteke na kojoj se nalazi tabela programskih zaglavlja (engl. *program header*).
- **e_shoff** – predstavlja udaljenost (engl. *offset*) od početka ELF datoteke na kojoj se nalazi tabela sekcijskih zaglavlja (engl. *section header*).
- **e_flags** – sadrži zastavice vezane uz ELF datoteku koje su specifične za procesorsku jedinicu.
- **e_ehsize** – sadrži veličinu zaglavlja ELF datoteke.
- **e_phentsize** – sadrži veličinu jednog unosa u tabeli programskih zaglavlja.
- **e_phnum** – sadrži broj elemenata u tabeli programskih zaglavlja.
- **e_shentsize** – sadrži veličinu jednog unosa u tabeli sekcijskih zaglavlja.

- **e_shnum** – sadrži broj elemenata u tabeli sekcijskih zaglavlja.
- **e_shstrndx** – sadrži indeks koji pokazuje na sekciju (unutar tabele sekcijskih zaglavlja) u kojoj su definirana imena sekcijskih zaglavlja.

2.2. Sekcijsko zaglavlje

Binarna ELF datoteka sastoji se od nekoliko sekcija. Svaka sekcija sadrži niz oketa koji pripadaju samo toj sekciji, pri čemu se sadržaj sekcija ne smije preklapati. Sekcije ELF datoteke opisane su u tabeli sekcijskih zaglavlja. Tabela sekcijskih zaglavlja sadrži niz `Elf32_Shdr` struktura od kojih svaka opisuje jednu sekciju u ELF datoteci. Prvi unos u tabeli sekcijskih zaglavlja obično je nula i ne opisuje ni jednu sekciju. Važno je napomenuti da sekcijska zaglavlja nisu bitna za učitavanje programa u memoriju i njegovo normalno izvršavanje. Iz tog razloga moguće je potpuno obrisati tabelu sekcijskih zaglavlja što će znatno otežati analizu ELF binarne datoteke. Još zanimljivije rješenje je postavljanje pogrešnih vrijednosti u tabelu sekcijskih zaglavlja, što osobu koja radi analizu ELF datoteke može navesti na krivi put. U nastavku je priložena `Elf32_Shdr` struktura jednog sekcijskog zaglavlja.

```
typedef struct
{
  Elf32_Word    sh_name;                /* Section name (string tbl index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds table */
} Elf32_Shdr;
```

U nastavku su ukratko opisani elementi strukture `Elf32_Shdr`.

- **sh_name** – označava indeks unutar sekcije koja sadrži imena sekcija. Ime sekcije može biti npr. "`text`", "`rodata`", "`bss`" itd.
- **sh_type** – definira sadržaj sekcije koju opisuje sekcijsko zaglavlje.
- **sh_flags** - označava atribute sekcije. Atributi mogu biti tipa `SHF_WRITE`, `SHF_ALLOC`, `SHF_EXECINSTR` i `SHF_MASKPROC`. Prva tri atributa označavaju da li se prilikom izvršavanja programa u sekciju može pisati, da li se sekcija mapira u memoriju prilikom pokretanja programa i da li se podaci u sekciji izvršavaju kao programske instrukcije.
- **sh_addr** – označava virtualnu adresu na kojoj će se nalaziti prvi oktet sekcije nakon što se program mapira u memoriju.
- **sh_offset** – označava udaljenost (engl. *offset*) od početka ELF datoteke na kojoj se nalazi prvi oktet sekcije.
- **sh_size** – označava veličinu sekcije u oktetima.
- **sh_link** – sadrži indeks vezu unutar tabele sekcijskih zaglavlja čija interpretacija ovisi o tipu sekcije.
- **sh_info** – sadrži dodatne informacije o sekciji čija interpretacija također ovisi o tipu sekcije.
- **sh_addralign** – sadrži informaciju o poravnavanju (engl. *align*) sekcije u radnoj memoriji sustava.
- **sh_entsize** – opisuje veličinu jednog elementa unutar sekcije i koristi se za sekcije čiji sadržaj je uvijek iste veličine (npr. tablica simbola).

U nastavku je prikazan ispis nekoliko sekcijskih zaglavlja izvršne ELF datoteke pomoću `objdump` alata.

```
root@t-rex:~/PROJECTS/ELF# objdump -h ./seo-test

./seo-test:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         00000017  080480d4  080480d4  000000d4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text         0004f560  08048100  08048100  00000100  2**5
```

```
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 __libc_freeres_fn 000007a0  08097660  08097660  0004f660  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .fini         0000001b  08097e00  08097e00  0004fe00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .rodata       00015e60  08097e20  08097e20  0004fe20  2**5
....
....
....
```

Označena linija predstavlja sekcijsko zaglavlje koje opisuje `text` sekciju koja sadrži programske instrukcije programa. Kao što je vidljivo iz priloženog primjera, ime sekcije je "`.text`", veličina sekcije je `0x4f560` okteta, a pri pokretanju programa sekcija će se mapirati na adresu `0x08048100`. Udaljenost prvog bajta sekcije od početka ELF datoteke je 0x100 okteta. Sekcija sadrži programske instrukcije i po njoj se ne može pisati, a pri pokretanju program se mapira u memoriju.

2.3. Programsko zaglavlje

Za mapiranje programa u memoriju koristi se tabela programskih zaglavlja, i za razliku od tabele sekcijskih zaglavlja ona je vrlo važna za mapiranje i pokretanje programa. Tabela programskih zaglavlja sastoji se od niza `Ehdr32_Phdr` struktura koje definiraju segmente unutar ELF binarne datoteke. Svaki segment se obično sastoji od više programskih sekcija. U nastavku je prikazan izgled `Ehdr32_Phdr` strukture.

```
typedef struct
{
  Elf32_Word     p_type;          /* Segment type */
  Elf32_Off      p_offset;        /* Segment file offset */
  Elf32_Addr     p_vaddr;         /* Segment virtual address */
  Elf32_Addr     p_paddr;         /* Segment physical address */
  Elf32_Word     p_filesz;        /* Segment size in file */
  Elf32_Word     p_memsz;         /* Segment size in memory */
  Elf32_Word     p_flags;         /* Segment flags */
  Elf32_Word     p_align;         /* Segment alignment */
} Elf32_Phdr;
```

U nastavku su ukratko opisani elementi strukture.

- **p_type** - definira tip segmenta. Za razumijevanje ovog dokumenta, važni su tipovi segmenata `PT_LOAD` (sadržaj segmenta se mapira u memoriju prilikom pokretanja programa) i `PT_NOTE` (segment sadrži dodatne, uglavnom nebitne informacije, i ne mapira se u memoriju).
- **p_offset** – definira udaljenost prvog okteta segmenta od početka ELF datoteke.
- **p_vaddr** – definira virtualnu memorijsku adresu na koju će se mapirati segment.
- **p_paddr** – definira fizičku adresu segmenta za sustave na kojima je važna fizička adresa.
- **p_filesz** – definira veličinu segmenta u ELF datoteci.
- **p_memsz** – definira veličinu segmenta u radnoj memoriji sustava.
- **p_flags** - definira zastavice važne za pojedini segment. Zastavice određuju koje operacije se mogu izvršavati nad podacima unutar segmenta. Npr. da li se po segmentu može pisati ili ne.
- **p_align** – definira poravnavanje segmenta u memoriji. Segmenti tipa `PT_LOAD` moraju imati poravnavanje na bazi veličine stranice (`PAGE_SIZE`) koja je obično 4096.

U nastavku je prikazan sadržaj tabele programskih zaglavlja ELF datoteke pomoću `readelf` programa.

```
root@t-rex:~/PROJECTS/ELF# readelf -l ./seo-test


Elf file type is EXEC (Executable file)
Entry point 0x8048100
There are 4 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x66b54 0x66b54 R E 0x1000
  LOAD           0x067000 0x080af000 0x080af000 0x00cf8 0x01a08 RW  0x1000
```

```
  NOTE           0x0000b4 0x080480b4 0x080480b4 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
....
....
....
```

Kao što je vidljivo iz primjera, ulazna točka programa je `0x08048100`, ELF datoteka je tipa `ET_EXEC` i sadrži četiri programska zaglavlja koji opisuju segmente. Označena linija predstavlja segment unutar kojeg se nalazi i "`text`" sekcija. Segment je tipa `PT_LOAD`, udaljenost od početka ELF datoteke mu je nula, mapira se na virtualnu adresu 0x08048000, veličine je 0x66b54 okteta, ima zastavice `READONLY` i `EXECUTABLE` i poravnava se na 0x1000 (4096 decimalno), odnosno `PAGE_SIZE`.

3. Anti-debugging tehnike

Kako bi se otkrio ustroj i sadržaj ELF datoteke, potrebno ju je analizirati alatima koji su razvijeni u te svrhe. Popularni alati za Linux operacijske sustave dostupni za analizu binarnih ELF datoteka su `objdump`, `readelf`, `gdb`, `strace`, `ltrace`, itd. `Objdump` i `gdb` svoju analizu baziraju na `libbfd` (engl. *Binary File Descriptor Library*) biblioteci, pa njihova funkcionalnost zavisi od funkcionalnosti navedene biblioteke. Za analizu izvršavanja programa, `gdb` koristi prije spomenuto `ptrace()` sučelje koje je vrlo primitivno i prilično ga je lako onemogućiti. `Readelf` je alat sličan `objdump` alatu, no umjesto `libbfd` biblioteke, on koristi vlastite funkcije za interpretaciju sadržaja ELF datoteke. `Strace` i `ltrace` služe za analizu programa u izvršavanju (engl. *runtime analysis*) i također koriste `ptrace()` sučelje što ih čini ranjivim na sve *anti-debuging* tehnike vezane uz `ptrace()` poziv.

Anti-debuging tehnike odnose se na otežavanje, ili u nekim slučajevima potpuno onemogućavanje analize ELF datoteka, a baziraju se na nedostacima alata ili sučelja koji se koriste za analizu izvršne datoteke.

3.1. ptrace() tehnika

Za analizu i kontrolu programa u izvršavanju, na UNIX odnosno Linux sustavima postoji već spomenuto `ptrace()` sučelje, koje je dostupno putem `ptrace()` sistemskog poziva. Pomoću `ptrace()` sučelja, proces može analizirati ili kontrolirati bilo koji drugi proces na sustavu za koji ima potrebne ovlasti.

U nastavku je prikazan prototip `ptrace()` sistemskog poziva.

```
long  ptrace(enum __ptrace_request request, pid_t pid,void *addr, void *data);
```

Prvi argument je vrsta operacije koja se želi izvršiti. Npr. `PTRACE_ATTACH` za povezivanje na proces koji se želi analizirati, `PTRACE_DETACH` za odspajanje sa istog procesa, `PTRACE_KILL` za terminiranje procesa, `PTRACE_TRACEME` ukoliko proces (obično proces dijete) očekuje da bude nadziran od strane nekog drugog procesa, itd. Drugi argument je PID broj procesa koji će se analizirati. Treći i četvrti argument su adresa i podatak ukoliko se radi o pisanju ili čitanju po memorijskom prostoru procesa koji se analizira. Važno je napomenuti da proces može biti analiziran samo od strane jednog procesa, dok će svi drugi `ptrace()` pokušaji na taj proces rezultirati greškom (`ptrace()` poziv će vratiti vrijednost -1). Ta činjenica može se iskoristiti za implementaciju *anti-debug* tehnike.

Ukoliko je neki proces već nadziran od strane nekog drugog procesa, pokušaj pokretanja `PTRACE_TRACEME` operacije rezultirati će greškom, a ukoliko proces nije nadziran, `ptrace()` sistemski poziv će vratiti vrijednost 0. U nastavku je priložen program koji prikazuje prethodno objašnjenu *anti-debug* tehniku.

anti-ptrace.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>


main ()
{
```

```
        if (ptrace (PTRACE_TRACEME,0,0,0) < 0)
        {
                printf ("DEBUGER DETEKTIRAN!!!\n");
                exit(-1);
        }
        else printf ("NEMA DEBUGERA!!!\n");
}
```

U nastavku je prikazano prevođenje i pokretanje `anti-ptrace.c` programa u normalnom okruženju.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gcc anti-ptrace.c -o ptrace
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# ./ptrace
NEMA DEBUGERA!!!
```

Kao što je vidljivo iz primjera, program je normalno izvršio PTRACE_TRACEME operaciju što znači da ni jedan drugi proces ne nadzire taj `ptrace` proces. U nastavku je prikazano pokretanje programa unutar `gdb` *debugera*.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gdb ./ptrace
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) r
Starting program: /home/testme/ptrace
DEBUGER DETEKTIRAN!!!

Program exited with code 0377.
(gdb)
```

Označena linija predstavlja pristutnost *debugera* jer je proces nadziran i `ptrace()` sistemski poziv je za PTRACE_TRACEME operaciju vratio vrijednost -1. Ova jednostavna `ptrace()` *anti-debug* tehnika vrlo se često koristi u svrhu otkrivanja *debugera* i prilično je korisna.

3.2. SIGTRAP tehnika

Proces prilikom izvršavanja može primiti različite signale koji ga upućuju na neko specijalno stanje ili aktivnost. Npr. ako proces pokuša čitati memorijsku adresu izvan svog adresnog prostora primit će SIGSEGV signal (koji obično rezultira *"Segmentation fault (core dumped)"* porukom), ili ako primi SIGKILL signal izvršavanje procesa bezuvjetno će se prekinuti. Proces može imati rukovatelj signala (engl. *handler*) koji u slučaju primitka određenog signala izvršava određenu funkciju, ili ga samo ignorira. Izuzetak je SIGKILL signal koji ne može imati svojeg rukovatelja. Rukovatelji signala postavljaju se sistemskim pozivom `signal()`koji definira o kojem signalu se radi i koja funkcija se pokreće ukoliko proces primi taj signal. Ukoliko se program nadzire putem `ptrace()` sučelja, svaki signal koji proces dobiva uzrokuje prekid izvršavanja procesa i proces koji ga nadzire dobiva SIGCHLD signal nakon čega može utvrditi o kakvom se signalu radi i sl. Rukovatelji signala postavljeni sa strane procesa koji se nadzire se ne izvršavaju. Interesantan signal je SIGTRAP koji proces dobiva u slučaju nailaska na prekidnu točku (eng. *breakpoint*), odnosno `int3` instrukcije koja se uglavnom koristi za *debugiranje* programa. SIGTRAP označava da je proces došao do prekidne točke i da je izvođenje programa prekinuto. Ukoliko program ima definiran rukovatelj signala za SIGTRAP signal, on će se u tom trenutku izvršiti. Pomoću SIGTRAP signala i odgovarajućeg rukovatelja, moguće je vrlo jednostavno otkriti pristutnost procesa koji nadzire tekući proces. U nastavku je priložen jednostavan program koji otkriva pristutnost *debuger* alata pomoću SIGTRAP tehnike.

sigtrap.c

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

int flag=0;
```

```
void int3 ()
{
       flag++;
}


main ()
{
       signal (SIGTRAP,int3);
       __asm ("int3\n");
       if (flag == 0)
       {
              printf ("DEBUGER JE DETEKTIRAN\n");
              printf ("ANTI-DEBUG TEHNIKA\n");
              exit(-1);
       }
       else printf ("DEBUGER NIJE DETEKTIRAN\n");

}
```

Program ima deklariranu globalnu varijablu `flag` koja će se koristiti za detekciju *debuger* alata. U programu je postavljen rukovatelj signala za SIGTRAP signal. Funkcija koja se izvršava po primitku SIGTRAP signala je `int3()`, a njezina jedina svrha je da inkrementira varijablu `flag`. Unutar `main()` funkcije, nakon što se postavi rukovatelj signala, izvršava se `int3` instrukcija nakon koje proces dobiva SIGTRAP signal i rukovatelj signala (funkcija `int3()`), varijablu `flag` postavlja na vrijednost 1. U slučaju da je na proces spojen neki *debuger*, rukovatelj signala se neće izvršiti i varijabla `flag` se neće inkrementirati. Ukoliko je nakon `int3` instrukcije varijabla `flag` i dalje 0 znači da je proces nadziran i izvođenje programa se prekida.

U nastavku je prikazano prevođenje i pokretanje `sigtrap.c` programa u normalnim okolnostima.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gcc sigtrap.c -o sigtrap
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# ./sigtrap
DEBUGER NIJE DETEKTIRAN
```

Program se je normalno izvršio, varijabla `flag` je postavljena na 1 što znači da je rukovatelj signala izvršen. Izvršavanje programa unutar gdb *debuger* alata prikazano je u nastavku.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gdb ./sigtrap
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) r
Starting program: /root/PROJECTS/ELF/ANTIDEBUG/anti-debug

Program received signal SIGTRAP, Trace/breakpoint trap.
0x08048422 in main ()
(gdb) c
Continuing.
DEBUGER JE DETEKTIRAN
ANTI-DEBUG TEHNIKA

Program exited with code 0377.
(gdb)
```

Kao što je vidljivo iz primjera, gdb javlja da je program primio SIGTRAP signal i izvođenje programa se zaustavlja. Rukovatelj signala kojeg je postavio `sigtrap.c` nije izvršen i program je detektirao prisutnost *debuger* alata.

Analiza `sigtrap.c` programa pomoću `strace` alata prikazana je u nastavku.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# strace ./sigtrap
execve("./sigtrap ", ["./sigtrap"], [/* 45 vars */]) = 0
brk(0)                                  = 0x80496dc
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=91112, ...}) = 0
```

```
old_mmap(NULL, 91112, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0 U\1\000"..., 512) =
512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1357414, ...}) = 0
old_mmap(NULL, 1166612, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
= 0x4002e000
mprotect(0x40144000, 27924, PROT_NONE)  = 0
old_mmap(0x40145000,                16384,              PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x116000) = 0x40145000
old_mmap(0x40149000,                 7444,              PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40149000
close(3)                                = 0
mprotect(0x40145000, 4096, PROT_READ)   = 0
munmap(0x40017000, 91112)               = 0
rt_sigaction(SIGTRAP,     {0x80483f4,     [TRAP],     SA_RESTORER|SA_RESTART,
0x40056db8}, {SIG_DFL}, 8) = 0
fstat64(1, {st_mode=S_IFCHR|0720, st_rdev=makedev(136, 3), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x40017000
write(1, "DEBUGER JE DETEKTIRAN\n", 22DEBUGER JE DETEKTIRAN
) = 22
write(1, "ANTI-DEBUG TEHNIKA\n", 19ANTI-DEBUG TEHNIKA
)    = 19
munmap(0x40017000, 4096)                = 0
exit_group(-1)                          = ?
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG#
```

Rukovatelj signala kojeg je postavio program `sigtrap.c` nije izvršen, varijabla `flag` nije inkrementirana i program detektira prisutnost *debuger* alata, te prekida izvršavanje.

3.3. "Single step" tehnika

Pri analizi izvršavanja programa, program se često izvršava instrukciju po instrukciju kako bi se moglo detaljno analizirati njegovo "ponašanje". Za izvršavanje programa instrukciju po instrukciju u EFLAGS registar procesora postavlja se TF (TRAP) zastavica. U svrhu izvršavanja odnosno analize programa instrukciju po instrukciju koristi se PTRACE_SINGLESTEP operacija u `ptrace()` sučelju. Program koji je analiziran može vrlo lako otkriti da li ga se analizira instrukciju po instrukciju jednostavnom provjerom EFLAGS procesorskog registra za TF zastavicom. U nastavku je priložen program koji otkriva prisutnost *debuger* alata pomoću ispitivanja EFLAGS registra.

anti-single.c

```
int check_trace ()
{

        __asm ("pushf\n"
               "popl %eax\n"
               "and $0x100,%eax\n");
}

main ()
{

       if (check_trace() == 0x100)
       {
               printf ("DEBUGER JE DETEKTIRAN\n");
               printf ("ANTI-DEBUG TEHNIKA\n");
               exit(-1);
        }
        else printf ("DEBUGER NIJE DETEKTIRAN\n");

}
```

Označene linije predstavljaju dohvat sadržaja EFLAGS registra. Instrukcija `pushf` na stog stavlja sadržaj EFLAGS registra, koji se zatim POP instrukcijom pohranjuje u EAX registar nad kojim se obavlja AND operacija sa vrijednošću 0x100 koja ispituje prisutnost TF zastavice. Ukoliko je nakon toga sadržaj EAX registra 0x100, TF zastavica je postavljena i *debuger* je otkriven. U nastavku je prikazano prevođenje i pokretanje programa u normalnom okruženju.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gcc anti-single.c -o anti-single
```

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# ./anti-single
DEBUGER NIJE DETEKTIRAN
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG#
```

Kao što je vidljivo iz primjera, TF zastavica nije otkrivena i *debuger* nije prisutan. U nastavku je prikazano pokretanje programa unutar `gdb` *debugera* pri analizi programa instrukciju po instrukciju.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gdb ./anti-single
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) break *check_trace
Breakpoint 1 at 0x80483c4
(gdb) r
Starting program: /root/PROJECTS/ELF/ANTIDEBUG/anti-single

Breakpoint 1, 0x080483c4 in check_trace ()
(gdb) stepi
0x080483c5 in check_trace ()
(gdb)
0x080483c7 in check_trace ()
(gdb)
0x080483c8 in check_trace ()
(gdb)
0x080483c9 in check_trace ()
(gdb)
0x080483ce in check_trace ()
(gdb)
0x080483cf in check_trace ()
(gdb)
0x080483e5 in main ()
(gdb) c
Continuing.
DEBUGER JE DETEKTIRAN
ANTI-DEBUG TEHNIKA

Program exited with code 0377.
(gdb)
```

Kao što je vidljivo iz primjera, prva označena linija predstavlja `stepi` naredbu koja izvršavanje programa izvodi instrukciju po instrukciju. Nakon provjere EFLAGS registra, program koji se nadzire otkrio je prisutnost TF zastavice i prekida izvođenje.

3.4. Checksum funkcija

Prilikom analize izvršavanja programa *debuger* alatom, obično se u program stavljaju prekidne točke kako bi se izvršavanje programa privremeno zaustavilo na određenoj adresi ili pri pozivu neke funkcije. Prekidna točke se postavlja tako da se na željenu adresu ubaci već prije spomenuta `int3` instrukcija, koja uzrokuje `SIGTRAP` signal i privremeno prekida izvođenje programa. Pomoću *checksum* funkcija moguće je vrlo jednostavno otkriti prisutnost `int3` instrukcija unutar programskog koda. U nastavku je prikazana jednostavna *checksum* funkcija koja računa *checksum* `main()` funkcije i uspoređuje ga s ranije postavljenom vrijednošću. Ukoliko je novo izračunata vrijednost drugačija, znači da je došlo do izmjena unutar programskog koda što može upućivati na ubacivanje `int3` instrukcija. U nastavku je priložen jednostavan primjer koji demonstrira objašnjenu tehniku.

`checksum.c`

```
int checkit (unsigned long *addr)
{
        unsigned long xor=0,*dome;
        char *find;

        find = (char*)addr;
        while (strncmp ("\xc9\xc3",find,2) != 0) find++;
         // find leave/ret opcodes
```

```
        find +=2;
        dome = addr;
        while (dome < (unsigned long*)find)  {
               xor ^= *(unsigned long*)dome;
               dome++;

        }
        if (xor != 0x124f268)
        {
               printf ("WRONG CHECKSUM!!!\n");
               exit(-1);
        }
        else printf ("XOR value correct: 0x%x\n",xor);
        return 0;
}

main ()
{

        checkit((unsigned long*)main);
        printf ("Simple code to test checksum protection!!!\n");
        __asm ("inc %eax\n");
        printf ("DONE!!!\n");
}
```

Program uzima početnu adresu `main()` funkcije i traži epilog funkcije nakon čega pomoću isključivo-ILI (XOR) operatora računa *checksum* `main()` funkcije u tom rasponu. Ukoliko se izračunata vrijednost ne poklapa sa 0x124f268 znači da je došlo do izmjena unutar programskog koda `main()` funkcije i prekida se izvođenje programa. U nastavku je prikazano prevođenje i pokretanje programa.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gcc checksum.c -o checksum
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# ./checksum
XOR value correct: 0x124f268
Simple code to test checksum protection!!!
DONE!!!
```

Kao što je vidljivo iz primjera, *checksum* vrijednost se poklapa što znači da je sadržaj `main()` funkcije nepromijenjen. U nastavku je prikazano pokretanje programa unutar `gdb` *debuger*a nakon postavljanja prekidne točke.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# gdb ./checksum
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

Program exited with code 010.
(gdb) break *main
Breakpoint 1 at 0x804848e
(gdb) r
Starting program: /root/PROJECTS/ELF/ANTIDEBUG/checksum

Breakpoint 1, 0x0804848e in main ()
(gdb) c
Continuing.
WRONG CHECKSUM!!!

Program exited with code 0377.
(gdb)
```

Prva označena linija predstavlja postavljanje prekidne točke, odnosno ubacivanje `int3` instrukcije na početak `main()` funkcije. Nakon što se program pokrene, izvršavanje se prekida i `gdb` program javlja da je došao do prekidne točke. Naredbom 'c' (engl. *continue*) izvršavanje se nastavlja i program kojeg se nadzire javlja da je *checksum* vrijednost pogrešna.

3.5. Lažne jmp instrukcije

Prilikom analize programskih odnosno asemblerskih instrukcija programa, vrlo je važno da se instrukcije točno interpretiraju. Neke *disasembler* programe je moguće zavarati na način da pogrešno interpretiraju asemblerske instrukcije programa. Problem se najčešće javlja u linearnim *disasemblerima,* koji asemblerske instrukcije interpretiraju jednu za drugom, bez ugrađene heuristike koja bi analizom logike programa uvidjela da se radi o tzv. *junk* instrukcijama koje se nikad ne izvršavaju. Ranije spomenuti `gdb` *debuger* alat također ima ugrađeni linearni *disasembler* koji se može zavarati *junk* instrukcijama. Kako se takve *junk* instrukcije ne bi izvršavale prilikom pokretanja programa, one se preskaču `jmp` instrukcijom. U nastavku je priložena implementacija objašnjene tehnike.

fake-jump.asm

```
Section .text
global _start
_start:

jmp fake
db 0xe9
fake:

inc eax
inc ebx
mov eax, edx

jmp fake2
db 0xe9
fake2:

mov eax,1
int 0x80
```

Označene linije predstavljaju lažne instrukcije koje linearni *disasembler* pogrešno interpretira i ispisuje pogrešni asemblerski kod. Kod `0xe9` označava `jmp` instrukciju, a sljedeća četiri okteta interpretiraju se kao memorijska adresa na koju se skače. U nastavku je prikazano prevođenje i *disasembliranje* `fake-jump.asm` programa `gdb` *debugerom*.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# nasm -f aout fake-jump.asm
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# ld fake-jump.o -o fake-jump

GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) disass _start
Dump of assembler code for function _start:
0x08048078 <_start+0>:  jmp    0x804807e <_start+6>
0x0804807d <_start+5>:  jmp    0xd88dc3c2
0x08048082 <_start+10>: jmp    0x8048088 <_start+16>
0x08048087 <_start+15>: jmp    0x8048244
0x0804808c <_start+20>: add    %cl,%ch
0x0804808e <_start+22>: .byte 0x80
0x0804808f <_start+23>: nop
End of assembler dump.
(gdb)
```

Kao što je vidljivo iz primjera, `gdb` program ispisuje potpuno krive asemblerske instrukcije. Ista situacija je i sa `objdump` alatom.

```
root@t-rex:~/PROJECTS/ELF/ANTIDEBUG# objdump -d ./fake-jump

./fake-jump:     file format elf32-i386

Disassembly of section .text:

08048078 <_start>:
```

```
 8048078:    e9 01 00 00 00          jmp    804807e <_start+0x6>
 804807d:    e9 40 43 89 d0          jmp    d88dc3c2 <__bss_start+0xd0893332>
 8048082:    e9 01 00 00 00          jmp    8048088 <_start+0x10>
 8048087:    e9 b8 01 00 00          jmp    8048244 <_start+0x1cc>
 804808c:    00 cd                   add    %cl,%ch
 804808e:    80                      .byte 0x80
 804808f:    90                      nop
```

3.6. Ranjivosti alata za analizu

Kao što je spomenuto na početku dokumenta, uspješnost analize izvršnih datoteka ovisi o alatima koji se koriste za njihovu analizu. Tijekom razvoja SEO alata za jednostavnu obfuskaciju ELF datoteka, priloženog u dodatku A, otkriveno je nekoliko ranjivosti u standardnim Linux alatima za analizu ELF datoteka. Konkretno, ranjivosti su pronađene u `libbfd` biblioteci koja se koristi za interpretaciju strukture sadržaja ELF datoteka. Problemi se javljaju ukoliko ELF datoteka ima pogrešne vrijednosti unutar tabele sekcijskih zaglavlja. U nekim slučajevima alati koji koriste `libbfd` biblioteku ne mogu otvoriti ELF datoteku sa neodgovarajućom tabelom sekcijskih zaglavlja, a ponekad i nasilno prekinu izvršavanje zbog SIGSEGV (*Segmentation fault*) signala. U nastavku je priloženo nekoliko primjera u kojima su demonstrirane ranjivosti gdb i `objdump` alata.

```
root@t-rex:~/PROJECTS/ELF# gdb ./gdbfallen.new
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...BFD: BFD 2.15.93 20041018
assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1783
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1783
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1783
Segmentation fault
root@t-rex:~/PROJECTS/ELF#

root@t-rex:~/PROJECTS/ELF# objdump -h ./gdbfallen.new
BFD: BFD 2.15.92.0.2 20040927 assertion fail elf.c:1781
BFD: BFD 2.15.92.0.2 20040927 assertion fail elf.c:1781
BFD: BFD 2.15.92.0.2 20040927 assertion fail elf.c:1782
BFD: BFD 2.15.92.0.2 20040927 assertion fail elf.c:1781
BFD: BFD 2.15.92.0.2 20040927 assertion fail elf.c:1782
BFD: BFD 2.15.92.0.2 20040927 assertion fail elf.c:1781
BFD: BFD 2.15.92.0.2 20040927 assertion fail elf.c:1782
Segmentation fault
root@t-rex:~/PROJECTS/ELF#
```

Označane linije predstavljaju nasilni prekid izvršavanja prikazanih alata. U nastavku je detaljnije analizirana ranjivost gdb alata (zanimljiva je činjenica da se u ovom slučaju ranjivost gdb alata analizira upravo samim gdb alatom).

```
root@t-rex:~/PROJECTS/ELF# gdb gdb
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...(no debugging symbols
found)
Using host libthread_db library "/lib/libthread_db.so.1".

(gdb) r ./gdbfallen.new
Starting program: /usr/bin/gdb ./gdbfallen.new
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
(no debugging symbols found)
```

```
(no debugging symbols found)
(no debugging symbols found)
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...BFD: BFD 2.15.93 20041018
assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1783
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1783
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1782
BFD: BFD 2.15.93 20041018 assertion fail elf.c:1783

Program received signal SIGSEGV, Segmentation fault.
0x0819f4ea in bfd_section_from_shdr ()
(gdb)
```

Označena linija predstavlja lokaciju identificirane ranjivosti. U nastavku je prikazan drugi slučaj u kojem gdb također nije u stanju interpretirati ELF tabelu sekcijskih zaglavlja.

```
root@t-rex:~/PROJECTS/ELF# gdb ./KILLGDBENCRYPTED
GNU gdb 6.3
Copyright 2004 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 "i486-slackware-linux"...BFD: BFD 2.15.93 20041018
internal error, aborting at elfcode.h line 190 in bfd_elf32_swap_symbol_in

BFD: Please report this bug.
```

Ovo nisu jedine ranjivosti gdb i objdump alata prilikom interpretiranja lažnih tabela sekcijskih zaglavlja. U određenim situacijama moguće je npr. u potpunosti onesposobiti terminal unutar kojeg se pokreće objdump program.

4. Maskiranje ELF datoteke

Iako prije objašnjene *anti-debug* tehnike pomažu u zaštiti izvršnih ELF datoteka, one samo usporavaju analizu programskog koda. Za napredniji oblik zaštite potrebno je na neki način kriptirati sadržaj ELF datoteke. Za demonstraciju takve vrste zaštite razvijen je program pod nazivom SEO (engl. ***S**imple **E**LF **O**bfuscation*), koji je priložen na kraju dokumenta (DODATAK A). Program SEO kombinacijom lažiranja tabele sekcijskih zaglavlja i kriptiranjem text sekcije, odnosno segmenta u kojem se nalaze programske instrukcije, štiti ELF datoteku od jednostavne analize. SEO program također uništava sekcije koje sadrže dodatne informacije za analizu programa, kao i sekciju koja sadrži verziju prevoditelja. Tabela sekcijskih zaglavlja lažira se tako da se sva imena sekcija zamijene, pa se npr. za text sekciju ispisuje rodata, a za bss sekciju ctors. Za dodatno lažiranje ELF tabele sekcijskih zaglavlja, potrebno je ukloniti komentare iz SEO.c programa iza linije "uncomment this to confuse debuggers", što rezultira dodatnim uništavanjem informacija iz tabele sekcijskih zaglavlja. Takve postavke u određenim situacijama mogu proizvesti nelegitimnu ELF datoteku koju objdump i gdb alati nisu u stanju otvoriti. Sama enkripcija prilično je jednostavna i svodi se na XOR operaciju između programskog koda i lozinke odnosno ključa koji je veličine 20 okteta. Za dekripciju programskog koda u ELF datoteci uništava se prije spomenuti NOTE segment i postavlja mu se tip PT_LOAD, što znači da će prilikom pokretanja programa biti mapiran u memoriju. Sadržaj tog segmenta dodaje se na kraj ELF datoteke, pa se kompletno mijenja i struktura programskog zaglavlja koja ga opisuje. Sadržaj novog PT_LOAD segmenta je programski kod koji čita 20 okteta odnosno ključ sa standardnog ulaza (STDIN), i pomoću njega XOR operacijom dekriptira programske instrukcije text segmenta ELF datoteke. Ulazna točka ELF datoteke se mijenja i postavlja se na novi PT_LOAD segment koji dekriptira text segment. Nakon što je program dekriptiran, izvršavanje se preusmjerava na programske instrukcije u text segmentu. Ukoliko je ključ pogrešan, text segment će se pogrešno dekriptirati i prilikom pokušaja izvršavanja instrukcija iz text segmenta, program će

nasilno prekinuti s izvršavanjem. U nastavku je prikazan program koji demonstrira zaštitu pomoću SEO.C programa.

seo-test.c

```
main ()
{
        int a=100;
        int b=200;
        printf ("TEST SEO ZASTITE\n");
        printf ("REZULTAT: a(100) + b(200) = %d\n",a+b);
        exit(0);
}
```

Prevođenje i zaštita seo-test programa je prikazana u nastavku.

```
root@t-rex:~/PROJECTS/ELF# gcc seo-test.c -o seo-test -static
root@t-rex:~/PROJECTS/ELF# gcc seo.c -o seo
root@t-rex:~/PROJECTS/ELF# ./seo ./seo-test
[----------------------------------------]
[           SIMPLE ELF OBFUSCATION       ]
[----------------------------------------]
[ Coded by Leon Juranic <ljuranic@lss.hr> ]
[  LSS Security / http://security.lss.hr/ ]
[----------------------------------------]
o File size: 482701
o ENTRY POINT: 0x8048100
-----------------------------------------
Phoff: 34(size:80), Shoff: 6a5fc: size 488

o Searching for segments...
-----------------------------------------
-> SEGMENT 0: vaddr: 0x8048000
o TEXT SEGMENT FOUND, ENCRYPT IT!!!
Password: certtest
filesz: 420692
memsize: 420692
-> file offset entry: 0x100
-----------------------------------------
-> SEGMENT 1: vaddr: 0x80af000
-----------------------------------------
-> SEGMENT 2: vaddr: 0x80480b4
o NOTE SEGMENT FOUND, ADJUST IT FOR DECRYPT LOOP!!!
-> vaddr: 0x80480b4
-> p_offset: 0x75d8d
o NEW SEGMENT
-> vaddr: 0x80f5d90
->p_offset: 0x75d90
-----------------------------------------
-> SEGMENT 3: vaddr: 0x0
-----------------------------------------
o Read sections and clean some of them
o Scrambling section names
o Obfuscate sections
-----------------------------------------

o DONE!!!
```

Označena linija predstavlja ključ sa kojim je kriptiran text segment ELF datoteke. Nakon završetka obrade seo-test programa, SEO kreira novu datoteku seo-test.new koja predstavlja zaštićenu ELF datoteku. U nastavku je prikazan ispis tabele programskih zaglavlja seo-test i seo-test.new ELF datoteka.

root@t-rex:~/PROJECTS/ELF# readelf -l ./seo-test

```
Elf file type is EXEC (Executable file)
Entry point 0x8048100
There are 4 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x66b54 0x66b54 R E 0x1000
  LOAD           0x067000 0x080af000 0x080af000 0x00cf8 0x01a08 RW  0x1000
  NOTE           0x0000b4 0x080480b4 0x080480b4 0x00020 0x00020 R   0x4
```

```
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00        .init  .text  __libc_freeres_fn  .fini  .rodata  __libc_subfreeres
__libc_atexit .eh_frame .note.ABI-tag
   01         .ctors  .dtors  .jcr  .data.rel.ro  .got  .got.plt  .data  .bss
__libc_freeres_ptrs
   02     .note.ABI-tag
   03


root@t-rex:~/PROJECTS/ELF# readelf -l ./seo-test.new

Elf file type is EXEC (Executable file)
Entry point 0x80f5d90
There are 4 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x66b54 0x66b54 RWE 0x1000
  LOAD           0x067000 0x080af000 0x080af000 0x00cf8 0x01a08 RW  0x1000
  LOAD           0x075d90 0x080f5d90 0x080480b4 0x01000 0x01000 RWE 0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00           __libc_subfreeres   .debug_str   .debug_aranges   .note.ABI-tag
__libc_freeres_fn .rodata .fini .dtors .init
   01     .debug_line .got.plt .got .data .eh_frame .jcr __libc_atexit .ctors
.text
   02
   03
```

Kao što je vidljivo iz primjera, `seo-test.new` više nema NOTE segmenta, a ulazna točka je postavljena u novi LOAD segment. Imena sekcija i segmenti na koje se one odnose su također izmiješani. U nastavku je prikazan početak programskog koda (`_start` simbol) unutar `seo-test` i `seo-test.new` programa prilikom unosa pogrešne lozinke.

```
root@t-rex:~/PROJECTS/ELF# gdb ./seo-test
....
 (gdb) disass _start
Dump of assembler code for function _start:
0x08048100 <_start+0>:  xor    %ebp,%ebp
0x08048102 <_start+2>:  pop    %esi
0x08048103 <_start+3>:  mov    %esp,%ecx
0x08048105 <_start+5>:  and    $0xfffffff0,%esp
0x08048108 <_start+8>:  push   %eax
0x08048109 <_start+9>:  push   %esp
0x0804810a <_start+10>: push   %edx
0x0804810b <_start+11>: push   $0x80484c0
0x08048110 <_start+16>: push   $0x8048460
0x08048115 <_start+21>: push   %ecx
0x08048116 <_start+22>: push   %esi
0x08048117 <_start+23>: push   $0x8048214
0x0804811c <_start+28>: call   0x8048270 <__libc_start_main>
0x08048121 <_start+33>: hlt
0x08048122 <_start+34>: nop
0x08048123 <_start+35>: nop
End of assembler dump.

root@t-rex:~/PROJECTS/ELF# gdb ./seo-test.new
....
(gdb) r
Starting program: /root/PROJECTS/ELF/seo-test.new
warning: shared library handler failed to enable breakpoint
password:testtesttesttest

Program received signal SIGILL, Illegal instruction.
0x08048104 in ?? ()
(gdb) x/i 0x08048100
0x8048100:      sub    %dh,%bh
(gdb)
```

```
0x8048102:      inc    %edx
(gdb)
0x8048103:      xchg   %eax,%edi
(gdb)
0x8048104:      lock xchg %eax,%ebp
(gdb)
0x8048106:      imul   %cl
(gdb)
0x8048108:      dec    %ebx
(gdb)
0x8048109:      inc    %ecx
(gdb)
0x804810a:      push   %ebx
(gdb)
0x804810b:      jae    0x80480e3
(gdb)
0x804810d:      orb    $0x98,0xced011d
(gdb)
```

Pri pokretanju `seo-test.new` programa, upisana je pogrešna lozinka i `text` segment je pogrešno dekriptiran, što rezultira nasilnim prekidom izvršavanja programa uz `SIGILL` signal. Važno je napomenuti da je SEO zaštita namijenjena samo za statički prevedene ELF datoteke.

5. Zaključak

Zaštita izvršnih ELF datoteka opširno je područje i u budućnosti će zasigurno donijeti mnogo novih tehnika i alata. U dokumentu je dan pregled nekih osnovnih principa zaštite ELF datoteka, dok se u praksi mogu pronaći mnogo kvalitetniji primjeri kao što su već prije spomenuti `burneye` i `Shiva` alati. Nažalost, razvojem naprednih alata za zaštitu ELF datoteka i forenzička analiza malicioznih programa postaje mnogo teža, što je otežavajuća okolnost za sigurnosne stručnjake koji se bave ovim područjem.

6. Reference

[1] Silvio Cesare, "Linux anti-debugging techniques", http://vx.netlux.org/lib/vsc04.html
[2] cut and grugq, "Armouring the ELF", http://www.phrack.org/show.php?p=58&a=5
[3] Andrew Griffiths, "Binary protection schemes", http://www.felinemenace.org
[4] Tool Interface Standards (TIS), "Executable and Linkable Format (ELF)"
[5] Binary File Descriptor Library, http://www.skyfree.org/linux/references/bfd.pdf

DODATAK A

SEO.c

```
/*
        SIMPLE ELF OBFUSCATION
        ----------------------

        This program will obfuscate *static* ELF file so it is impossible
        to execute it without correct password. ELF Section table is also
        scrambled, so it won't help to attacker who tries to reverse
        engineer that binary. Encryption itself is very simple and it
        is based on XOR operator. This isn't meant to be strong obfuscation
        or encryption, so it is possible to break protection with some effort.
        However, this protection will keep alot of "security" people away
        from executing that binary. Since this is more or less PoC, no
        anti-debugging techniques are implemented in it.

        Coded by Leon Juranic <ljuranic@lss.hr>
        LSS Security - http://security.lss.hr/

*/



#include <stdio.h>
#include <unistd.h>
#include <elf.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>

#define PAGE_SIZE 4096
#define LOAD_ADDR 0x08048000


char shellcode[]= // decrypt loop
{
0x60,0x9c,0xe9,0x80,0x00,0x00,0x00,0x5e,
0xb8,0x04,0x00,0x00,0x00,0xbb,0x01,0x00,
0x00,0x00,0x89,0xf1,0xba,0x09,0x00,0x00,
0x00,0xcd,0x80,0xe9,0x76,0x00,0x00,0x00,
0x5e,0xb8,0x03,0x00,0x00,0x00,0xbb,0x00,
0x00,0x00,0x00,0x89,0xf1,0xba,0x14,0x00,
0x00,0x00,0xcd,0x80,0xbe,0x44,0x43,0x42,
0x41,0xb9,0x48,0x47,0x46,0x45,0x8b,0x7c,
0x24,0xfc,0xc1,0xf9,0x02,0xba,0x00,0x00,
0x00,0x00,0x8b,0x06,0x8b,0x1f,0x3d,0xef,
0xbe,0xad,0xde,0x75,0x05,0xbb,0xef,0xbe,
0xad,0xde,0x31,0xd8,0x89,0x06,0x81,0xc6,
0x04,0x00,0x00,0x00,0x81,0xc7,0x04,0x00,
0x00,0x00,0x42,0x81,0xfa,0x05,0x00,0x00,
0x00,0x75,0x09,0xba,0x00,0x00,0x00,0x00,
0x8b,0x7c,0x24,0xfc,0xe2,0xcc,0x9d,0x61,
0xb8,0x63,0x62,0x61,0x60,0xff,0xe0,0xe8,
0x7b,0xff,0xff,0xff,0x70,0x61,0x73,0x73,
0x77,0x6f,0x72,0x64,0x3a,0x00,0xe8,0x85,
0xff,0xff,0xff,0x41,0xd8,0xe8,0x06,0x81,
0xc6,0x04,0x00,0x00,0x00,0x84,0x17,0x14,
0x00,0x10,0x00,0x22,0xe8,0xfa,0xe8,0x90
};



char buf[24];

void crypt (char *addr,int size)
{
        unsigned long x=0,y=0,i=0,j=0,z=0;
```

```
        for (i=0,j=0;j<size;i+=4,j+=4)
        {
                if (i==20) i=0;

                x = *(long*)(addr+j);
                y = *(long*)(buf+i);
                z = x ^ y;
                if (x == 0)
                        z = 0xdeadbeef;

                *(unsigned long*)&addr[j] = z;
        }


}

void banner ()
{
        printf ("[-----------------------------------------]\n"
                "[           SIMPLE ELF OBFUSCATION        ]\n"
                "[-----------------------------------------]\n"
                "[ Coded by Leon Juranic <ljuranic@lss.hr> ]\n"
                "[  LSS Security / http://security.lss.hr/ ]\n"
                "[-----------------------------------------]\n");

}

main (int argc, char **argv)
{

        Elf32_Ehdr *ehdr;
        Elf32_Phdr *phdr;
        Elf32_Shdr *shdr;
        char *ph,*sh,*out,*tmp;
        int fd,fd2,i,plen,stroff=0,strtaboff=0,strtablen,tlen,inc=0;
        struct stat st;
        char *strtab;
        unsigned long *shtname;
        struct timeval tv;
        long tmpval=0,realent,savefiletextoff;


        banner();
        if (argc < 2)
        {
                printf ("Usage: %s <file_to_protect>\n",argv[0]);
                exit(-1);
        }

        fd = open (argv[1],O_RDONLY);
        fd2 = open (strcat (argv[1],".new"),O_WRONLY|O_CREAT|O_TRUNC,00777);

        fstat (fd, &st);
        printf ("o File size: %d\n",st.st_size);

        out  = (char*) malloc (st.st_size+PAGE_SIZE);
        memset (out,0,st.st_size);
        tmp  = (char*) malloc (st.st_size);

        read (fd, tmp, st.st_size);

        if (memcmp (tmp,"\x7f\x45\x4c\x46", 4) != 0) // check if that is ELF
        {
                printf ("o ERROR: Not an ELF file!!!\n");
                exit(-1);
        }

        ehdr = (struct Elf32_Ehdr*)tmp;
        ph = tmp + ehdr->e_phoff;
        sh = tmp + ehdr->e_shoff;
```

```
	shtname = (unsigned long*)calloc (ehdr->e_shnum,sizeof(unsigned long));    // for section string
table scramble

	printf ("o ENTRY POINT: 0x%x\n",ehdr->e_entry);

	// fix decode function
	*(long*)&shellcode[129] = ehdr->e_entry;  // fix e_entry

	printf ("-----------------------------------------\n");
	printf  ("Phoff:  %x(size:%x),  Shoff:  %x:  size  %x\n\n",ehdr->e_phoff,ehdr->e_phentsize  *  ehdr-
>e_phnum, ehdr->e_shoff,ehdr->e_shentsize * ehdr->e_shnum);

	phdr = (struct Elf32_Phdr*)ph;

	printf ("o Searching for segments...\n");
	for (i=0; i < ehdr->e_phnum; i++)
	{


		printf ("-----------------------------------------\n");
		printf ("-> SEGMENT %d: vaddr: 0x%x\n",i,phdr->p_vaddr);
		if (phdr->p_type == PT_LOAD && phdr->p_offset == 0) // text segment
		{

			printf ("o TEXT SEGMENT FOUND, ENCRYPT IT!!!\n");
			printf ("Password: ");
			fgets(buf,21,stdin);
			if (strlen(buf) < 20) {
				printf ("ERROR: Password must be 20 bytes long!!!\n");
				exit(-1);
			}
			phdr->p_flags = phdr->p_flags + PF_W;
			printf ("filesz: %d\nmemsize: %d\n",phdr->p_filesz,phdr->p_memsz);


			savefiletextoff=ehdr->e_entry - LOAD_ADDR;
			printf ("-> file offset entry: 0x%x\n",savefiletextoff);
			crypt (tmp+savefiletextoff,phdr->p_memsz);
			*(long*)&shellcode[53] = ehdr->e_entry;        // text section address
			*(long*)&shellcode[58] = phdr->p_memsz-savefiletextoff; // text section size
		}


		if (phdr->p_type == PT_NOTE) {
			printf ("o NOTE SEGMENT FOUND, ADJUST IT FOR DECRYPT LOOP!!!\n");
			phdr->p_type = PT_LOAD;
			phdr->p_flags += PF_W + PF_X;

			phdr->p_offset = st.st_size;

			printf ("-> vaddr: 0x%x\n-> p_offset: 0x%x\n", phdr->p_vaddr,phdr->p_offset);
			for (inc=0;inc<20;inc++)
			{
				if (phdr->p_offset % 4 != 0)
				 ++(phdr->p_offset);
				else break;
			}
			tmpval = 0xffffffff + phdr->p_offset;

			phdr->p_vaddr = ((LOAD_ADDR + st.st_size + 0x10000) & 0xffff0000) | tmpval;
			phdr->p_vaddr++;

			ehdr->e_entry = phdr->p_vaddr; // LOAD_ADDR + st.st_size + inc;
			printf ("o NEW SEGMENT\n");
			printf ("-> vaddr: 0x%x\n->p_offset: 0x%x\n", phdr->p_vaddr,phdr->p_offset);
			phdr->p_filesz = PAGE_SIZE;
			phdr->p_memsz = PAGE_SIZE;

		}
		phdr++;
	}
```

```
	printf ("-----------------------------------------\n");
	shdr = (struct Elf32_Shdr*)sh;
	printf ("o Read sections and clean some of them\n");
	for (i=0 ; i < ehdr->e_shnum; i++)
	{

		if (((shdr->sh_type == SHT_STRTAB) && (shdr->sh_flags == 0)) && stroff == 0) {
			stroff = shdr->sh_offset;

		}
		if (((shdr->sh_type == SHT_STRTAB) && (shdr->sh_flags == SHF_ALLOC)))
		{
			strtaboff = shdr->sh_offset;
			strtablen = shdr->sh_size;
			memset (tmp+strtaboff,0,strtablen);    // clean strtab
		}
		shtname[i]  = shdr->sh_name;
		shdr++;
	}

	strtab = tmp + stroff; // for section string names

	printf ("o Scrambling section names\n");
	for (i=0;i<100;i++) {
		int a,b,c,d;
		gettimeofday(&tv,NULL);
		srand (tv.tv_usec);
		a = rand() % (ehdr->e_shnum-1);
		b = rand() % (ehdr->e_shnum-1);
		c = shtname[a];

		if (a == 0 || b == 0)
			continue;
		shtname[a] = shtname[b];   // change section names
		shtname[b] = c;

	}



	printf ("o Obfuscate sections\n");
	shdr = (struct Elf32_Shdr*)sh;
	 for (i=0 ; i < ehdr->e_shnum; i++)
	 {

		 if (strstr (strtab+shdr->sh_name,"comment") != NULL)
			 memset (tmp+shdr->sh_offset,0,shdr->sh_size);

		 if (strstr (strtab+shdr->sh_name,"debug") != NULL)
			 memset (tmp+shdr->sh_offset,0,shdr->sh_size);

		 shdr->sh_name  = shtname[i];
		shdr->sh_type = 1;

	//       uncomment this to confuse debuggers
	 //      shdr->sh_type = rand() %  10 ;
	 //      shdr->sh_addr  = (LOAD_ADDR-15000) + (rand() % 50000);
	 //      shdr->sh_size  = rand() % 65535;
		 shdr++;
	 }
	 memcpy (out,tmp,st.st_size);

	 memcpy (out + st.st_size + inc,shellcode,sizeof(shellcode));

	printf ("-----------------------------------------\n\n");
	printf ("o DONE!!!\n\n");


	write (fd2,out,st.st_size+PAGE_SIZE);

	free (out);
	free (tmp);
	free (shtname);
```

```
        close (fd);
        close (fd2);


}
```