

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Đorđe Todorović

**PODRŠKA ZA NAPREDNU ANALIZU
PROMENLJIVIH LOKALNIH ZA NITI
POMOĆU ALATA GNU GDB**

master rad

Beograd, 2018.

Mentor:

dr Mika MIKIĆ, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana ANIĆ, vanredni profesor
University of Disneyland, Nedodija

dr Laza LAZIĆ, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami, tati i dedi

Sadržaj

1	Uvod	1
2	Kako rade debageri?	2
2.1	Prevođenje programa	3
2.2	Fajl format DWARF	3
2.3	Fajl format ELF	7
2.4	Sistemska poziv <code>ptrace</code>	8
2.5	Realizacija osnovnih elemenata upotrebe debagera	9
3	GNU GDB alat	13
4	Datoteke jezgra	14
5	TLS	15
5.1	Motivacija	15
5.2	Rukovanje TLS-om tokom izvršavanja programa	15
5.3	Arhitekturno specifične zavisnosti	15
5.4	TLS Modeli pristupa	15
6	Implementacija rešenja	16
7	Zaključak	17
	Literatura	18

Glava 1

Uvod

Glava 2

Kako rade debageri?

Greške su sastavni deo svakog rada koji obavlja čovek, te ih i programeri prave. Greške mogu biti hardverske i softverske. One mogu imati razne poslednice. Neke su manje važne, kao npr. korisnički interfejs aplikacije ima neočekivanu boju pozadine. Postoje i greške koje mogu imati daleko veće posledice, pa čak i ugroziti živote drugih, kao npr. greške u softveru ili hardveru uređaja i aplikacija avio industrije. Faza testiranja je veoma važna u ciklusu razvoja softvera. Nakon faze testiranja obično sledi faza analize i otklanjanja grešaka.

Debager (eng. *debugger*) je softverski alat koji koriste programeri za testiranje, analizu i otklanjanje grešaka u programima. Sam proces korišćenja takvih alata nazivamo debugovanjem (eng. *debugging*). Debageri mogu pokrenuti rad nekog procesa ili se „nakačiti” na proces koji je već u fazi rada. U oba slučaja, debager preuzima kontrolu nad procesom. To mu omogućava da izvršava proces instrukciju po instrukciju, do postavlja tačke prekida (eng. *breakpoints*) itd. Proces izvršavanja programa od strane debagera sekvencijalno, instrukciju po instrukciju ili liniju po liniju, nazivamo koraćanje. Tačke prekida su mogućnost debagera da zaustavi izvršavanje programa na određenoj tački. To može biti trenutak kada program izvrši određenu funkciju, liniju koda itd. Neki debageri imaju mogućnost izvršavanja funkcija programa koji se debuguje, uz ograničenje da program pripada istoj procesorskoj arhitekturi kao i domaćinski sistem na kojem se debager izvršava. Čak i struktura programa može biti promenjena, prateći proratne efekte.

Podršku debagerima, u opštem slučaju, daju operativni sistemi, kroz systemske pozive koji omogućavaju tim alatima da pokrenu i preuzmu kontrolu nad nekim drugim procesom. Za neke naprednije tehnike debugovanja poželjna je podrška od strane hardvera. U radu će detaljno biti obrađen rad UNIX-olikih, posebno Linux

debagera. Windows debageri i programski prevodioci ne prate standard DWARF [2] prilikom baratanja sa debug informacijama namenjene za taj operativni sistem. Alati za debugovanje u okviru operativnog sistema Windows koriste standard *Microsoft CodeView*. Više informacija o ovom standardu može se pronaći u literaturi [5].

2.1 Prevođenje programa

Programi koji se debuguju se prevode uz pomoć odgovarajuće opcije programskih prevodioca (za prevodioce GCC i LLVM/Clang, to je opcija `-g`) koja obezbeđuje generisanje pomoćnih debug informacija. Ukoliko je program koji se analizira preveden bez optimizacija, debug informacije koje prate program su potpune. Programi koji se puštaju u produkciju, da bi bili brži i zauzimali manje memorije, se prevode uz pomoć optimizacija. Nivoi optimizacija produkcijskih programa su „-O2” i „-O3”. Prilikom optimizacija se gube razne debug informacije. Neke promenljive i funkcije programa neće biti predstavljene debug informacijama. Npr. promenljiva programa može biti živa samo u nekim određenim delovima programa, pa programski prevodioci generišu debug informacije o njenim lokacijama samo u tim određenim delovima koda. Prilikom optimizacija na nivou mašinskog koda život promenljive može biti skraćen, pa čitanje vrednosti promenljive iz debagera u nekim situacijama neće biti moguće, iako gledajući izvorni kod očekujemo da je ona živa u tom trenutku.

2.2 Fajl format DWARF

DWARF je debug fajl format koji se koristi od strane programskih prevodioca (kao npr. GCC ili LLVM/Clang) i debagera (kao npr. GNU GDB) da bi se omogućilo debugovanje na nivou izvornog koda. Omogućava podršku za razne programske jezike kao što su C/C++ i Fortran, ali je dizajniran tako da se lako može proširiti na ostale jezike. Arhitekturno je nezavisan i predstavlja „most” između izvornog koda i izvršnog fajla. Trenutno je poslednji realizovani standard verzija 5 formata DWARF.

Debug fajl format DWARF se odnosi na Unix-olike operativne sisteme, kao što su Linux i MacOS. Generisane debug informacije, prateći DWARF standard, su podeljene u nekoliko sekcija sa prefiksom `.debug_`. Neke od njih su `.debug_line`, `.debug_loc` i `.debug_info` koje redom predstavljaju informacije o linijama izvor-

nog koda, lokacijama promenljivih i ključna debug sekcija koja sadrži debug informacije koje referišu na informacije iz ostalih debug sekcija. DWARF je predstavljen kao drvolika struktura, smeštena u `.debug_info` sekciju, koja razne entitete programskog jezika opisuje osnovnom debug jedinicom DIE (eng. Debug Info Entry). Osnovna debug jedinica može opisivati lokalnu promenljivu programa, formalni parametar, funkciju itd. Svaka od njih je identifikovana DWARF tagom koji predstavlja informaciju o toj jedinici, gde je npr. tag za lokalne promenljive predstavljen sa `DW_TAG_local_variable`, ili tag za funkciju je obeležen sa `DW_TAG_subprogram`. Svaka debug jedinica je opisana određenim DWARF atributima sa prefiksom `DW_AT_`. Oni mogu ukazivati na razne informacije o entitetu kao sto su ime promenljive ili funkcije, liniju deklaracije, itd. Koren svakog DWARF stabla je predstavljen debug jedinicom, sa tagom `DW_TAG_compile_unit`, koja predstavlja kompilacionu jedinicu, tj. izvorni kod programa. Primer dela DWARF stabla za primer predstavljen slikom 2.1 je dat slikom 2.2.

```
#include <stdio.h>

int main()
{
    int x;
    x = 5;

    printf("The value is %d\n", x);

    return 0;
}
```

Slika 2.1: Izvorni kôd test primera koji se analizira.

Funkcija `main` primera sa slike 2.1 na slici 2.2 je predstavljena DWARF tagom `DW_TAG_subprogram`. Atribut te debug jedinice predstavljen sa `DW_AT_name` ima vrednost imena funkcije. Atributi `DW_AT_low_pc` i `DW_AT_high_pc` redom predstavljaju adresu prve mašinske instrukcije te funkcije u memoriji programa i ofset na kojem se nalazi poslednja mašinska instrukcija te funkcije. Sledeći čvor drveta predstavlja promenljivu `x` istog test primera. Ta debug jedinica je dete čvora koji predstavlja funkciju `main` i ukazuje da se promenljiva `x` nalazi unutar funkcije `main`. Promenljiva `x` je predstavljena DWARF tagom `DW_TAG_variable`. Atribut `DW_AT_name` predstavlja ime promenljive, `DW_AT_type` referiše na debug jedinicu


```

<1><73>: Abbrev Number: 4 (DW_TAG_subprogram)
  <74> DW_AT_external      : 1
  <74> DW_AT_name          : (indirect string, offset: 0xa6): main
  <78> DW_AT_decl_file     : 1
  <79> DW_AT_decl_line    : 3
  <7a> DW_AT_type          : <0x57>
  <7e> DW_AT_low_pc        : 0x400526
  <86> DW_AT_high_pc       : 0x2a
  <8e> DW_AT_frame_base    : 1 byte block: 9c          (DW_OP_call_frame_cfa)
  <90> DW_AT_GNU_all_tail_call_sites: 1
<2><90>: Abbrev Number: 5 (DW_TAG_variable)
  <91> DW_AT_name          : x
  <93> DW_AT_decl_file     : 1
  <94> DW_AT_decl_line    : 5
  <95> DW_AT_type          : <0x57>
  <99> DW_AT_location      : 2 byte block: 91 6c      (DW_OP_fbreg: -20)

```

Slika 2.2: Debug jedinice DWARF stabla primera sa slike 2.1.

koja predstavlja tip promenljive, dok `DW_AT_location` atribut predstavlja lokaciju promenljive u memoriji programa.

Debug promenljive

Svaka promenljiva programa prevedenog sa debug informacijama, ukoliko se ne radi o optimizovanom programu, je predstavljena DWARF tagom `DW_TAG_variable`. Atribut `DW_AT_location` ukazuje na lokaciju promenljive. Lokacija može biti predstavljena DWARF izrazom, kao npr. lokacija promenljive `x` na slici 2.2. DWARF izraz te promenljive ukazuje da se ona nalazi na ofsetu -20 trenutnog stek okvira `main` funkcije. U neoptimizovanom kodu sve promenljive imaju lokacije zadate DWARF izrazom. Njihove vrednosti su dostupne debagerima u bilo kom delu koda u kom su definisane.

U optimizovanom kodu lokacija promenljive može sadržati referencu na informaciju o lokaciji u `.debug_loc` sekciji. Lokacije u toj sekciji su predstavljene listama lokacija. Jedna promenljiva u optimizovanom kodu može biti smeštena na raznim memorijskim lokacijama ili registrima. Elementi liste opisuju lokacije promenljive na mestima u kodu gde je ona živa. Ukoliko promenljiva nije živa u nekom delu koda, programski prevodioci u optimizovanom kodu neće pratiti njenu lokaciju. Slika 2.3 predstavlja primer lokacije promenljive u optimizovanom kodu.

Lokacijska lista promenljive `x` je predstavljena na slici 2.4. U ovom konkretnom primeru, promenljiva živi samo na jednom mestu. Potencijalno je mogla imati još elemenata lokacijske liste. `Offset` predstavlja informaciju gde se lokacijska lista

```

<2><2cd>: Abbrev Number: 19 (DW_TAG_variable)
<2ce>   DW_AT_name      : x
<2d0>   DW_AT_decl_file  : 1
<2d1>   DW_AT_decl_line  : 5
<2d2>   DW_AT_type       : <0x5e>
<2d6>   DW_AT_location   : 0x0 (location list)

```

Slika 2.3: Debug jedinica promenljive `x` u optimizovanom kodu.

određene promenljive nalazi u `.debug_loc` sekciji. `Begin` i `End` predstavljaju informaciju od koje do koje adrese u programu važi data lokacija, tj. od koje do koje instrukcije je određena promenljiva živa. `Expression` predstavlja DWARF izraz koji opisuje lokaciju promenljive.

```

Contents of the .debug_loc section:

Offset  Begin          End          Expression
00000000 0000000000400568 0000000000400583 (DW_OP_fbreg: -28)
00000014 <End of list>

```

Slika 2.4: Lokacijska lista promenljive `x` u optimizovanom kodu.

2.3 Fajl format ELF

ELF (eng. *Executable and Linkable Format*) [1] je format izvršnih fajlova, deljenih biblioteka, objektnih fajlova i datoteka jezgara.

ELF sadrži razne informacije o samom fajlu. Podeljen je u dva dela: ELF zaglavlje i podaci fajla. ELF zaglavlje sadrži informacije o arhitekturi za koju je program preveden i definiše da li program koristi 32-bitni ili 64-bitni adresni prostor. Zaglavlje 32-bitnih programa je dužine 52 bajta, dok kod 64-bitnih programa zaglavlje je dužine 64 bajta. Podaci fajla mogu sadržati programsku tabelu zaglavlja (eng. *Program header table*), sekcijску tabelu zaglavlja (eng. *Section header table*) i ulazne tačke prethodne dve tabele. Slika 2.5 prikazuje primer prikaza fajla u formatu ELF.

```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                             UNIX - System V
  ABI Version:                         0
  Type:                                EXEC (Executable file)
  Machine:                             Advanced Micro Devices X86-64
  Version:                             0x1
  Entry point address:                 0x4005c0
  Start of program headers:            64 (bytes into file)
  Start of section headers:           7944 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:             56 (bytes)
  Number of program headers:           10
  Size of section headers:            64 (bytes)
  Number of section headers:           39
  Section header string table index: 36

Section Headers:
[Nr] Name                               Type           Address         Offset
    Size                               EntSize        Flags Link Info Align
[ 0]                                NULL            0000000000000000 00000000
    0000000000000000 0000000000000000 0 0 0
[ 1] .interp                          PROGBITS        0000000000400270 00000270
    0000000000000001c 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag                     NOTE            000000000040028c 0000028c
    00000000000000020 0000000000000000 A 0 0 4
[ 3] .note.gnu.build-id                 NOTE            00000000004002ac 000002ac
    00000000000000024 0000000000000000 A 0 0 4
[ 4] .gnu.hash                          GNU_HASH        00000000004002d0 000002d0
    0000000000000001c 0000000000000000 A 5 0 8
[ 5] .dynsym                           DYNSYM          00000000004002f0 000002f0
    000000000000000d8 0000000000000018 A 6 1 8

```

Slika 2.5: ELF fajl format počitan alatom *objdump*.

2.4 Sistemski poziv ptrace

Operativni sistem GNU Linux pruža sistemski poziv `ptrace` [3] koji debagerima omogućava rad. Ovaj sistemski poziv omogućava jednom procesu kontrolu nad izvršavanjem nekog drugog procesa i menjanje memorije i registara istog. Potpis ove funkcije je:

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

Prvi argument sistemskog poziva predstavlja informaciju kojom operativnom sistemu jedan proces, ne nužno debager, ukazuje na nameru preuzimanja kontrole drugog procesa. Ukoliko taj argument ima vrednost `PTRACE_TRACEME` to ukazuje na nameru praćenja (eng. *tracing*) određenig procesa, `PTRACE_PEEKDATA` i `PTRACE_POKEDATA` redom ukazuju na nameru čitanja i pisanja memorije, `PTRACE_GETREGS` i `PTRACE_SETREGS` se odnose na čitanje i pisanje registara. To su samo neki osnovni slučajevi korišćenja, za više informacija pogledati [3]. Drugi argument sistemskog poziva `pid` ukazuje na identifikacioni broj ciljanog procesa. Treći i četvrti argument se po potrebi koriste u zavisnosti od namere korišćenja sistemskog poziva `ptrace` za čitanje ili pisanje sa adrese datom trećim argumentom, pri tom baratajući podacima na adresi zadatoj četvrtim argumentom. To znači ukoliko se koristi `PTRACE_TRACEME` poslednja tri argumenta sistemskog poziva se ignorišu. Ukoliko se koristi `PTRACE_GETREGS` sa adrese `addr` se čita jedna reč iz memorije. Navodimo par osnovnih primera korišćenja `ptrace` sistemskog poziva, a u nastavku će biti navedeno još primera.

Primer 1 Program inicira da će biti praćen od strane roditeljskog procesa:

```
ptrace(PTRACE_TRACEME, 0, NULL, NULL);
```

Primer 2 Čitanje vrednosti registara procesa sa identifikatorom 8845 i upisivanje tih vrednosti na adresu promenljive `regs`:

```
ptrace(PTRACE_GETREGS, 8845, NULL, &regs);
```

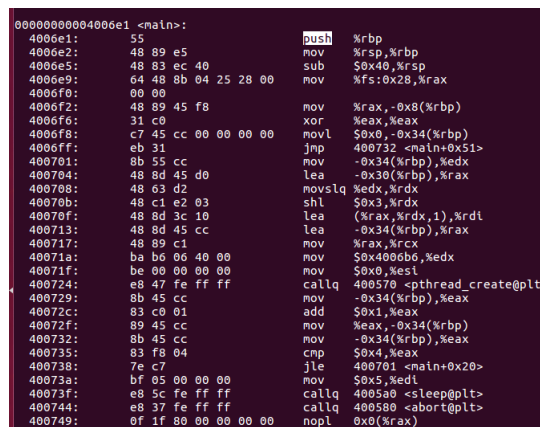
2.5 Realizacija osnovnih elemenata upotrebe debagera

Tačke prekida

Postoje dve vrste tačaka prekida (eng. *breakpoints*): softverske i hardverske [4].

Osvrnimo se prvo na softverske tačke prekida. Postavljanje tačaka prekida predstavlja jednu od najkorišćenijih mogućnosti debagera, te stoga navedimo par smernica kako je ista realizovana, u opštem slučaju. Ali takođe treba napomenuti da ne postoji jedinstveni poziv nekog sistemskog poziva za postavljanje tačke prekida, već se ista obavlja kao kombinacija više mogućnosti sistemskog poziva `ptrace`. Opišimo ceo postupak na jednostavnom primeru.

Program je preveden za procesorsku arhitekturu Intel x86-64 i asemblerski kôd `main` funkcije primera izgleda kao u primeru sa slike 2.6.



```

000000004006e1 <main>:
4006e1: 55                push    %rbp
4006e2: 48 89 e5          mov     %rsp,%rbp
4006e5: 48 83 ec 40       sub     $0x40,%rsp
4006e9: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
4006f0: 00 00
4006f2: 48 89 45 f8       mov     %rax,-0x8(%rbp)
4006f6: 31 c9            xor     %eax,%eax
4006f8: c7 45 cc 00 00 00 00 movl    $0x0,-0x34(%rbp)
4006ff: eb 31            jmp     400732 <main+0x51>
400701: 8b 55 cc          mov     -0x34(%rbp),%edx
400704: 48 8d 45 d0       lea     -0x30(%rbp),%rax
400708: 48 03 d2          movslq  %edx,%rdx
40070b: 48 c1 e2 03       shl     $0x3,%rdx
40070f: 48 8d 3c 10       lea     (%rax,%rdx,1),%rdi
400713: 48 8d 45 cc       lea     -0x34(%rbp),%rax
400717: 48 89 c1          mov     %rax,%rcx
40071a: ba b6 06 40 00    mov     $0x4006b6,%edx
40071f: be 00 00 00 00    mov     $0x0,%esi
400724: e8 47 fe ff ff    callq  400570 <pthread_create@plt>
400729: 8b 45 cc          mov     -0x34(%rbp),%eax
40072c: 83 c9 01          add     $0x1,%eax
40072f: 89 45 cc          mov     %eax,-0x34(%rbp)
400732: 8b 45 cc          mov     -0x34(%rbp),%eax
400735: 83 f8 04          cmp     $0x4,%eax
400738: 7e c7            jle     400701 <main+0x20>
40073a: bf 05 00 00 00    mov     $0x5,%edi
40073f: e8 5c fe ff ff    callq  4005a0 <sleep@plt>
400744: e8 37 fe ff ff    callq  400580 <abort@plt>
400749: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)

```

Slika 2.6: Asemblerski kôd `main` funkcije test primera na x86-64 platformi.

Primeru radi, želimo da postavimo tačku prekida na treću po redu instrukciju funkcije `main`:

```
48 89 e5 mov %rsp,%rbp
```

Da bismo to uradili, menjamo prvi bajt instrukcije sa posebnom magičnom vrednošću, obično `0xcc`, i kada izvršavanje dostigne do tog dela koda ono će se zaustaviti na tom mestu.

Pošto `0x48` menjamo sa `0xcc` i na tom mestu u kodu dobijamo instrukciju:

```
cc 89 e5 int3
```

Ukoliko korisnik želi da nastavi dalje, instrukcija prekida se zamenjuje sa originalnom instrukcijom koja se izvršava i nastavlja se sa radom programa.

Instrukcija `int3` je posebna instrukcija procesorske arhitekture Intel x86-64, koja izazva softverski prekid. Kada registar programski brojač (eng. *CPU register pc*) stigne do `int3` instrukcije izvršavanje se zaustavlja na toj tački. Debager je već upoznat od strane korisnika da je tačka prekida postavljena te on čeka na signal koji ukazuje na to da je program dostigao do instrukcije prekida. Operativni sistem prepoznaje instrukciju `int3`, poziva se specijalni obrađivač tog signala (na Linux sistemima `do_int3()`), koji dalje obaveštava debager šaljući mu signal sa kodom `SIGTRAP` koji on obrađuje na željeni način. Treba napomenuti da ovo važi za procesorsku arhitekturu Intel x86-64, instrukcija prekida za arhitekture kao što su ARM, MIPS, PPC itd., se drugačije kodira, ali postupak implementacije tačaka prekida je isti.

Ukoliko želimo da stavimo tačku prekida eksplicitno na funkciju `main`, za to koristimo posrednika u vidu DWARF debug informacija. U tom slučaju debager traži element DWARF stabla koji ukazuje na informacije o `main` funkciji, i odatle dohvata informaciju na kojoj adresi u memoriji se nalazi prva mašinska instrukcija date funkcije. Na slici 2.7 vidimo DWARF element koji opisuje funkciju uz pomoć atributa. Debager će pročitati `DW_AT_low_pc` atribut i na tu adresu postaviti `int3` instrukciju. Napomenimo da DWARF debug simbole generišemo uz pomoć `-g` opcije kompajlera.

```
<1><bd>: Abbrev Number: 6 (DW_TAG_subprogram)
<be> DW_AT_external : 1
<be> DW_AT_name : (indirect string, offset: 0x21): main
<c2> DW_AT_decl_file : 1
<c3> DW_AT_decl_line : 15
<c4> DW_AT_prototyped : 1
<c4> DW_AT_type : <0x57>
<c8> DW_AT_low_pc : 0x4006e1
<d0> DW_AT_high_pc : 0x68
<d8> DW_AT_frame_base : 1 byte block: 9c (DW_OP_call_frame_cfa)
<da> DW_AT_GNU_all_tail_call_sites: 1
<da> DW_AT_sibling : <0xfa>
```

Slika 2.7: `main` funkcija prikazana DWARF potprogramom.

Hardverske tačke prekida su direktno povezane sa hardverom u vidu specijalnih registara. Postavlja se na određenu adresu i hardverski *watchpoint* monitori za zadatu adresu mogu signalizirati razne promene, npr. čitanje, pisanje ili izvršavanje, što im daje prednost u odnosu na softverske tačke prekida. Mane u odnosu na softverske tačke prekida su performanse, koje su neuporedivo sporije, i takođe neophodna hardverska podrška za korišćenje hardverskih tačaka prekida.

Koraćanje

Pod procesom koraćanja (eng. *stepping*) kroz program podrazumevamo izvršavanje programa sekvencu po sekvencu. Sekvenca može biti jedna procesorska instrukcija, linija koda ili pak neka funkcija programa koji se debuguje [4].

Instrukcijsko koraćanje na platformi Intel x86-64 je direktno omogućeno kroz sistemski poziv `ptrace`:

```
ptrace(PTRACE_SINGLESTEP, debuggee_pid, nullptr, nullptr);
```

Operativni sistem će poslati debageru signal `SIGTRAP` kada je korak izvršen.

Pored instrukcijskog koraćanja pomenućemo još jednu vrstu koraćanja u neku funkciju koja je pozvana `call` ili `jump` instrukcijom. Komanda debagara GNU GDB koja nam to omogućava jeste `step in`.

Treba napomenuti da postoje arhitekture za koje ovo ne važi, kao npr. platforma ARM, koja nema hardversku podršku za instrukcijsko koraćanje i za njih se koraćanje implementira na drugačiji način, uz pomoć emulacije instrukcija, ali u ovom radu neće biti reči o tome.

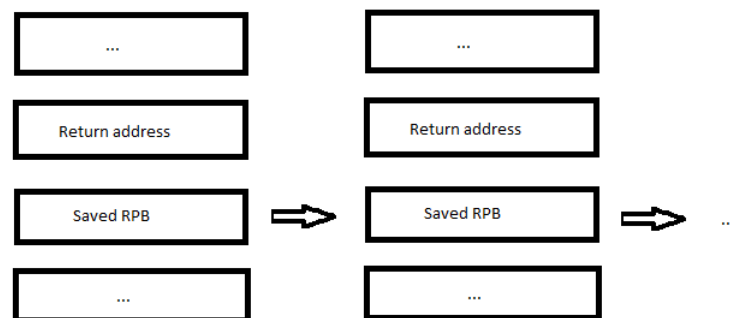
Izlistavanje pozivanih funkcija

Objasnimo komandu izlistavanje pozivanih funkcija (eng. *backtrace*) posmatrajući organizaciju stek okvira (eng. *stack frames*) na platformi Intel x86-64 [4].

Na slici 2.8 navedeni su stek okviri za dva funkcijska poziva. Pre povratne vrednosti funkcije obično se ređaju argumenti funkcije. Sačuvana adresa u registru `RBP` jeste adresa stek okvira svog pozivaoca. Prateći sve okvire kao elemente povezane liste dolazimo do svih pozivanih funkcija do zadate tačke. Ako se pitamo kako debager ima informaciju o imenu funkcije odgovor je u tome što pretražuje DWARF stablo sa debug informacijama, tražeći `DW_TAG_subprogram` sa odgovarajućom povratnom adresom, pritom čitajući `DW_AT_name` atribut tog elementa.

Čitanje vrednosti promenljivih

Za čitanje vrednosti promenljivih u programu, debager pretražuje DWARF stablo tražeći promenljivu sa zadatim imenom. U slučaju lokalnih promenljivih, traži se `DW_TAG_variable` element čiji `DW_AT_name` odgovara navedenoj promenljivoj. Kada



Slika 2.8: Primer ređanja stek okvira na x86-64 platformi.

se ista pronađe konsultuje se `DW_AT_location`, koji ukazuje na lokaciju gde se vrednost promenljive nalazi. Ukoliko ovaj atribut nije naveden debager će vrednost takve promenljive smatrati kao optimizovanu prijavljujući informaciju o tome [4].

Glava 3

GNU GDB alat

Glava 4

Datoteke jezgra

Glava 5

TLS

5.1 Motivacija

5.2 Rukovanje TLS-om tokom izvršavanja programa

5.3 Arhitekturno specifične zavisnosti

5.4 TLS Modeli pristupa

Glava 6

Implementacija rešenja

Glava 7

Zaključak

Literatura

- [1] *ELF Format*. on-line at: <http://www.cs.cmu.edu/afs/cs/academic/class/15213-s00/doc/elf.pdf>. 1992.
- [2] Free Software Foundation. *DWARF Format*. on-line at: <http://dwarfstd.org/>. 1992.
- [3] Linux Foundation. *ptrace*. on-line at: <http://man7.org/linux/man-pages/man2/ptrace.2.html>.
- [4] *GNU GDB*. on-line at: <https://www.gnu.org/software/gdb/>.
- [5] Reid Kleckner. *CodeView*. on-line at: <https://llvm.org/devmtg/2016-11/Slides/Kleckner-CodeViewInLLVM.pdf>.

Biografija autora