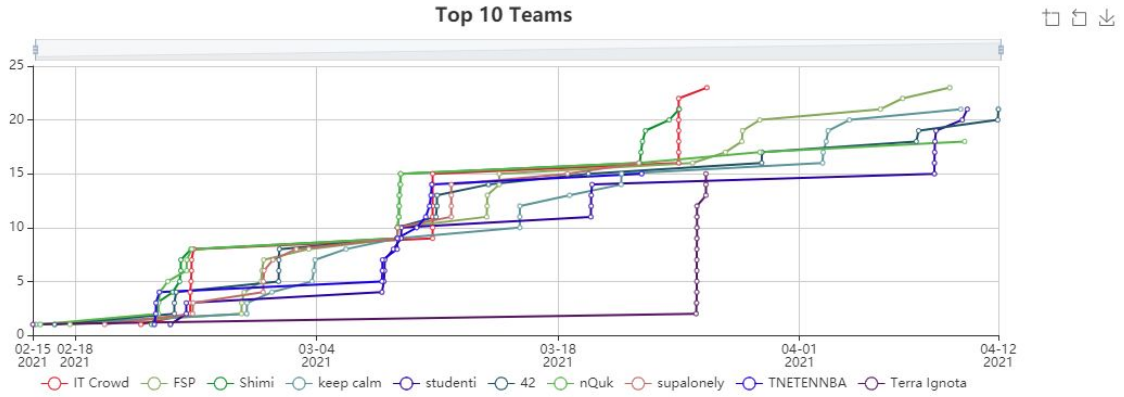
Návratovo orientované programovanie

Peter Švec
peter_svec@stuba.sk

Pamäťové zraniteľnosti výsledky



Place	Team	Score
1	IT Crowd	23
2	FSP	23
3	Shimi	21
4	keep calm	21
5	studenti	21
6	42	21

Úvod

- Máme buffer overflow zraniteľnosť.
- Máme však zapnutú **NX** ochranu (teoreticky vieme injektovať shellcode ale nevieme ho spustiť)
- Game over?

Návratovo orientované programovanie

- **ROP** (**R**eturn **O**riented **P**rogramming)
- Hlavnou myšlienkou je využiť už existujúci kód (.text sekcia, libc,...), ktorý má e**X**ecute oprávnenia!
- Základom sú krátke časti kódu (**gadget**), pomocou ktorých vieme zostaviť **ROP chain**.
- ROP je **Turing complete**.

Gadgety

- Lubovoľná postupnosť inštrukcií ukončená inštrukciou **RET**
- ROP gadget je väčšinou divná postupnosť inštrukcií (často skáčeme aj do stredu jednej inštrukcie)
- Gadgety vieme následne reťaziť vďaka inštrukcii **RET**

```
pop r12
pop rdi
pop rsi
ret
```

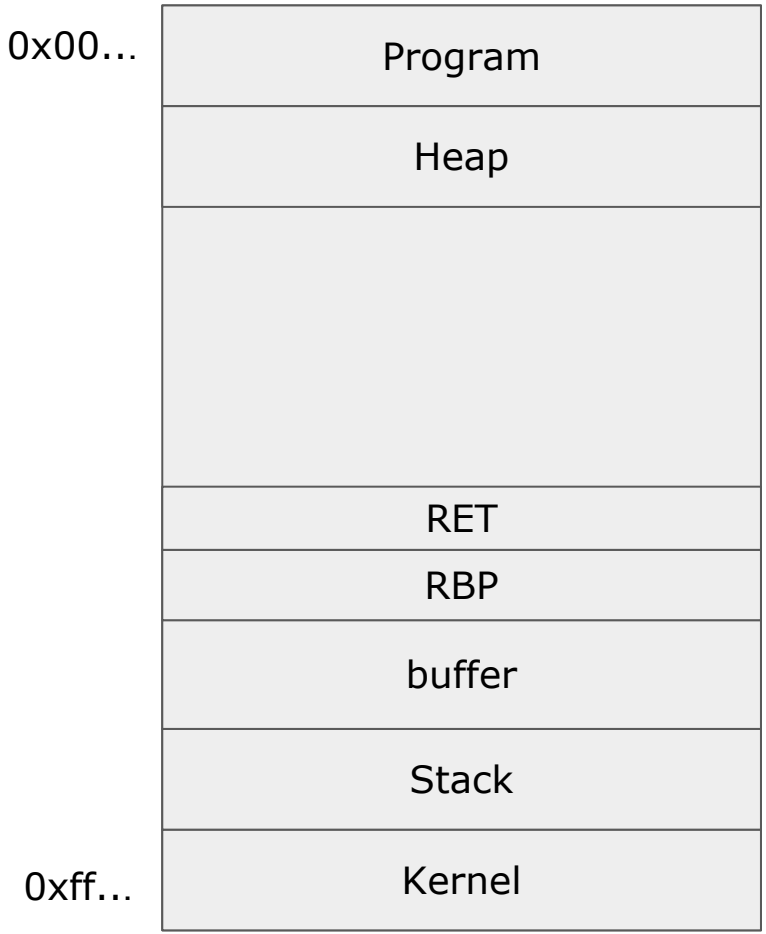
```
syscall
ret
```

```
add byte [rcx], al
pop rbp
ret
```

```
add rsp, rax
ret
```

Example (exit)

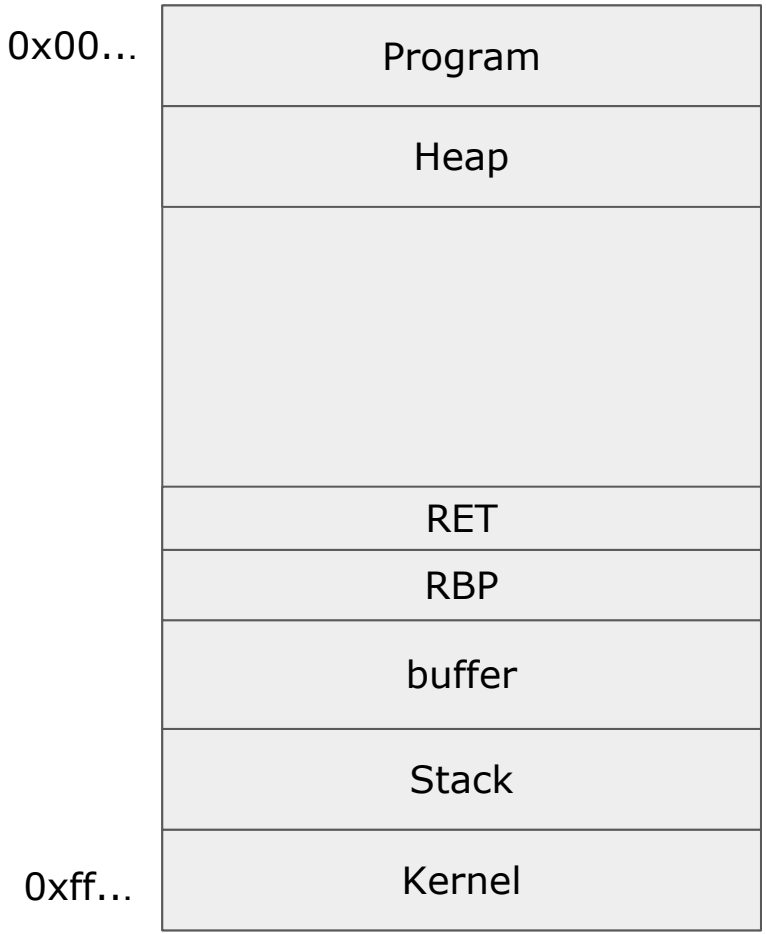
```
mov rax, 60
mov rdi, 0
syscall
```



Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

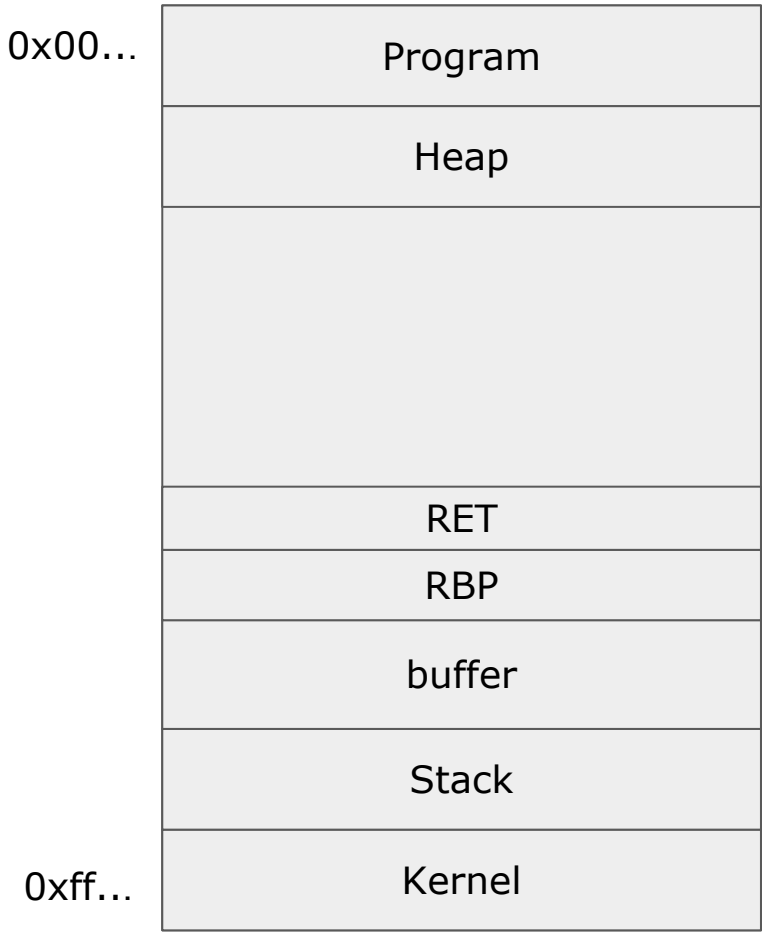


Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**



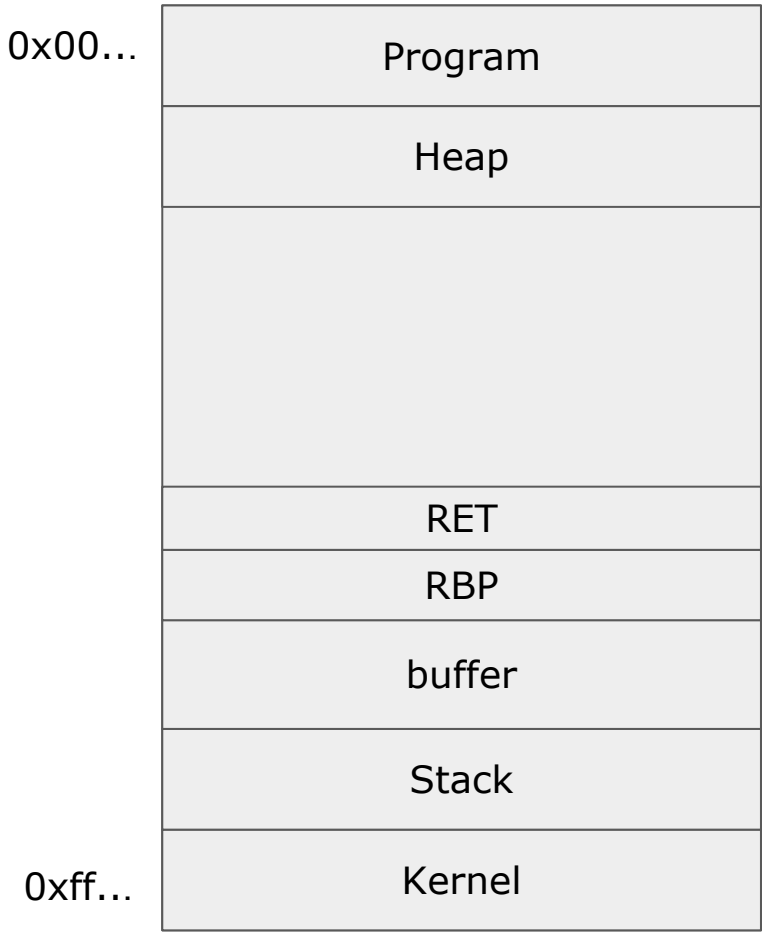
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

0x4000aaa **syscall**
0x4000aab **ret**



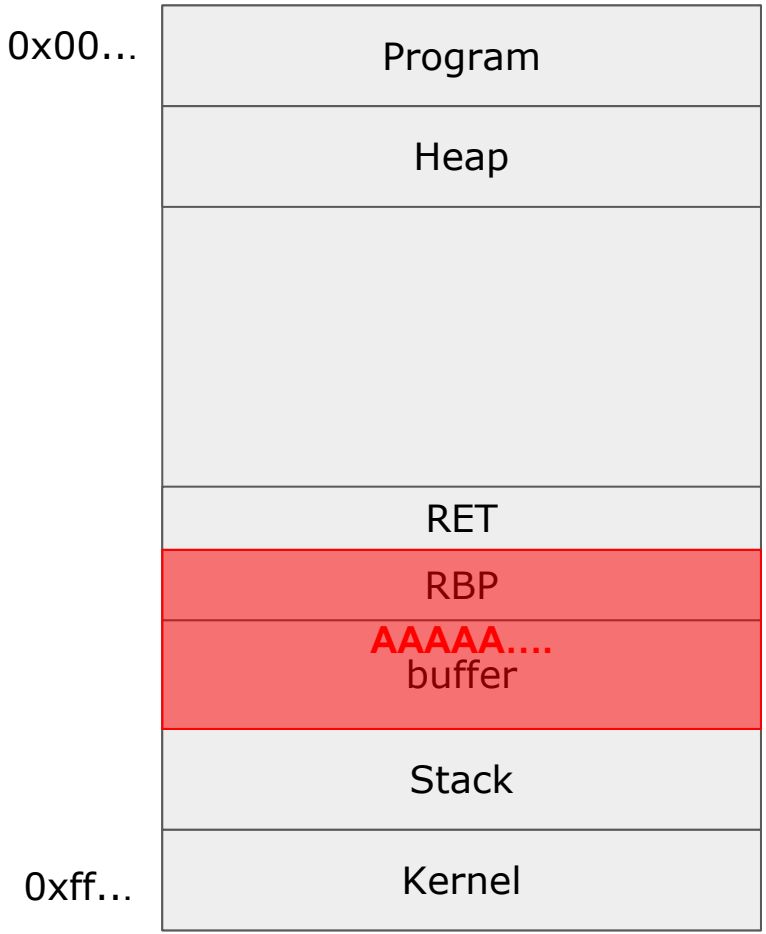
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

0x4000aaa **syscall**
0x4000aab **ret**



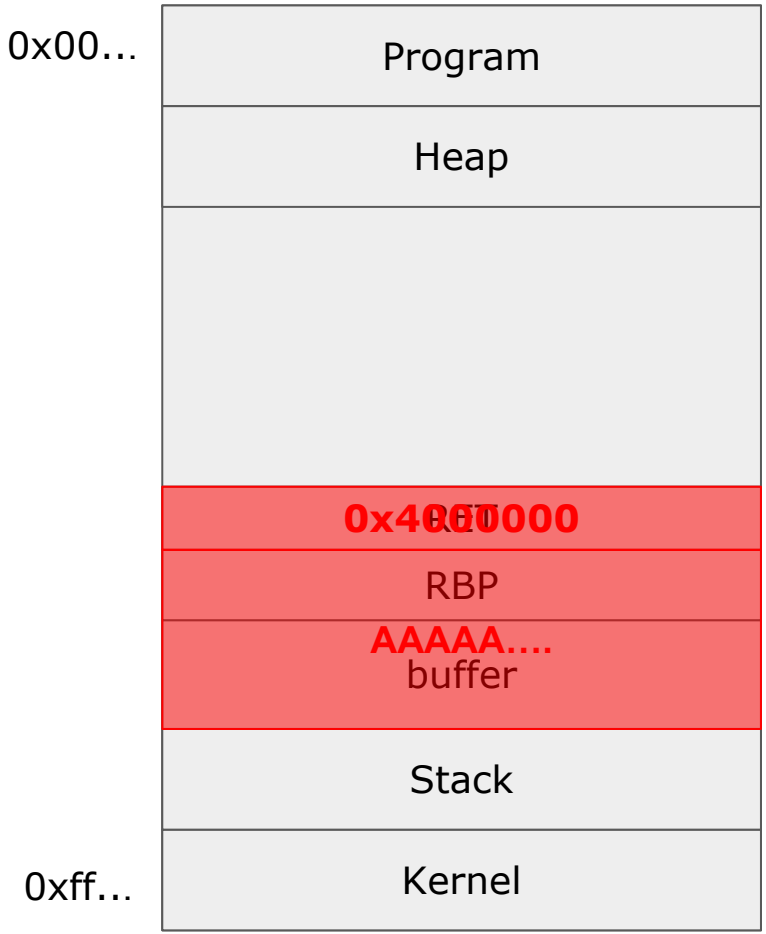
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

0x4000aaa **syscall**
0x4000aab **ret**



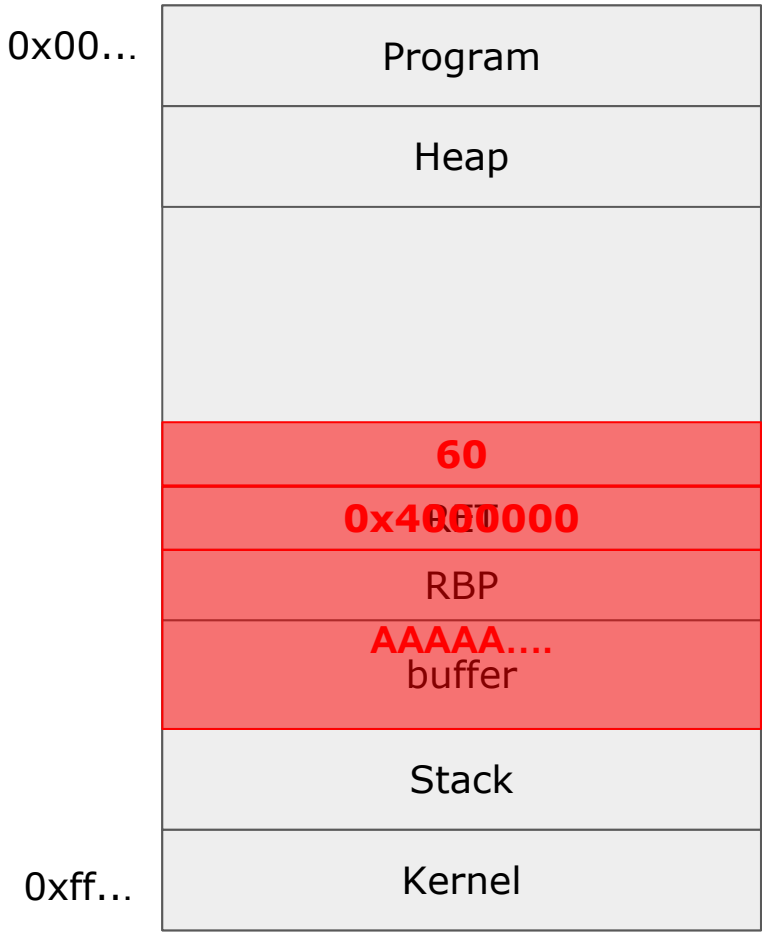
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

0x4000aaa **syscall**
0x4000aab **ret**



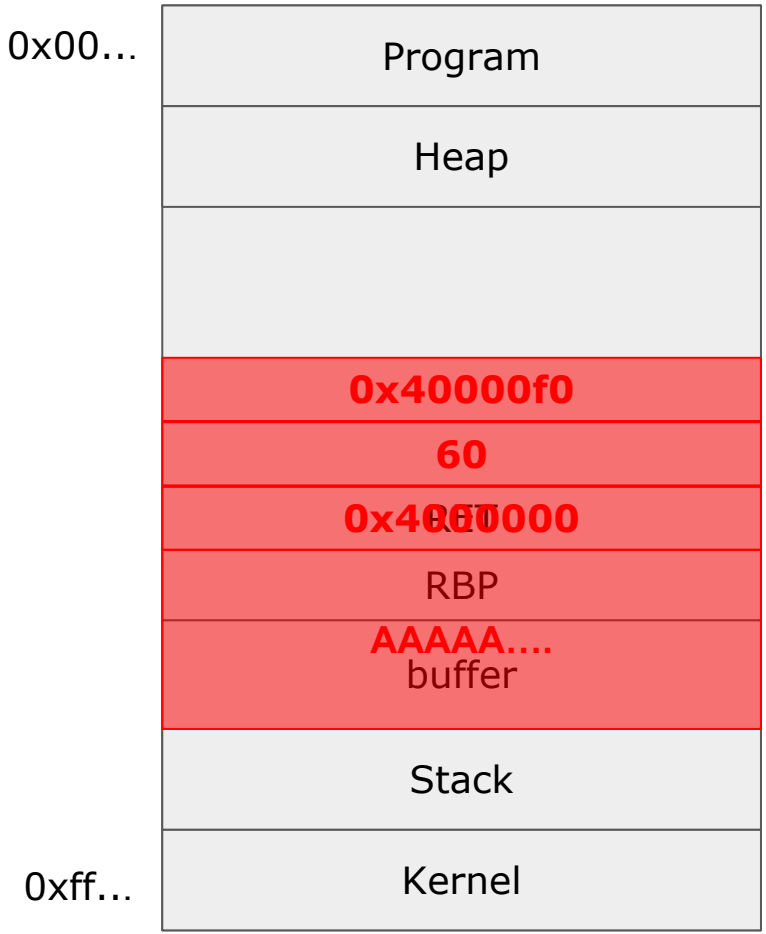
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

0x4000aaa **syscall**
0x4000aab **ret**



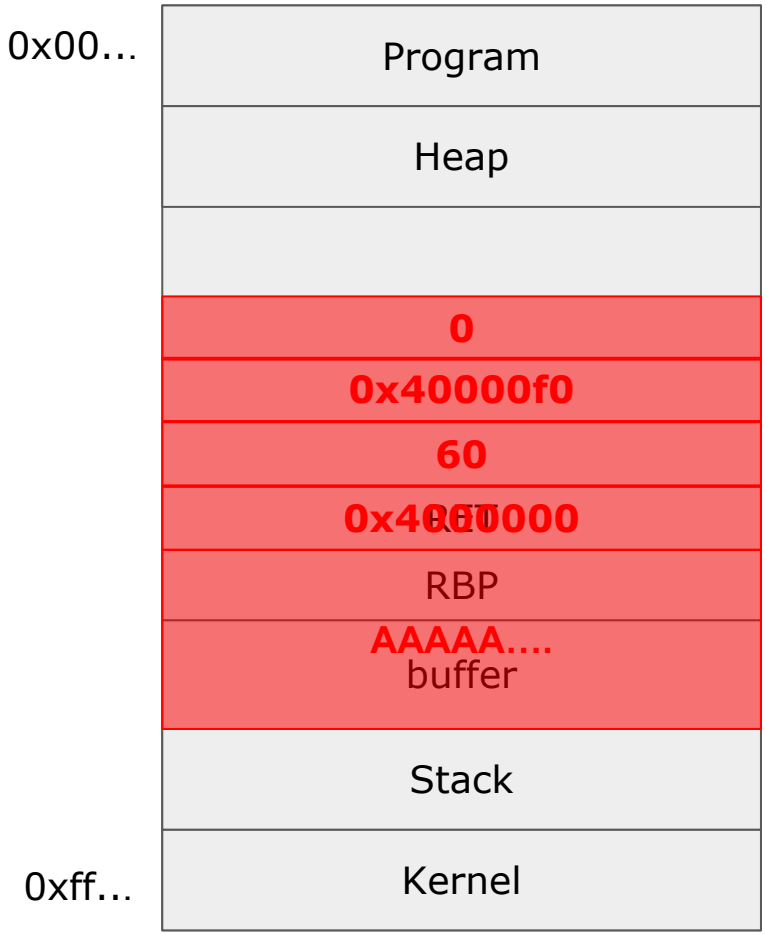
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

0x4000aaa **syscall**
0x4000aab **ret**



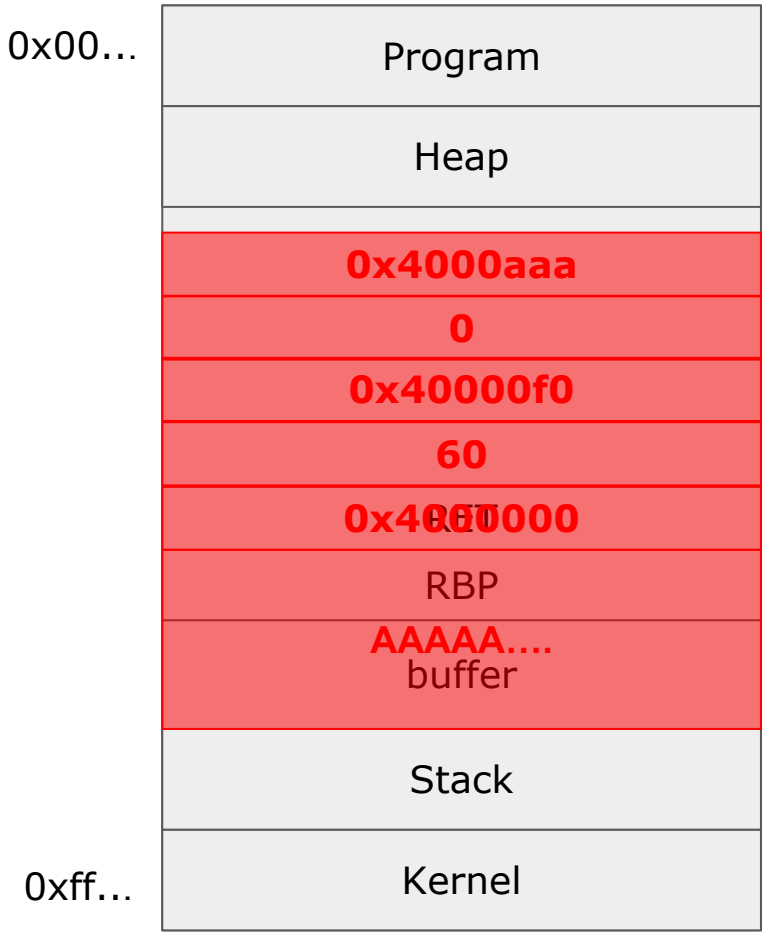
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

0x4000aaa **syscall**
0x4000aab **ret**



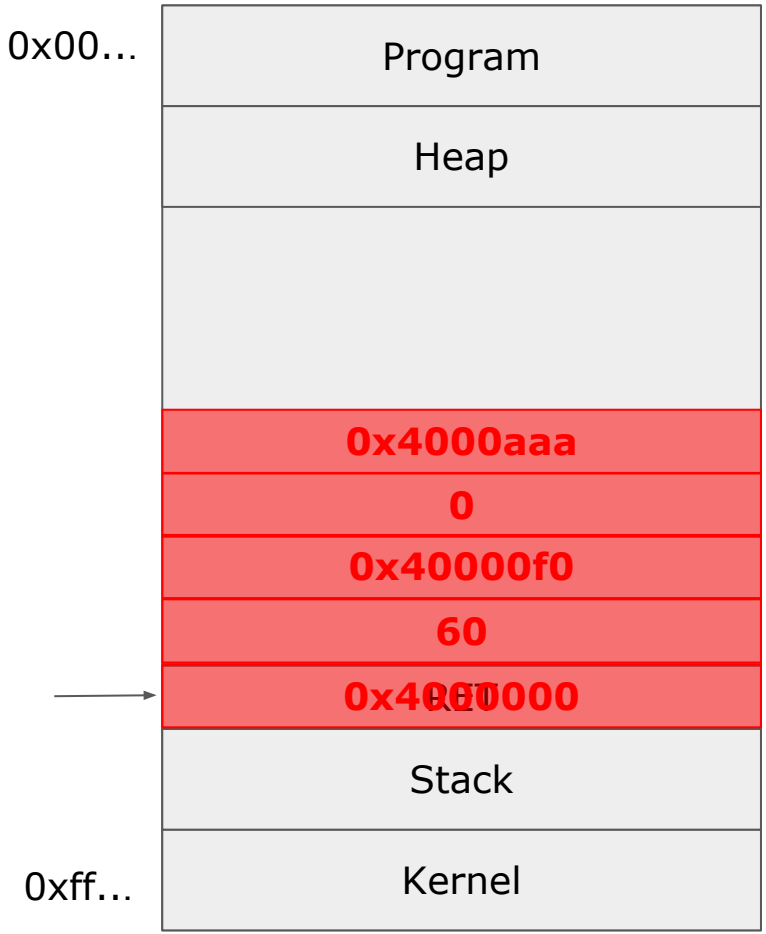
Example (exit)

mov rax, 60
mov rdi, 0
syscall

0x4000000 **pop** rax
0x4000002 **ret**

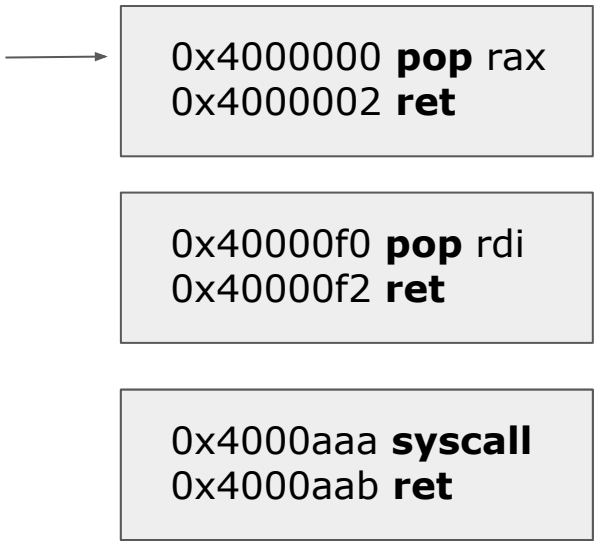
0x40000f0 **pop** rdi
0x40000f2 **ret**

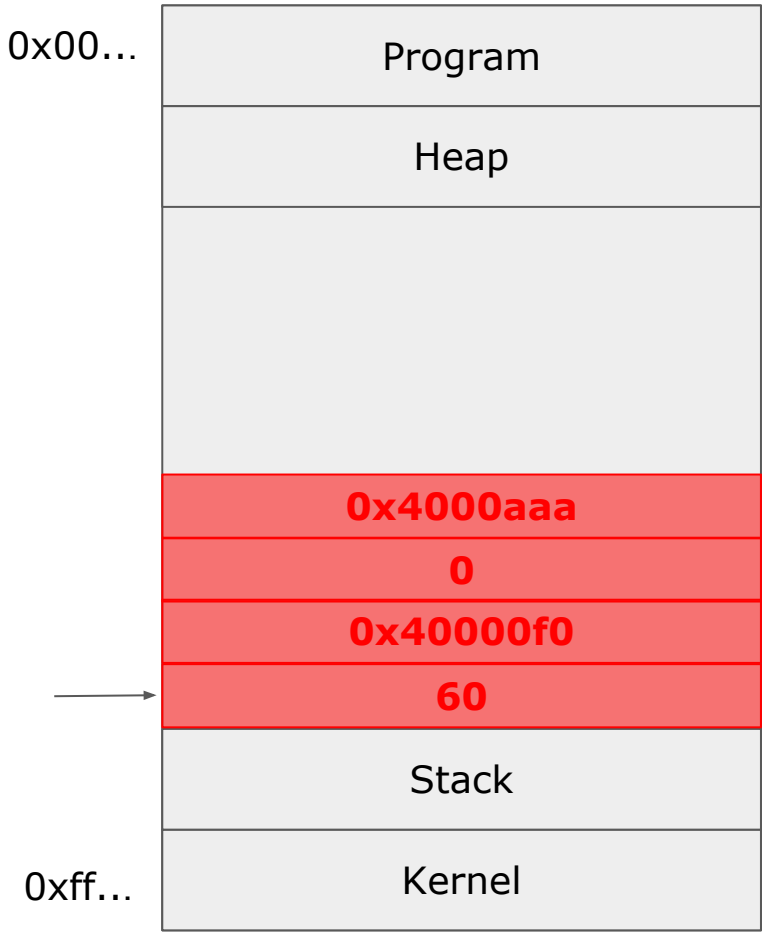
0x4000aaa **syscall**
0x4000aab **ret**



Example (exit)

mov rax, 60
mov rdi, 0
syscall

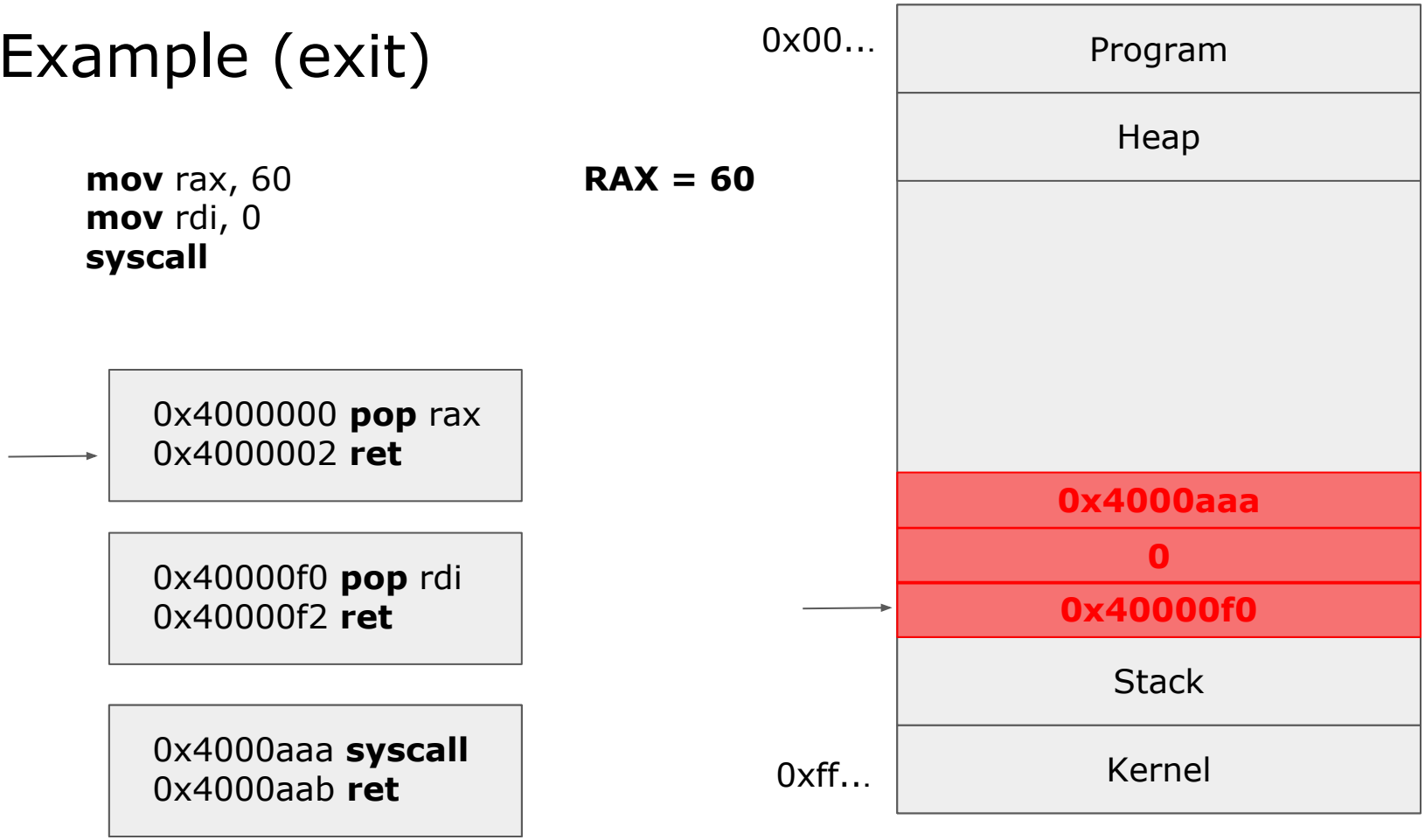




Example (exit)

mov rax, 60
mov rdi, 0
syscall

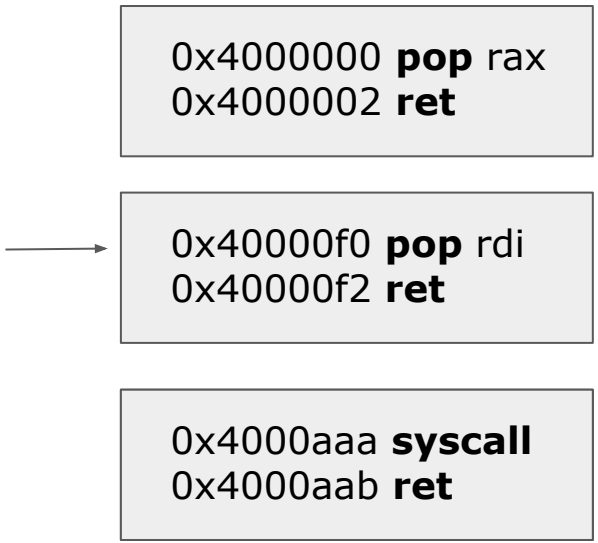
RAX = 60

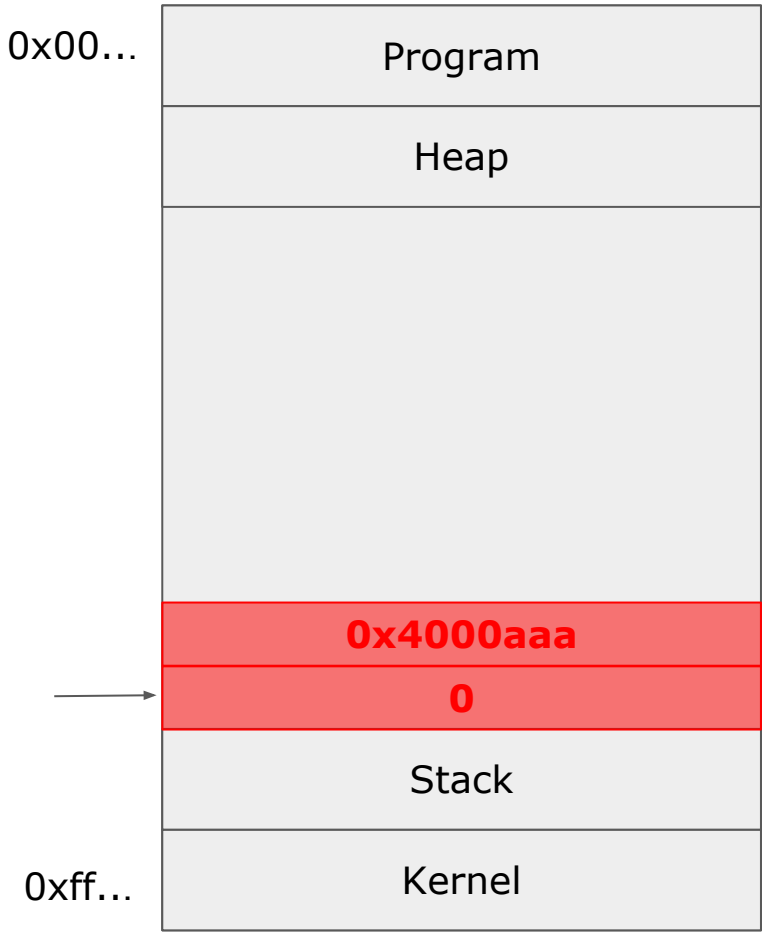


Example (exit)

mov rax, 60
mov rdi, 0
syscall

RAX = 60

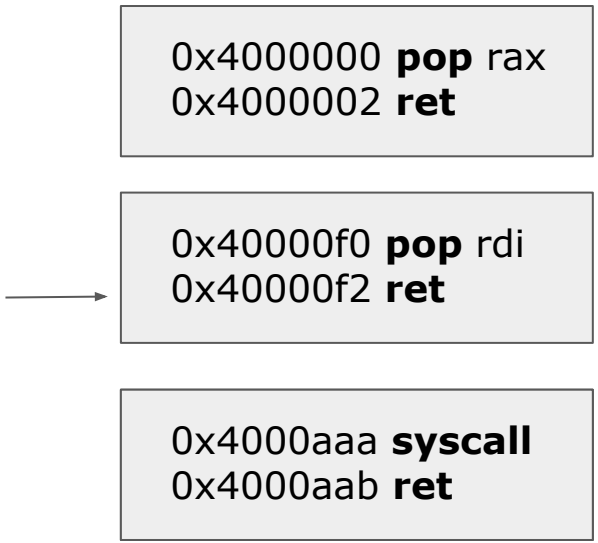


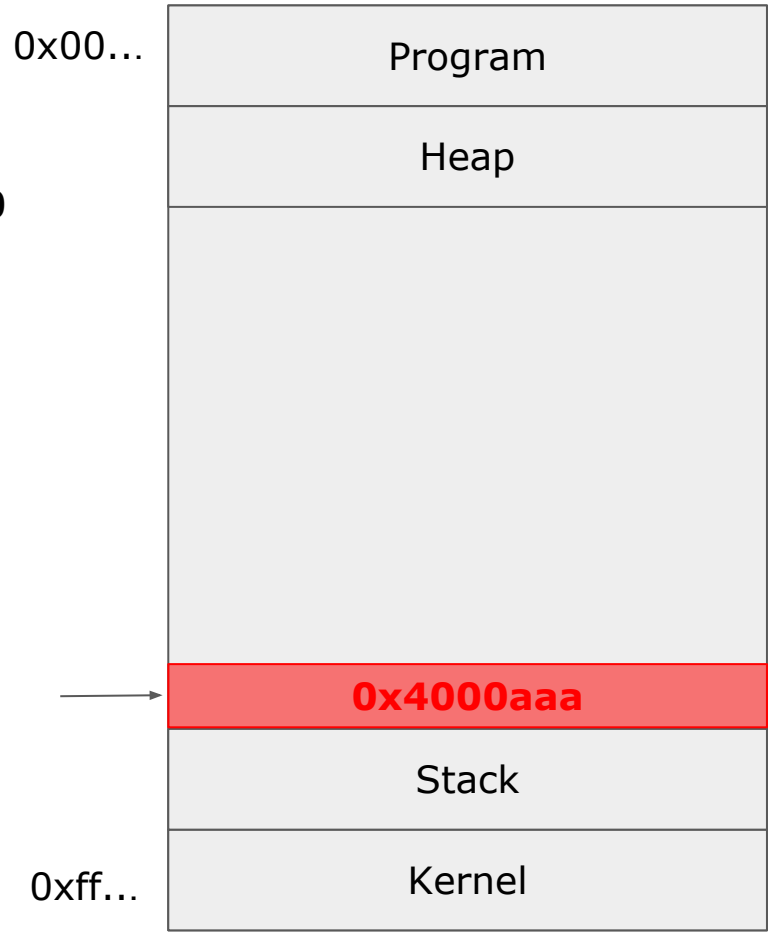


Example (exit)

mov rax, 60
mov rdi, 0
syscall

RAX = 60
RDI = 0





Example (exit)

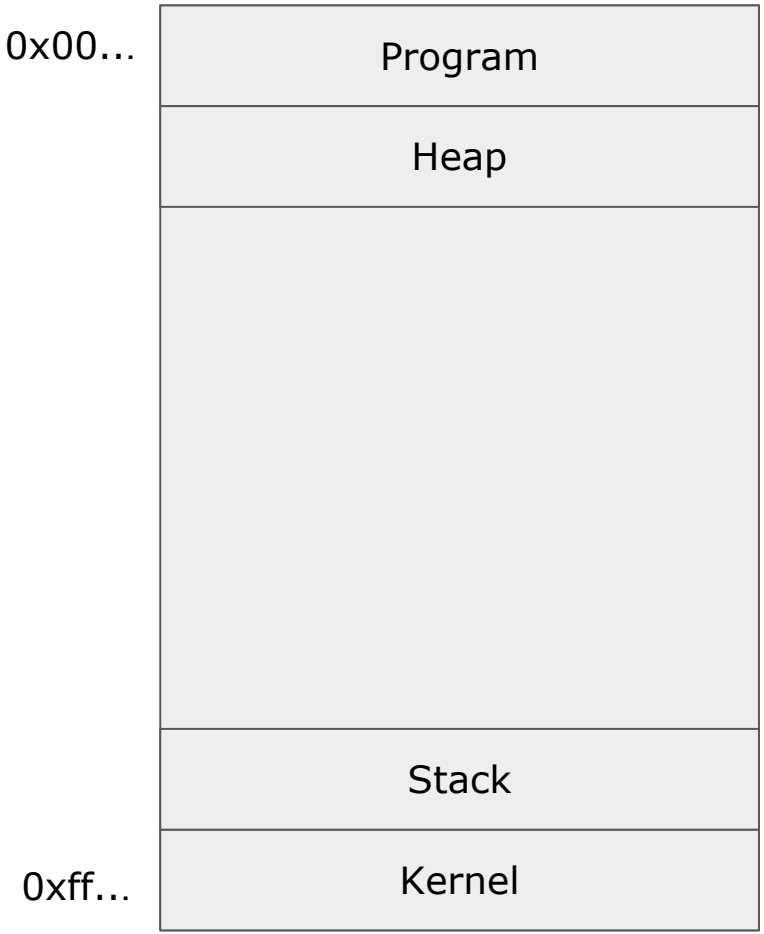
mov rax, 60
mov rdi, 0
syscall

RAX = 60
RDI = 0

0x4000000 **pop** rax
0x4000002 **ret**

0x40000f0 **pop** rdi
0x40000f2 **ret**

→ 0x4000aaa **syscall**
0x4000aab **ret**



Payload

```
p.send(b”A” * offset
    + pwn.p64(pop_rax_gadget)
    + pwn.p64(60)
    + pwn.p64(pop_rdi_gadget)
    + pwn.p64(0)
    + pwn.p64(syscall_gadget)
)
```

Ako hľadať gadgety?

- Našťastie máme nástroje.
- V CTF kontajneri: **rp++** a **grep**
- Ďalšie: ropper, ROPgadget,...

rp++ --unique -r 2 -f cesta_k_suboru | grep “co hladame”

Problémy

- V samotnej binárke býva väčšinou veľmi málo dobrých gadgetov :(
- Väčšina binároek však používa štandardnú knižnicu (**libc**), ktorá obsahuje extrémne množstvo gadgetov :)
- Ale...ASLR...potrebujeme leak :(

Riešenie 1

1. Majme leaknutú adresu: **malloc_leak = 0x7ff2defee** (pozor, mení sa pri každom spustení)

Riešenie 1

1. Majme leaknutú adresu: **malloc_leak = 0x7ff2defee** (pozor, mení sa pri každom spustení)
2. Vieme že malloc je vždy na fixnom offsete od začiatku binárky

    ldd cesta_k_binarke -> vráti nám cestu k libc na systéme

    readelf -s cesta_k_libc | grep “malloc” -> získame offset

Riešenie 1

1. Majme leaknutú adresu: **malloc_leak = 0x7ff2defee** (pozor, mení sa pri každom spustení)

2. Vieme že malloc je vždy na fixnom offsete od začiatku binárky

    ldd cesta_k_binarke -> vráti nám cestu k libc na systéme

    readelf -s cesta_k_libc | grep “malloc” -> získame offset

3. V predchádzajúcom kroku sme zistili offset: **malloc_offset = 0x000009d260**, vieme tak vypočítať base adresu libc v pamäti

    libc_base = malloc_leak - malloc_offset

Riešenie 1

1. Majme leaknutú adresu: **malloc_leak = 0x7ff2defee** (pozor, mení sa pri každom spustení)
2. Vieme že malloc je vždy na fixnom offsete od začiatku binárky

    ldd cesta_k_binarke -> vráti nám cestu k libc na systéme

    readelf -s cesta_k_libc | grep “malloc” -> získame offset

3. V predchádzajúcom kroku sme zistili offset: **malloc_offset = 0x000009d260**, vieme tak vypočítať base adresu libc v pamäti

    libc_base = malloc_leak - malloc_offset

4. Pomocou base adresy vieme následne vypočítať adresu ľubovoľného gadgetu v libc

    gadget_address = libc_base + gadget_offset

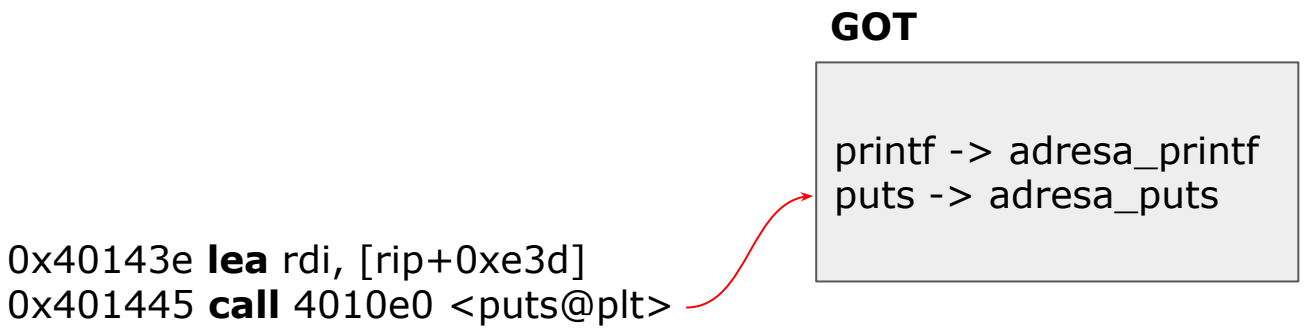
Riešenie 2

Dynamic Loader používa dve tabuľky:
GOT (**G**lobal **O**ffset **T**able) a **PLT** (**P**rocedure **L**inkage **T**able)

0x40143e **lea** rdi, [rip+0xe3d]
0x401445 **call** 4010e0 <puts@plt>

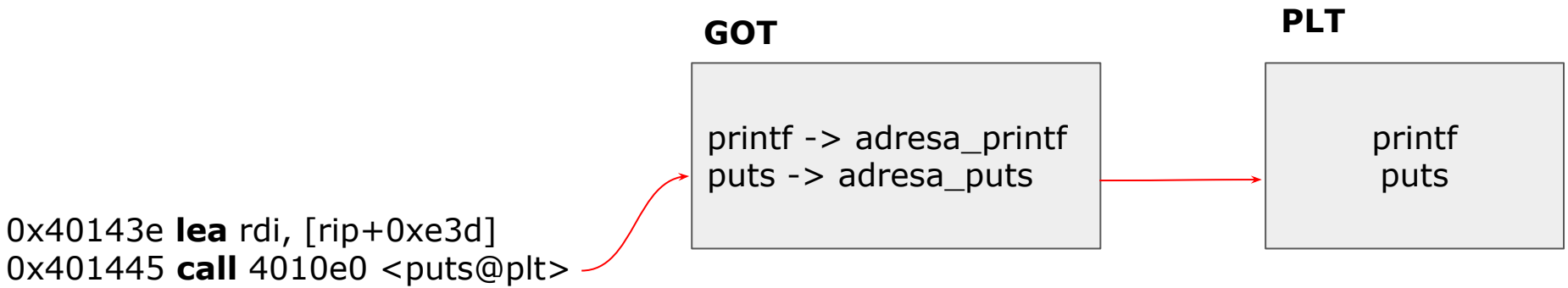
Riešenie 2

Dynamic Loader používa dve tabuľky:
GOT (**G**lobal **O**ffset **T**able) a **PLT** (**P**rocedure **L**inkage **T**able)



Riešenie 2

Dynamic Loader používa dve tabuľky:
GOT (**G**lobal **O**ffset **T**able) a **PLT** (**P**rocedure **L**inkage **T**able)



1. Nájde puts v libc
2. Uloží adresu do GOT
3. Zavolá puts

Riešenie 2

Dynamic Loader používa dve tabuľky:
GOT (**G**lobal **O**ffset **T**able) a **PLT** (**P**rocedure **L**inkage **T**able)

1. Adresy záznamov pre **GOT** a **PLT** sú fixných adresách:

```
e = pwn.ELF(‘./rop_level_1’)
puts_got = e.got[‘puts’]
puts_plt = e.plt[‘puts’]
```

Riešenie 2

Dynamic Loader používa dve tabuľky:
GOT (**G**lobal **O**ffset **T**able) a **PLT** (**P**rocedure **L**inkage **T**able)

1. Adresy záznamov pre **GOT** a **PLT** sú fixných adresách:

```
e = pwn.ELF(‘./rop_level_1’)
puts_got = e.got[‘puts’]
puts_plt = e.plt[‘puts’]
```

2. Pomocou ROP chainu vieme využiť volanie puts (z PLT) na prečítanie hodnoty puts z GOT:

pop_rdi + puts_got + puts_plt

3. puts nám vypíše na obrazovku adresu funkcie puts v libc...ďalší postup je rovnaký ako v predchádzajúcom riešení

Hinty

- Cez ROP môže byť problematické získať pointer na reťazec...prečo však nevyužiť už existujúce reťazce v binárke? (viď disassembler)
- Keep it simple. Snažiť sa postaviť komplikovaný chain môže byť náročné...prečo nevyužiť podobný prístup ako v Shellcode bloku (Level 7)
- Na opakovanie zraniteľností nie je vždy nutný epický HACK. Čo tak vrátiť sa do mainu? Alebo _start?