# ROP

CS-UY 3943-G / CS-GY 9223-H

# Motivation

- Consider a situation where we have:
  - A buffer overflow or other vulnerability that gives us control over the return address and values on the stack
- But:
  - No executable stack (NX)
  - No handy shell-like functions (give_shell, run_cmd, system)
- How can we still get code execution?

# Return-Oriented Programming

- Solution: use code that's already in the program!
- x86 code is *variable length* (each instruction can be 1 to 15 bytes long) and *dense* (most random byte sequences are valid instructions)
- This means that you can jump to the middle of an instruction and usually get a valid instruction

# Pop, pop, ret

- The core idea is to find sequences of instructions (called *gadgets*) in the binary that end in *ret*

- Each of these does a little bit of work and then transfers control to the next gadget via *ret*

# Simple Example: execve(“/bin/sh”)

- Let’s give ourselves some gadgets, and assume the string “/bin/sh” is in memory at 0x600000

- The full call to execve is
  execve(“/bin/sh”, NULL, NULL)

- In assembly, we roughly want
  mov rdi, 0x600000
  mov rsi, 0
  mov rdx, 0
  mov rax, 59
  syscall

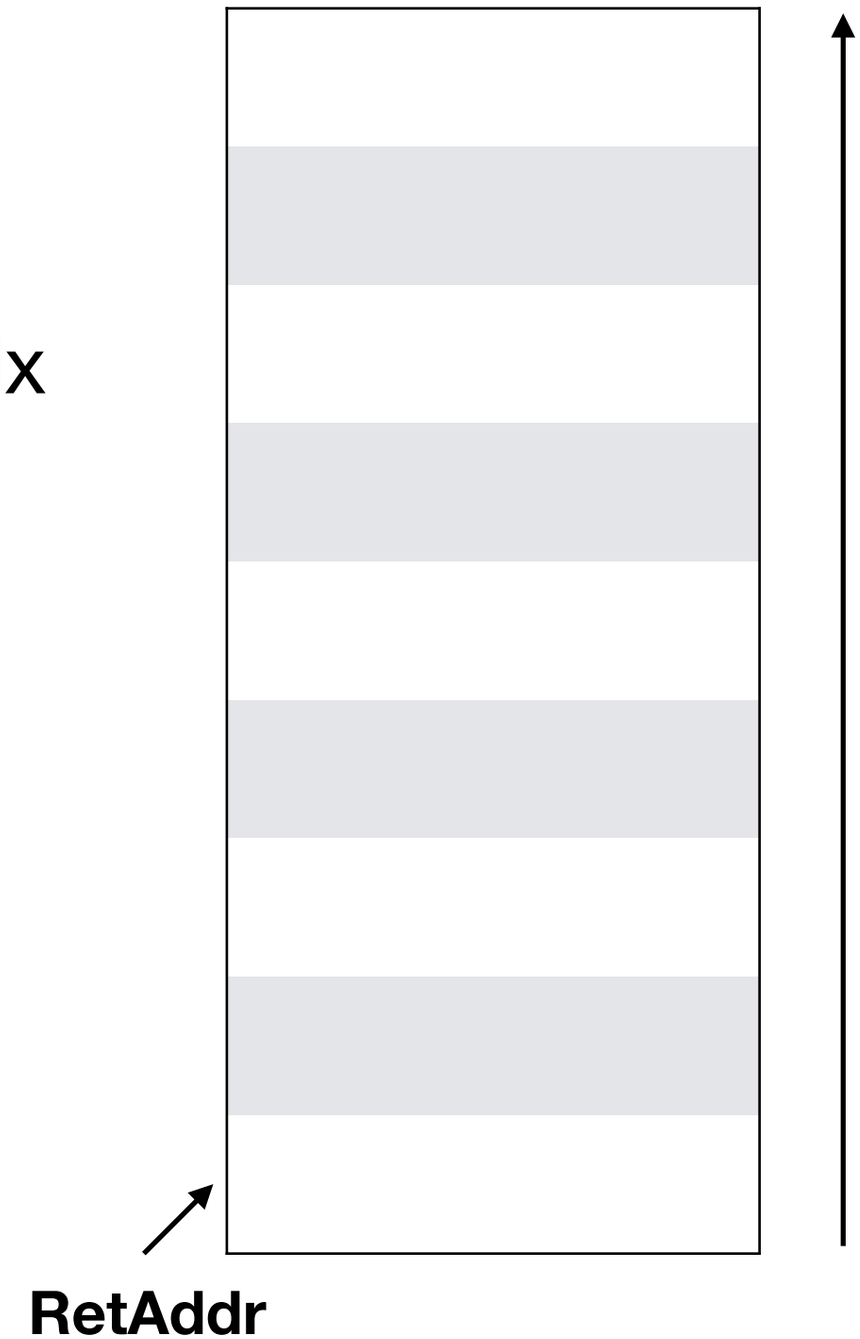
# Simple Example: execve(“/bin/sh”)

- Our gadgets:

| A | B | C | D |
|---|---|---|---|
| syscall<br>ret | pop rdi<br>ret | pop rsi<br>ret | pop rdx<br>ret |

| E |
|---|
| pop rax<br>ret |

- What should our stack look like?



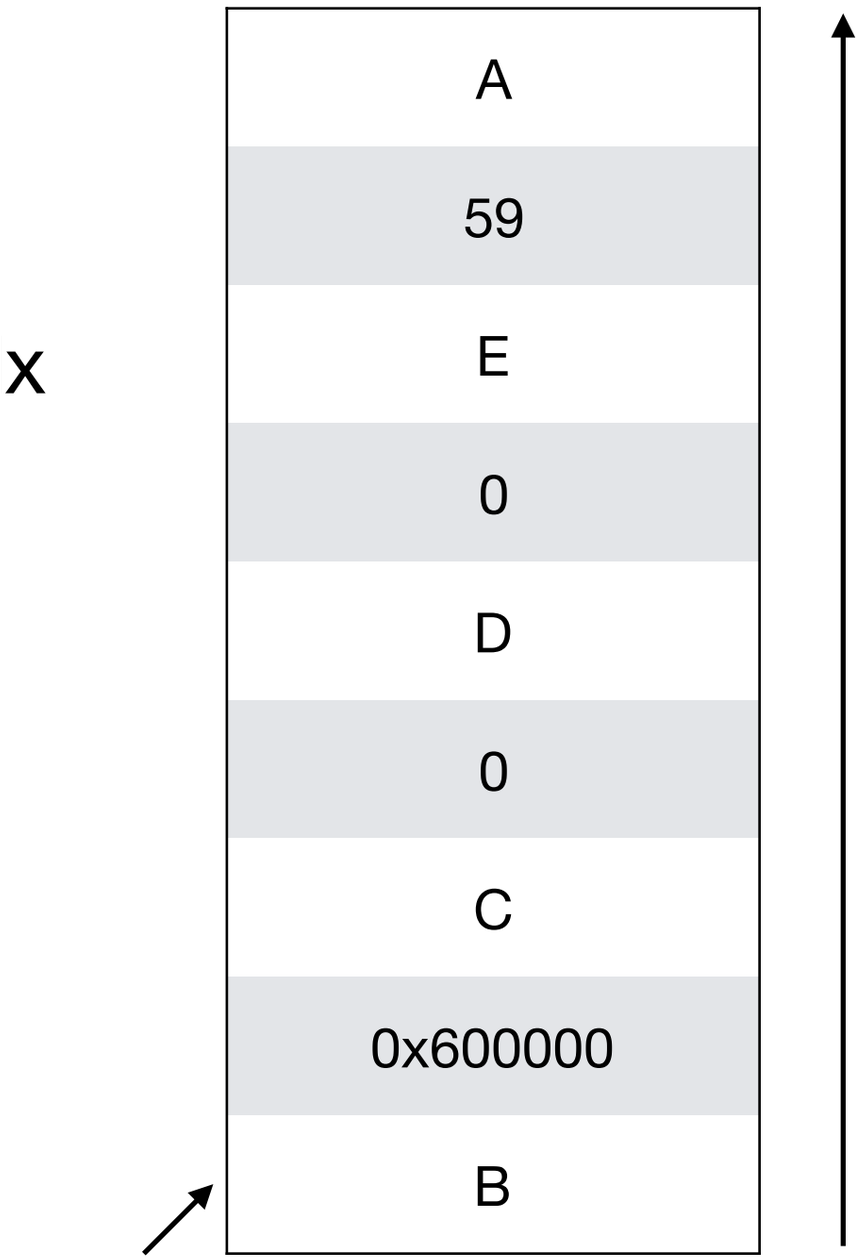
# Simple Example: execve(“/bin/sh”)

- Our gadgets:

| A | B | C | D |
|---|---|---|---|
| syscall<br>ret | pop rdi<br>ret | pop rsi<br>ret | pop rdx<br>ret |

| E |
|---|
| pop rax<br>ret |

- What should our stack look like?



# How do we find gadgets?

- Reminder: there are utilities that can do this for you!
- One that comes with pwntools is ROPgadget

```
(pwn) moyix@lorenzo:~/offsec/week_9/rop$ ROPgadget --binary ./rop
Gadgets information
============================================================
0x0000000000400564 : adc byte ptr [rax], ah ; jmp rax
0x00000000004005a4 : adc byte ptr [rax], ah ; jmp rdx
0x000000000040062d : add al, ch ; retf -1
0x00000000004006bf : add bl, dh ; ret
0x000000000040062b : add byte ptr [rax], al ; add al, ch ; retf -1
0x00000000004006bd : add byte ptr [rax], al ; add bl, dh ; ret
0x00000000004006bb : add byte ptr [rax], al ; add byte ptr [rax], al ; add bl, dh ; ret
0x00000000004006bc : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x00000000004004a3 : add byte ptr [rax], al ; add rsp, 8 ; ret
0x00000000004006be : add byte ptr [rax], al ; ret
0x00000000004005c8 : add byte ptr [rcx], al ; ret
0x00000000004005c4 : add eax, 0x200a8e ; add ebx, esi ; ret
0x000000000040058b : add eax, edx ; sar rax, 1 ; jne 0x40059c ; pop rbp ; ret
0x00000000004005c9 : add ebx, esi ; ret
[...]
```

# Dealing With LibC

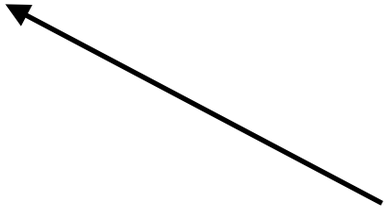
- Many binaries don’t have a lot of gadgets available
- You may want to call interesting functions in libc that aren’t imported by the target binary (so they’re not in the PLT/GOT)
- Unfortunately, the address of libc is *randomized*

# Finding LibC

- The trick is to notice that we have the address of several libc functions in the GOT already

- For example, after we call puts, the linker puts the address of puts in libc into the GOT for us:

```
(gdb) x/i 0x400715
   0x400715 <main+31>:  call   0x400580 <puts@plt>
(gdb) x/i 0x400580
   0x400580 <puts@plt>: jmp    QWORD PTR [rip+0x200a92]        # 0x601018
(gdb) x/xg 0x601018
0x601018:       0x00007ffff7a7c690
```



**This address is in libc!**

# Leaking a LibC address

- We can do the equivalent of puts(puts@GOT) to *leak* an address from libc

- Why does this help us?

- Once we know the address of puts, we can look up where puts is in libc

- Then just subtract off the offset of puts, and we have the base address of libc in memory

- From here we can get address of any other function in libc!

# Calculating libc Addresses

- In pwntools:

```
In [1]: from pwn import *

In [2]: # Suppose we know the address of puts is 0x00007ffff7a7c690

In [3]: puts_addr = 0x00007ffff7a7c690

In [4]: libc = ELF('libc-2.19.so')
[*] '/home/moyix/offsec/week_9/rop/libc-2.19.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

In [5]: libc_base = puts_addr - libc.symbols['puts']

In [6]: print hex(libc_base)
0x7ffff7a0c930

In [7]: system_addr = libc_base + libc.symbols['system']

In [8]: print hex(system_addr)
0x7ffff7a52ec0
```

# Bonus of having libc: More Gadgets!

### Main Binary

```
(pwn) moyix@lorenzo:~/offsec/week_9/rop$ ROPgadget --binary ./rop
Gadgets information
============================================================
0x0000000000400564 : adc byte ptr [rax], ah ; jmp rax
[...]
Unique gadgets found: 70
```

### libc

```
(pwn) moyix@lorenzo:~/offsec/week_9/rop$ ROPgadget --binary ./libc-2.19.so
Gadgets information
============================================================
0x000000000018a851 : adc ah, bh ; call qword ptr [rax - 0x1f0003ec]
[...]
Unique gadgets found: 25925
```

### libc also has lots of useful strings!

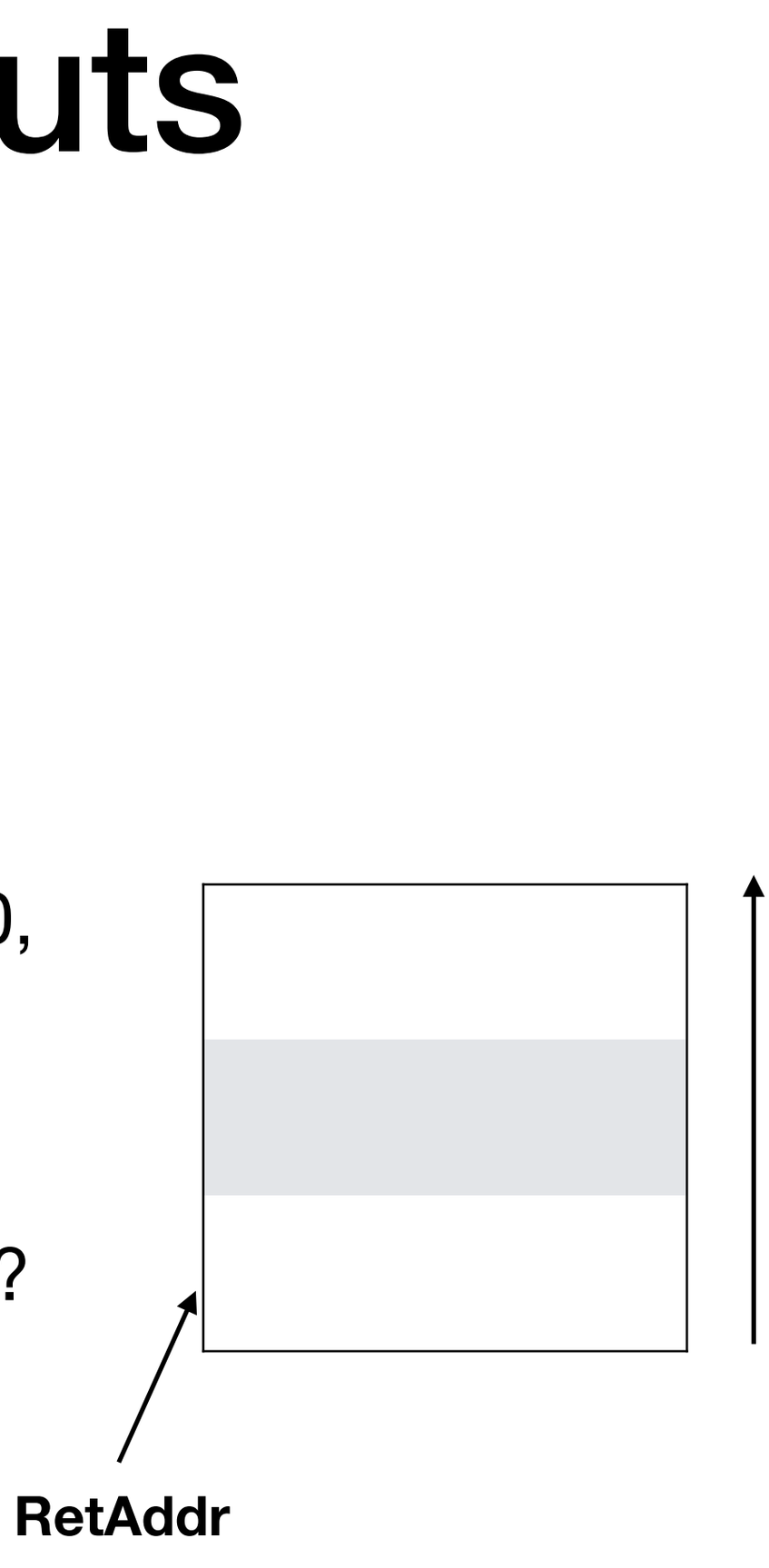
```
In [2]: libc = ELF('libc-2.19.so')

In [3]: libc.data.find('/bin/sh\x00')
Out[3]: 1574147
```

# Leaking puts

- Suppose we have a gadget
  A
  pop rdi
  ret

- The PLT entry for puts is at 0x400580, and the GOT entry for puts is at 0x601018

- What should our ROP chain look like?

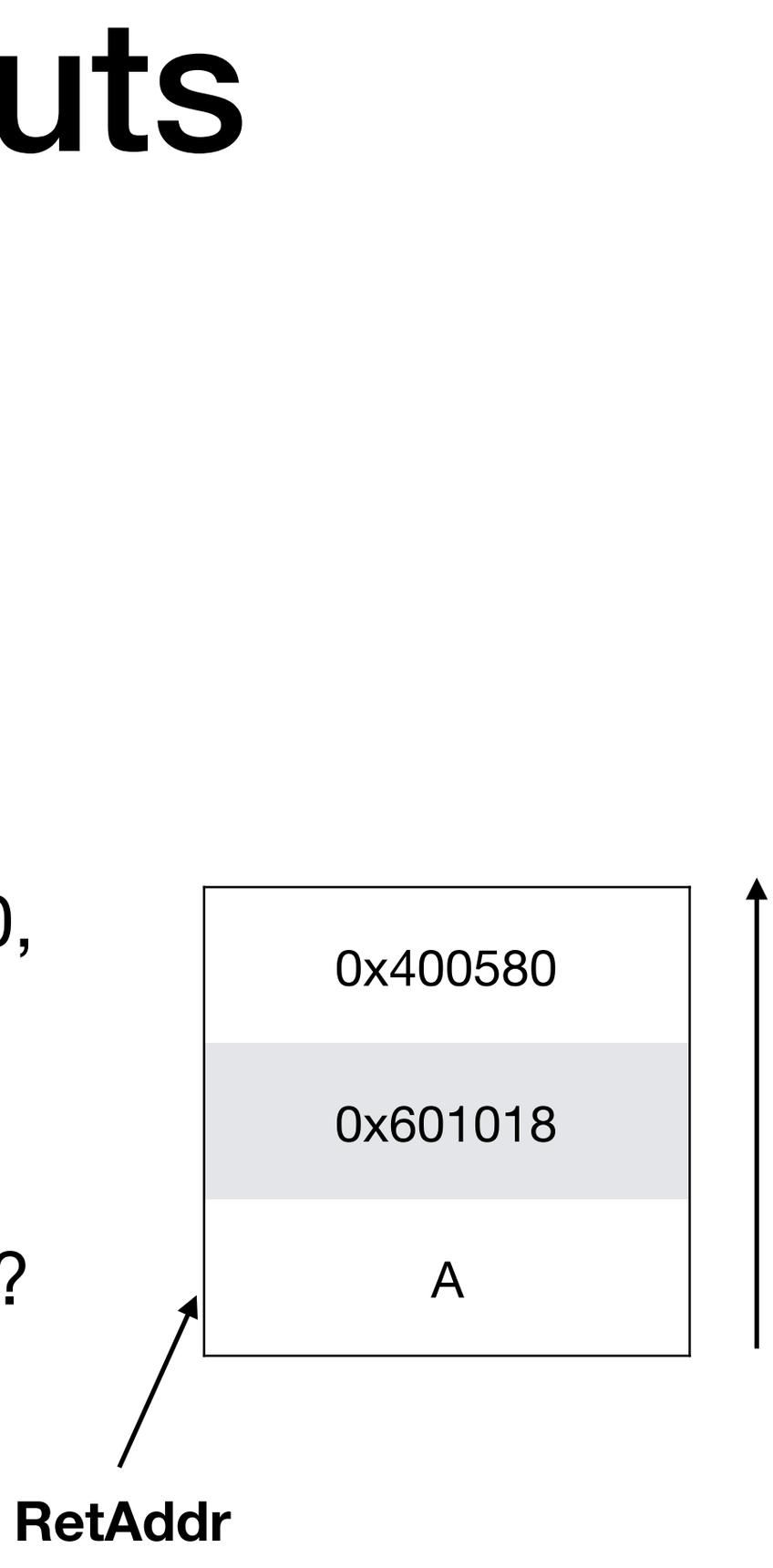


# Leaking puts

- Suppose we have a gadget
  A
  pop rdi
  ret

- The PLT entry for puts is at 0x400580, and the GOT entry for puts is at 0x601018

- What should our ROP chain look like?



# Proposed Exploit

- Last time we proposed this exploit:
  - We have full control of the stack
  - The program did gets(buf), or something similarly vulnerable
- We start our ROP…
  - puts(puts_got_address) to leak it => obtain address of puts in libc
  - We subtract the offset of puts in libc from that address to get the base address of libc
  - Then we add the offset of system in libc to get the real address of system
  - We jump there, with an argument of “/bin/sh”, and get a shell

# Problem!

- The exploit described on the last slide has a fatal flaw

- Do you see what it is?

# Problem!

- The exploit described on the last slide has a fatal flaw
- Do you see what it is?
- After we return from puts(puts_got_address) the program crashes!
- So although we can find out the address of libc, we can’t do anything with it
- Next time we run the program, the address of libc will be different – so our knowledge is useless!

# Solution: Return to main

- After our initial ROP chain that leaks the address of puts, we add the address of the main() function

- When the program finishes printing out the address, it will jump to the beginning of main... effectively restarting the program (but *without* changing where libc is)

- Now we can **re-exploit the program**

    - And we can make use of the libc addresses we learned the first time around!

# Advanced ROP

- You can’t always get what you want
- Sometimes your binary won’t have all the gadgets you need
- Example: need to do a 3 argument syscall, but we only have
  pop rdi; ret
  pop rsi; ret
  syscall; ret
- How can we get something into rdx?

# Some Ways to Set RDX

- Is there a mov rdx, ___ and a way to set ___ ?
    - E.g. mov rdx, rsi since we already have a pop rsi
- What is rdx at the point where our ROP chain starts?
    - Can we influence it?
- Can we do arithmetic in the ROP chain?
    - add, xor, inc, etc gadgets
    - Last resort, but possible for "nice" numbers or numbers near rdx

# Getting Useful Strings and Constants

- We often need to get some useful string or constant into a known location (classic example: “/bin/sh\x00”)
    - May not have a libc leak
    - May not have a stack leak
- Is there some way we can put our own data into a place that has some writable memory at a fixed address?

# The .bss and .data Sections

- Every program will have a section of the executable dedicated to storing global variables
- Initialized data goes into .data, uninitialized data goes into .bss
- Recall: operating systems allocate memory in units of a *page* (4096 bytes on x86)
- This means there is often “slack” space at the end of one of these sections!

# .bss Example

```
.data (PROGBITS) section started  {0x601050-0x601060}
00601050  __data_start:
00601050  00 00 00 00 00 00 00 00                          ........
00601058  __dso_handle:
00601058                          00 00 00 00 00 00 00 00           ........
.data (PROGBITS) section ended  {0x601050-0x601060}
```

---

```
.bss (NOBITS) section started  {0x601060-0x601090}
00601060  stdout:
00601060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00601070  stdin:
00601070  00 00 00 00 00 00 00 00                          ........
00601078  completed.7585:
00601078                          00 00 00 00 00 00 00 00           ........
00601080  buf:
00601080  00 00 00 00 00 00 00 00                          ........

00601088  int64_t data_601088 = 0x0
.bss (NOBITS) section ended  {0x601060-0x601090}
```

- Section “ends” at 0x601090
- ...but the rest of the page is mapped and writable
- So we have 0x601090-0x602000 for our own data

# Reading Data

- How do we actually put data there?
- ROP, of course!
- Assuming the program takes input, it must have a function imported like fgets, gets, or read
- So we can create a ROP chain that calls that function to read data into some space in the .bss
- Since the .bss is at a fixed location, this lets us put whatever data we want and know where it is

# Stack Pivots

- We saw one way (return to main) that you can make use of information you got from a ROP chain to continue your exploit

- Another way is a *stack pivot*:
  - We can call read() or similar to write a *second ROP chain* somewhere in the .bss
  - Then find a stack pivot gadget that allows us to **set the value of rsp** to point to our new ROP chain
    0x4006ad : **pop rsp** ; pop r13 ; pop r14 ; pop r15 ; ret
  - Everything after the **pop rsp** will happen on our **new stack**

# Stack Pivot Warning

- The .bss section is not that large
- And it’s where your stack is now
- As a result, calling complicated functions that use a lot of stack space could make your program crash
  - Or start overwriting your ROP chain
- Best to keep things simple and avoid using the stack too much!

# One Last Trick

- If you have figured out where libc is but don’t want to (or can’t) construct a ROP chain to call system(“/bin/sh”)
- It turns out there are places within libc that if you jump to them will do the work of executing a shell for you!
    - In particular, partway through the system() function
- There are tools to find these locations, like one_gadget https://github.com/david942j/one_gadget

# one_gadget

```
david942j at ~/one_gadget on master via 💎   v2.5.3
➔ one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rcx == NULL

0x4f322 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a38c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL

david942j at ~/one_gadget on master via 💎   v2.5.3
➔
```