



General

FA2026 • 2026-10-11

PWN II

Krish Kalra

ctf.sigpwny.com

sigpwny{AAAA.%p.%p.%s.%s.%p.%p}



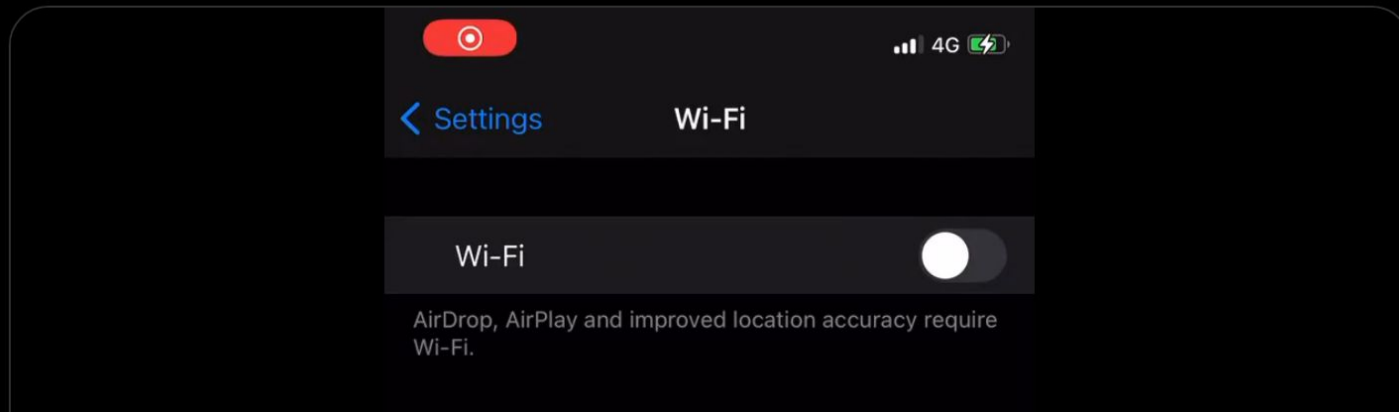
```
44 PROT_READ = 1
45 PROT_WRITE = 2
46 MAP_PRIVATE = 2
47 MAP_ANONYMOUS = 0x20
48 mmap_syscall = 0x09
49 def syscall(syscall_num, rdi, rsi, rdx, r10, r8, r9):
50     s(rdi)
51     s(rsi)
52     s(rdx)
53     s(r10)
54     s(r8)
55     s(r9)
56     s(syscall_num)
57
58 syscall(mmap_syscall, 0x20000, 4096, PROT_READ | PROT_WRITE,
59 ptr = int(sh.recvuntil('another').split(b'\n')[1].split(b'
60 print(hex(ptr))
61 s(1)
62
63
64
65
66 read_syscall = 0
67 bin_sh = '/bin/sh'
68
69 syscall(read_syscall, 0, ptr, len(bin_sh), 0, 0, 0)
70 sh.interactive()
71
```



Carl Schou

@vm_call

After joining my personal WiFi with the SSID “%p%s%s%s%s%n”, my iPhone permanently disabled it’s WiFi functionality. Neither rebooting nor changing SSID fixes it :~)



Review: PWN I

- Buffers and variables are stored on the stack, at a fixed size, contiguous in memory.
- Unsafe functions can write more data than the buffer can store, leading to **Buffer Overflow** Vulnerabilities.
- We can control the program flow by overflowing a local **stack variable** to overwrite the **return address**.



Shellcode

- Shellcode is a term for bytes of executable instructions that we plan to run.
- You can write your own, or google existing exploits
- <https://www.exploit-db.com/exploits/47008>
- Search for "x86_64 Linux Shellcode"
- This one opens a shell, but you can do anything, like allocate memory, open and write to files, etc.

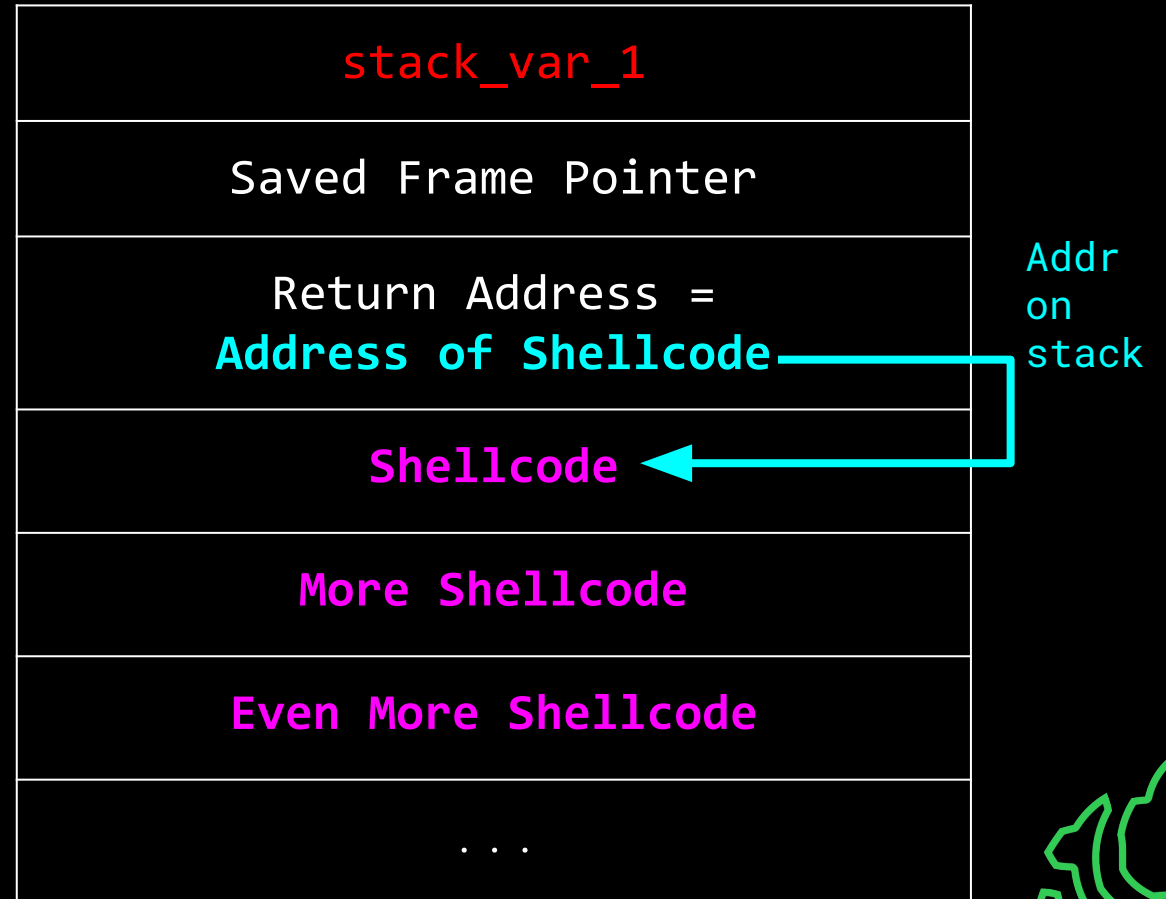
```
mov    eax, 32
xor     eax, eax
push   eax
pop     ebx
call   mysuperfunc
int     0x80
```



Shellcode

```
int vulnerable() {  
    puts("Say Something!\n");  
    char stack_var_1[8];  
    gets(stack_var_1);  
    return 0;  
}
```

```
> ./vulnerable  
Say Something!  
AAAAAAAABBBBBBBB  
{addr on stack}  
{shellcode}
```



Problem: in order to jump to our shellcode on the stack, we need an address of something on the stack!

Mitigation: NX (No-eXecute)

- ret2shellcode only works if you have permissions to both
 - **W**rite to the memory region
 - **eX**ecute the memory region
- Solution: memory should be **W^X** a.k.a **W**rite XOR **eX**ecute
 - Aside: when might we want both?
- The stack is given **RW** permissions, but never **X**.
 - Back in the day, this was not considered, and the stack was executable!



Virtual Memory Protections

- You will learn in CS233 or ECE391 about Virtual Memory and how it is handled
- For our purposes, understand that program data, program globals, stack, heap are all uniquely allocated sections
- The stack (with NX) has **RW-** perms
- The heap also has **RW-**
- Program Data has **R-X**
- Static Globals has **R--**
- Is there ever write-only perms?

LEGEND: STACK HEAP CODE DATA WX RODATA					
Start	End	Perm	Size	Offset	File
0x55555554000	0x55555555000	r--p	1000	0	/home/surg/CTF/csaw/vipblacklist/vip_blacklist
0x55555555000	0x55555556000	r-xp	1000	1000	/home/surg/CTF/csaw/vipblacklist/vip_blacklist
0x55555556000	0x55555557000	r--p	1000	2000	/home/surg/CTF/csaw/vipblacklist/vip_blacklist
0x55555557000	0x55555558000	r--p	1000	2000	/home/surg/CTF/csaw/vipblacklist/vip_blacklist
0x55555558000	0x55555559000	rw-p	1000	3000	/home/surg/CTF/csaw/vipblacklist/vip_blacklist
0x7ffff7c00000	0x7ffff7c28000	r--p	28000	0	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7c28000	0x7ffff7dbd000	r-xp	195000	28000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7dbd000	0x7ffff7e15000	r--p	58000	1bd000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e15000	0x7ffff7e16000	---p	1000	215000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e16000	0x7ffff7e1a000	r--p	4000	215000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e1a000	0x7ffff7e1c000	rw-p	2000	219000	/usr/lib/x86_64-linux-gnu/libc.so.6
0x7ffff7e1c000	0x7ffff7e29000	rw-p	d000	0	[anon_7ffff7e1c]
0x7ffff7fa0000	0x7ffff7fa3000	rw-p	3000	0	[anon_7ffff7fa0]
0x7ffff7fb0000	0x7ffff7fbd000	rw-p	2000	0	[anon_7ffff7fbb]
0x7ffff7fbd000	0x7ffff7fc1000	r--p	4000	0	[vvar]
0x7ffff7fc1000	0x7ffff7fc3000	r-xp	2000	0	[vdso]
0x7ffff7fc3000	0x7ffff7fc5000	r--p	2000	0	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fc5000	0x7ffff7fef000	r-xp	2a000	2000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fef000	0x7ffff7ffa000	r--p	b000	2c000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ffb000	0x7ffff7ffd000	r--p	2000	37000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ffd000	0x7ffff7fff000	rw-p	2000	39000	/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fff000	0x7ffff7fff000	rw-p	21000	0	[stack]
0xffffffff600000	0xffffffff601000	--xp	1000	0	[vsyscall]

Mitigation: Stack Canary

- A randomly generated number placed before **return address**
- Canary value verified before returning, **crashing** if modified.

```
int vulnerable() {  
    puts("Say Something!\n");  
    char stack_var_1[4];  
    gets(stack_var_1);  
    if (rbp+8 != r15){  
        __stack_chk_fail();  
    }  
    return 0;  
}
```



Problem: how do we leak the stack canary to bypass this check?



Mitigation: ASLR + PIE

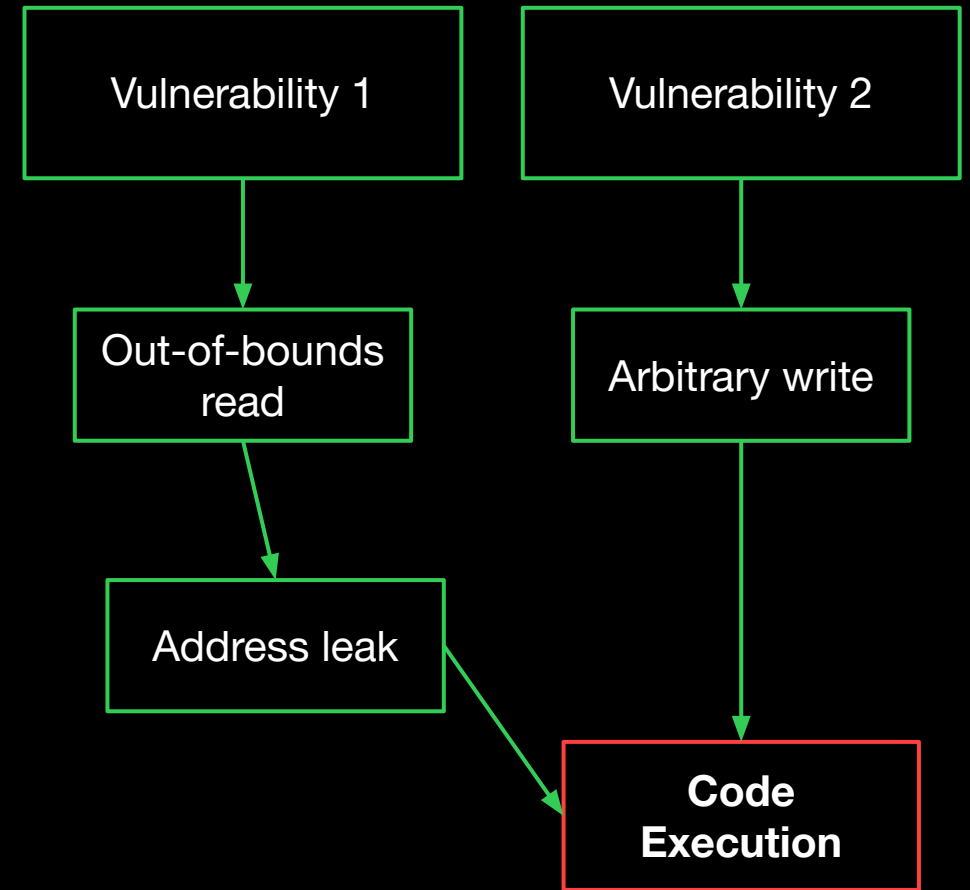
- Address Space Layout Randomization
- Position Independent Executable
- Without PIE, our code is loaded at a fixed address (traditionally `0x400000`).
- With PIE, our code only uses relative offsets.
- Now we can use ASLR, loading our code to a new random address every time.
 - e.g. first load: `0x551234`
 - e.g. second load: `0x559878`

Problem: how do we jump to a function if its absolute address keeps changing?



Exploit Primitives

- “Building blocks” of an exploit
- Read
 - Arbitrary (read anywhere)
 - Uncontrolled (read starting from some address)
- Write
 - Arbitrary (write anything anywhere)
 - Uncontrolled (write something anywhere)
 - Also uncontrolled (write anything somewhere)
- Leak
 - Usually done with a read, but not always
 - Useful when addresses are **randomized**



Exploit Primitives

- In PWN I, we had uncontrolled write with buffer overflow
- Now, we will give you binaries with ASLR/PIE/Canary/NX
- We will use arbitrary reads to leak information so we can:
 - Jump to a randomized (on run) location of memory
 - Keep the Canary intact
 - Use executable code wherever allowed



Bypassing Mitigations

- To bypass NX, we have to return to executable memory:
 - Code in the standard library (libc)
 - The target program itself
- To bypass Stack Canary, we need to **leak** stack memory to learn the canary's value.
- To bypass ASLR/PIE, we need to **leak** a pointer to program or stack memory
 - then, we can infer the randomized offset
 - $\text{offset} = \text{leak} - \text{base}$



Dangerous Function of the Day: `printf()`

- **Formatted** print function
- `printf("Hello %s!", "Kevin");`
 - Hello Kevin!
- `printf("My favorite number is %d", 1337);`
 - My favorite number is 1337
- `printf("%s, my favorite number is %d", "Kevin", 1337);`
 - Kevin, my favorite number is 1337
- `%s` and `%d` are **format specifiers**
 - Tells the function to read the next argument as a certain data type
 - `%s` -> string, `%d` -> decimal integer, `%p` -> pointer, etc.



Dangerous Function of the Day: `printf()`

- How might this go wrong?
- `printf("%d", "Kevin");` // prints 1302429700
 - Wrong specifier misinterprets the argument
 - In this case, it's the address of the string "Kevin"
- `printf("%d");` // prints 1397277592
 - Too few arguments are actually provided
 - But `printf` doesn't know that, so it reads them anyways



Dangerous Function of the Day: `printf()`

- What if it's just used as a print function?
- `printf(name)` // name is controlled by the user
- If name is 'Kevin', prints 'Kevin'
- If name is '%s', prints...
- Format specifiers follows the prototype:
 - %[flags][width][.precision][length]specifier



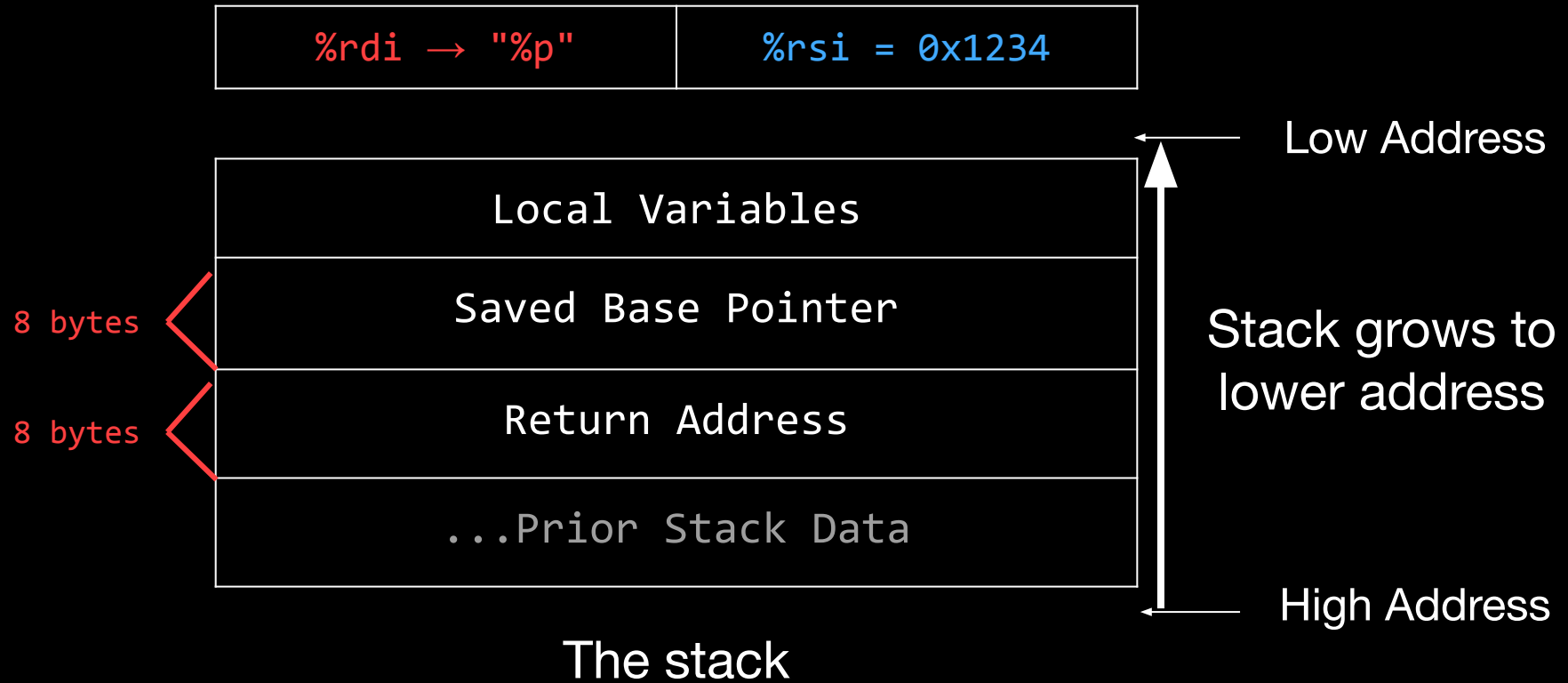
Primitive: Stack Read

- `%p` 'pointer' format specifier
 - `printf("%p", 0x13371337);`
 - Prints '0x13371337'
- `printf("%p");`
 - What happens now?



Review: Calling Functions

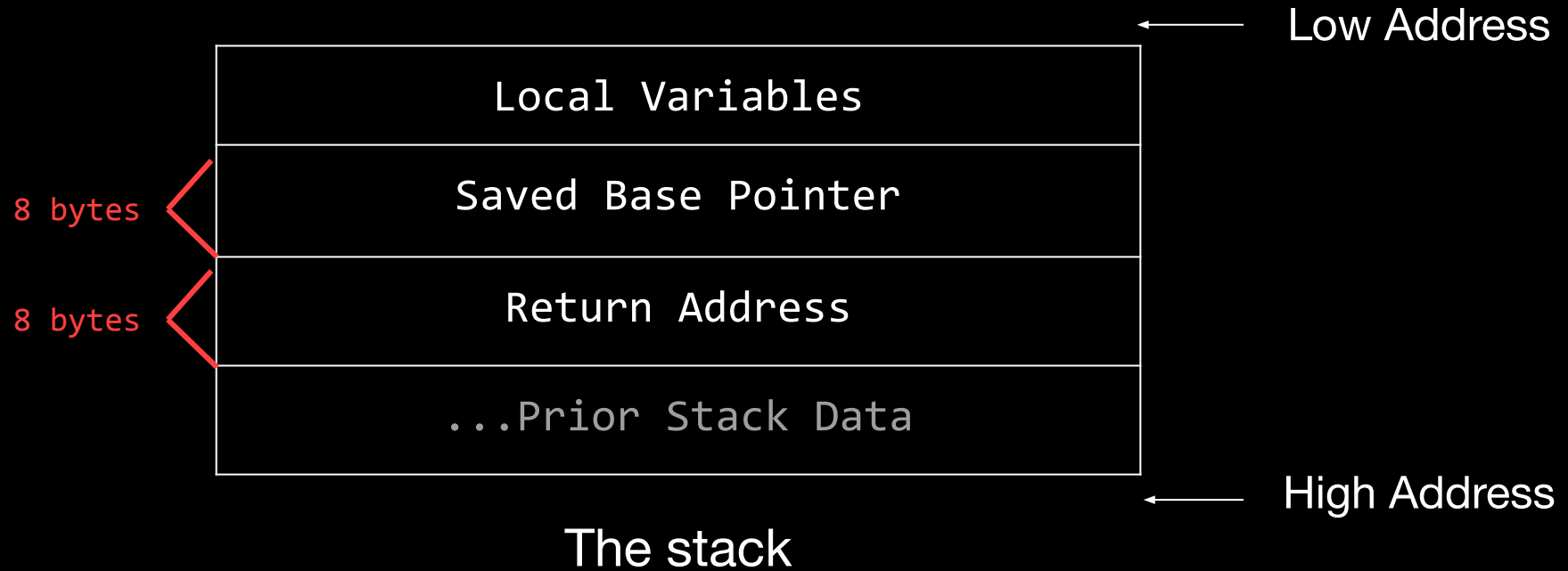
```
printf("%p", 0x1234);
```



New: Calling Functions

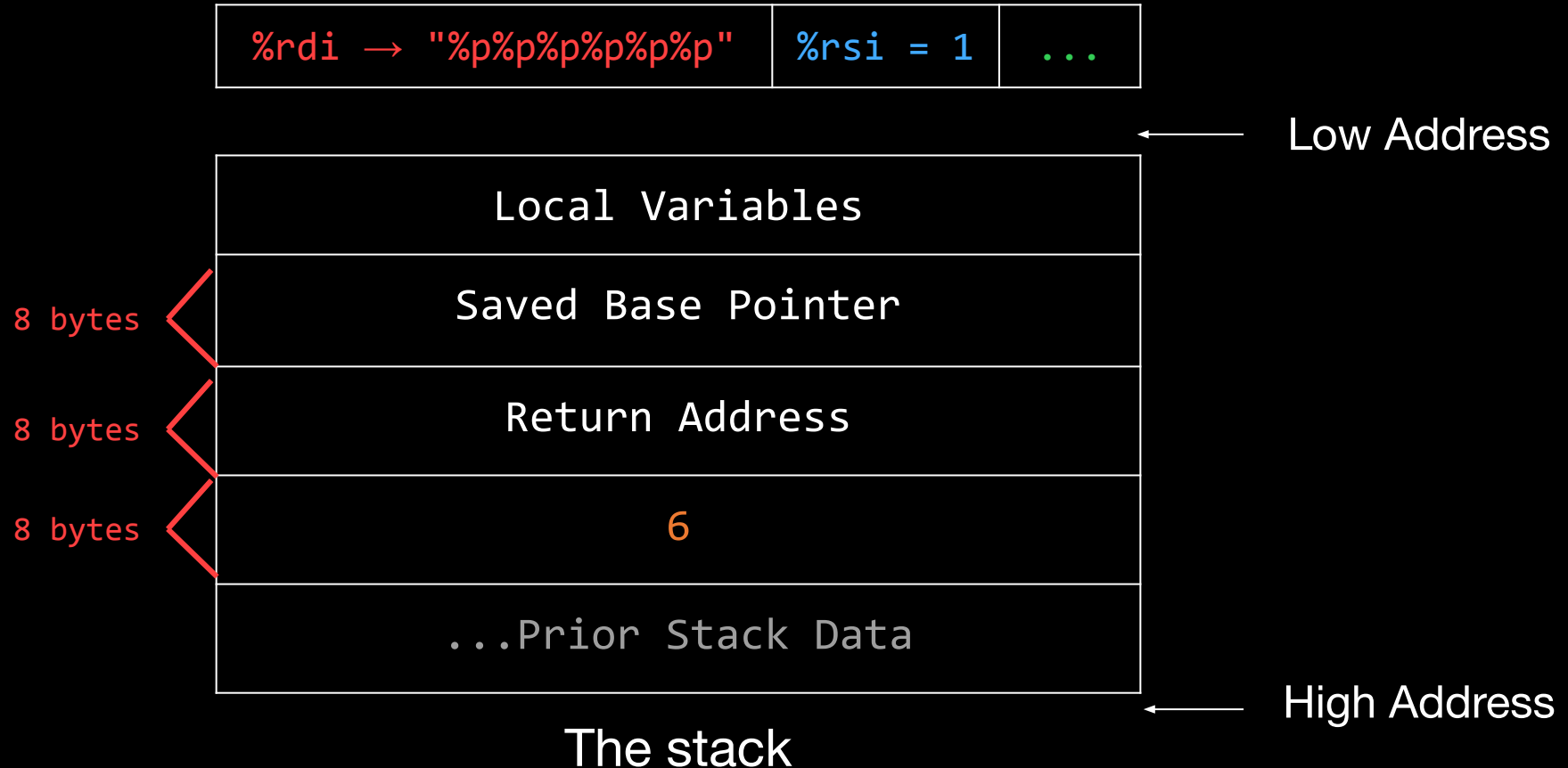
```
printf("%p%p%p%p%p", 1, ..., 5);
```

<code>%rdi</code> → <code>"%p%p%p%p%p"</code>	<code>%rsi</code> = 1	...	<code>%r9</code> = 5
---	-----------------------	-----	----------------------



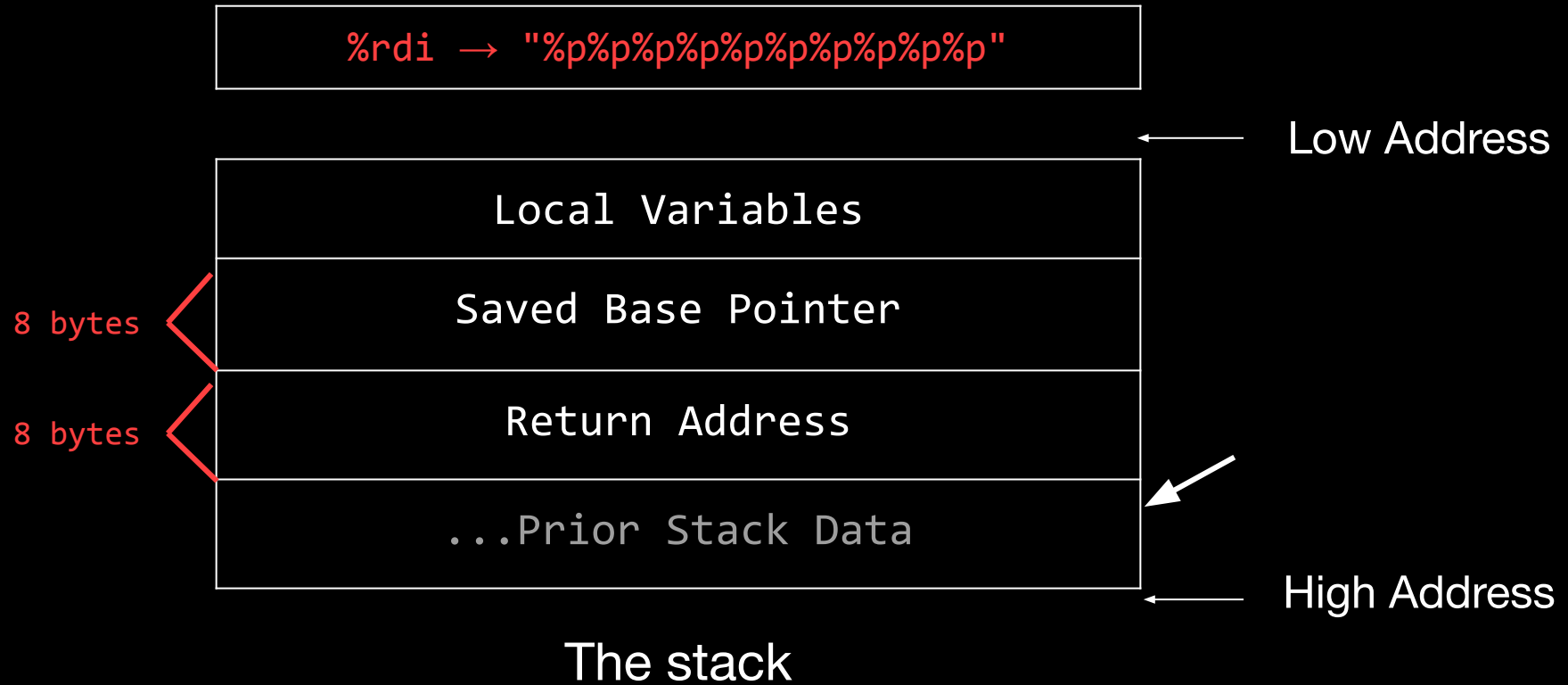
New: Calling Functions

```
printf("%p%p%p%p%p%p", 1, ..., 6);
```



printf Exploitation

```
printf( "%p%p%p%p%p%p%p%p%p" );
```



Primitive: Stack Read

- `%p` format specifier
 - `printf("%p", 0x13371337);`
 - Prints `'0x13371337'`
- `printf("%p");`
 - Whatever is next in arguments, eventually stack memory!
 - `printf("%p %p %p %p %p %p %p %p");`
 - Prints out some registers and stack memory, 8 bytes at a time
 - Figure out which data is the thing you want :)
 - If the string `'sigpwny{'` were on the stack, you might see:
 - `0x7b796e7770676973`
 - These are **hexadecimal ASCII values**, online converters may be useful
- Note:
 - `%p` interprets data as **little endian**



Primitive: Arbitrary Read

- `%s` format specifier
 - `printf("%s", "hello");`
 - Prints 'hello'
 - `printf("%s", 0x12345678);`
 - Prints the string starting from memory address `0x12345678`
 - `printf("%3$s", 0x100, 0x200, 0x300);`
 - Prints the string starting from memory address `0x300` (3rd argument)



Primitive: Arbitrary Read

- `char name[64]; // stored on stack`
- `fgets(name, 64, stdin); // '%n$p' <- n is a number`
- `printf(name);`
- For some `n`, the `%n$p` will print `name`!
 - E.g. `0x70243525`
- Key idea:
 - Format specifiers can read from the stack, and `name` is on the stack
 - Format specifiers can reference our input!
- If `name` is `'%n$s'` (for correct `n`)
 - Prints the string starting from a memory address **in our input**



Primitive: Arbitrary Read

- `char name[64]; // stored on stack`
- `fgets(name, 64, stdin);`
- `printf(name);`
- If `name` is `'%n$s_____\x11\x22\33\x44\x55\x66\x77\x88'` (for correct `n`)
 - Prints the string starting from memory address `0x8877665544332211`
 - We can read from memory addresses contained **in our input**
- Note: why the underscores?
 - Each argument is 8 bytes: `len('%n$s_____') == 8`, so the address is aligned correctly. **Pad to a multiple of 8 bytes before the address.**
- Testing strategy:
 - Develop with `%n$p` instead of `%n$s` and verify the correct address gets printed
 - Then switching to `%s` will make it read from the correct address!



Primitive: Arbitrary Write

- `%n` format specifier
 - Writes the number of bytes previously printed to the given address
- `printf("%n", &number);`
 - `number = 0;`
- `printf("AAAA%n", &number);`
 - `number = 4;`
- `printf("%500p%n", 1, &number);`
 - `number = 500;`
 - ‘%500p’ means format as pointer, padding to 500 characters
 - In this case, ‘0x1’ preceded by 497 spaces
 - Easy way to print a given number of bytes



Primitive: Arbitrary Write

- Testing strategy:
 - Develop with `%n$p` instead of `%n$n` and verify the correct address is printed
 - Then switching to `%n` will make it write to the correct address!
- Note: by default, `%n` writes 4 bytes
 - "h" is a size specifier flag
 - `%hn` writes 2 bytes, `%hhn` writes 1 byte



Libc

- Libc is a program that is loaded at the same time as your program, which hold the *standard library*
- If we get a leak to libc, we get access to many powerful functions we can control or strings (e.g. “/bin/sh”)
- The GOT (Global Offset Table) contains addresses to libc functions
- The GOT is writable! (e.g. puts->system)



one_gadget

- There is a tool called one_gadget, which given a binary, finds a location which will call `execve('/bin/sh/', ?, ?)`
- A method to pop a shell as a 'win function' (useful when NX is on)
- Provided that the register constraints are met, there are several positions in libc that we can return to.

```
srg@pop-os:~/CTF/defcamp/bistro2$ one_gadget libc-2.27.so
0x4f2a5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL

0x4f302 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a2fc execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```



Challenges!

- Format 0-3 + Quiz acts as a primer for using specifiers
- 3 - Execute and 4 - Format are pure pwnables covering no-NX and format
- Libc ROP may need one_gadget to solve



Next Meetings

2026-10-15 • This Thursday

- Jails
- Learn break out of sandboxes and restricted environments!

2026-10-18 • Next Sunday

- SIGPwny X SIGArch



ctf.sigpwny.com

sigpwny{AAAA.%p.%p.%s.%s.%p.%p}

Meeting content can be found at
sigpwny.com/meetings.

