SIGPwny

FA2024 Week 09 • 2024-11-03

PWN II

Sam Ruggerio and Jake Mayer

Announcements

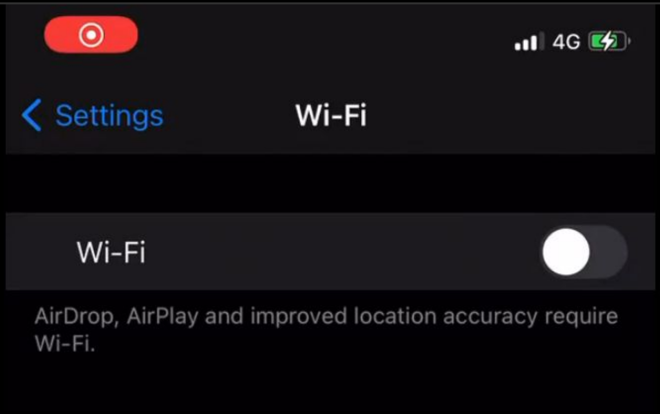
-

ctf.sigpwny.com

sigpwny{%200c%n%15$p%+d}

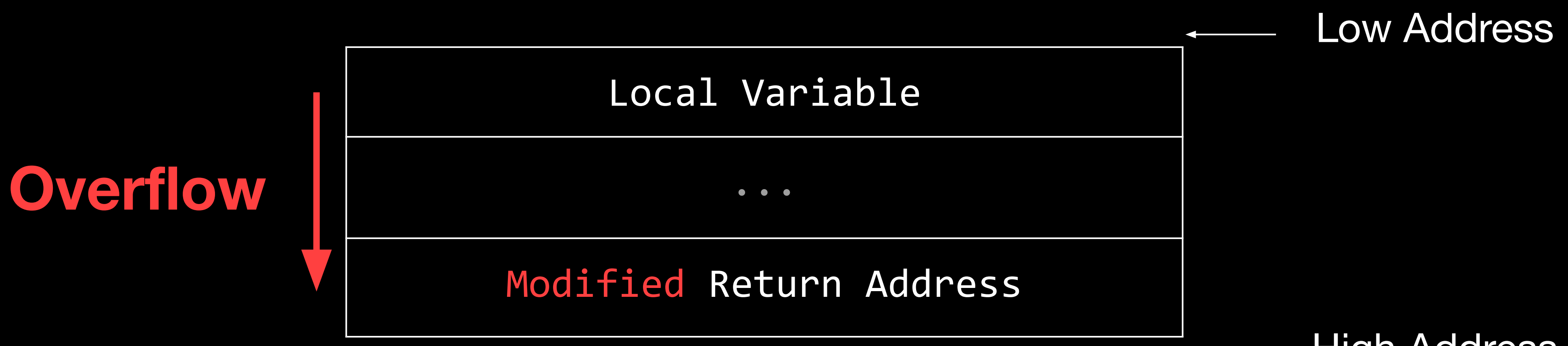
Carl Schou
@vm_call

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 the buffer (`local 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

```c
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}
```

stack_var_1
Saved Frame Pointer
Return Address = **Address of Shellcode**
Shellcode
More Shellcode
Even More Shellcode
. . .

Addr on stack → Shellcode

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

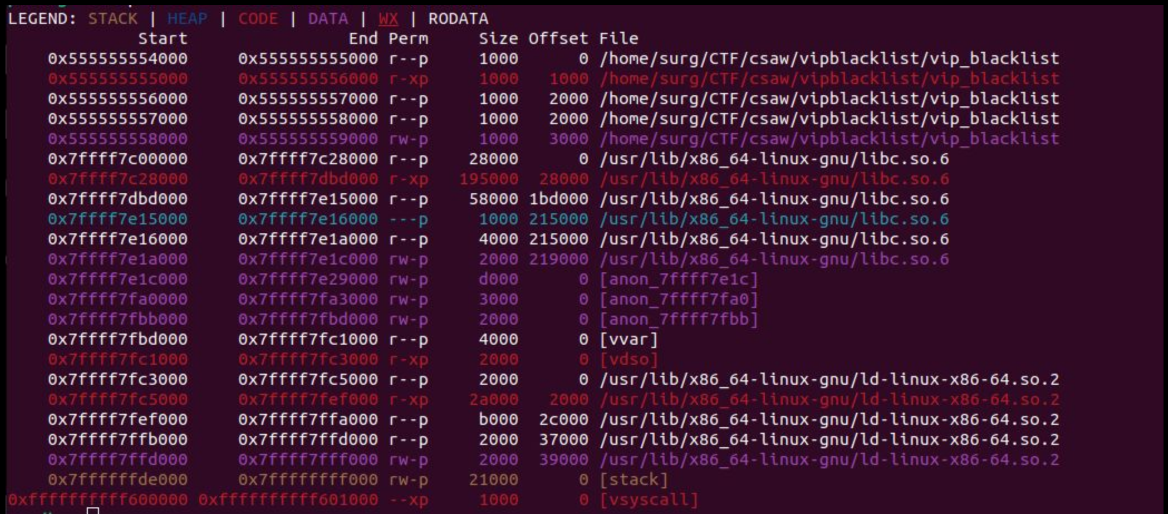
Mitigation: NX

- ret2shellcode only works if you have permissions to both
  - Write to the memory region
  - Execute the memory region
- There is a philosophy of how to manage memory regions: `W^X` a.k.a **W**rite XOR e**X**ecute
- In modern complication, 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

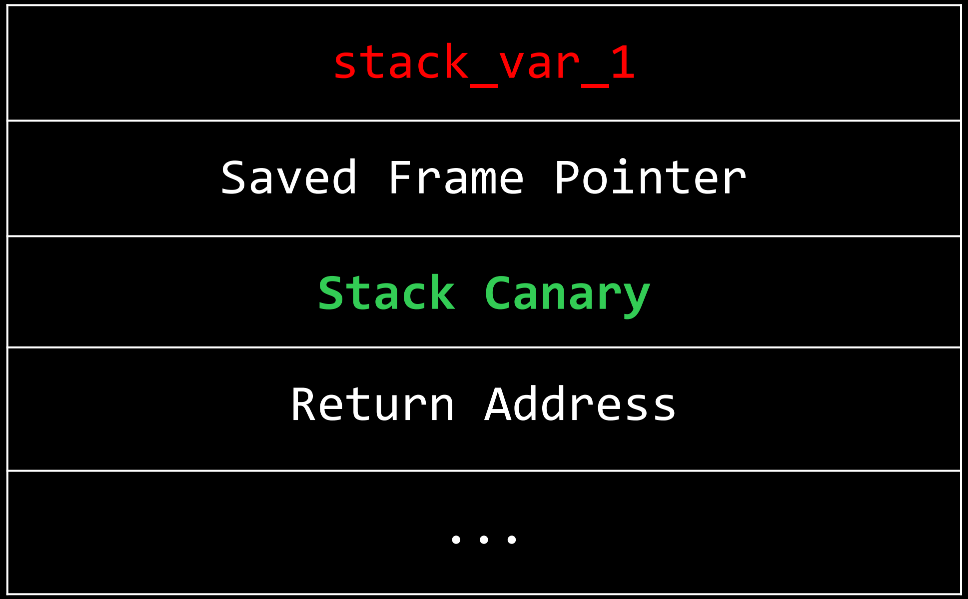
- The stack (with NX) has `RW-` perms
- The heap also has `RW-`
- Program Data has `R-X`
- Static Globals has `R--`
- Is there ever write-only perms?



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

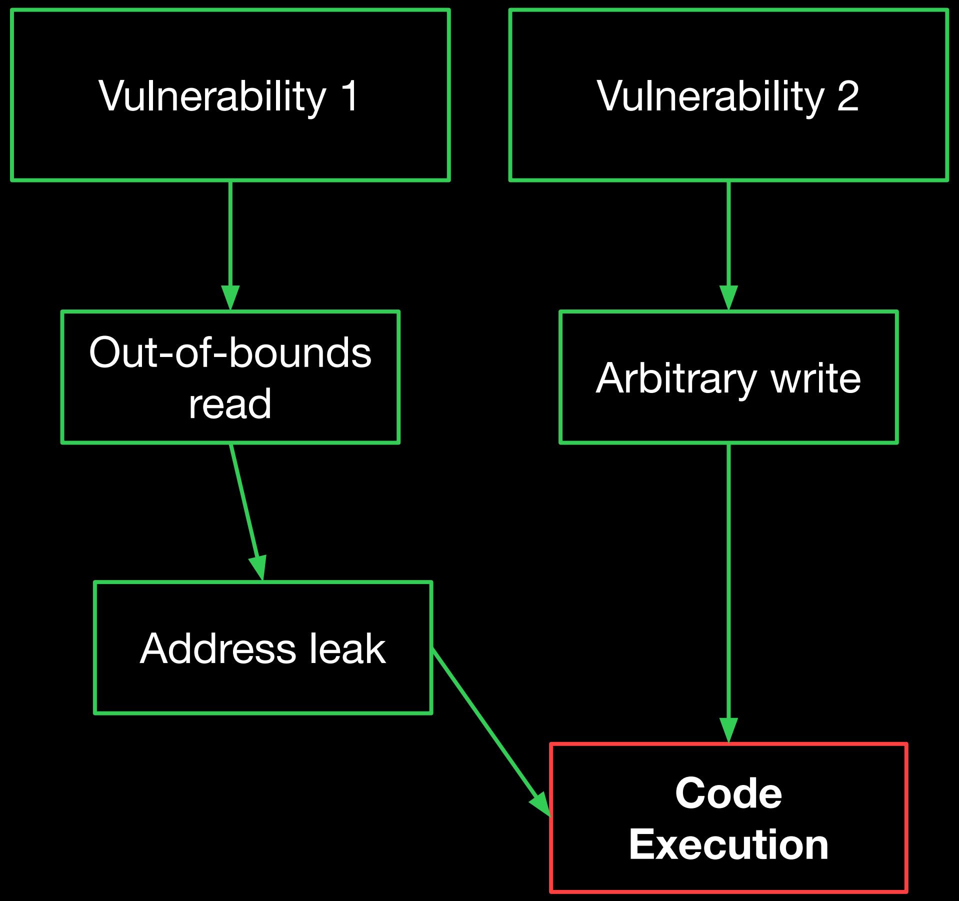
- **A**ddress **S**pace **L**ayout **R**andomization
- **P**osition **I**ndependent **E**xecutable

- 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
- Common primitives
  - 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
    - Necessary because addresses are often **randomized**



Exploit Primitives

- In PWN I, we had arbitrary/uncontrolled write with buffer overflow
- Now, we will give you binaries with ASLR/PIE/Canary/NX
- We now need 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
  - `offset = leak - base`

Dangerous Function of the Day: printf()

- **Formatted** print function
  - `printf("Hello %s!", "Kevin"); // prints '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.
- What if it's just used as a print function?
  - `printf(name) // name is controlled by the user`
  - If name is 'Kevin', prints 'Kevin'

Dangerous Function of the Day: `printf()`

- **Formatted** print function, Variadic
  - `printf("Hello %s!", "Kevin"); // prints ‘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.`
- What if it’s just used as a print function?
  - `printf(name) // name is controlled by the user`
  - If name is ‘`%s`’, prints…

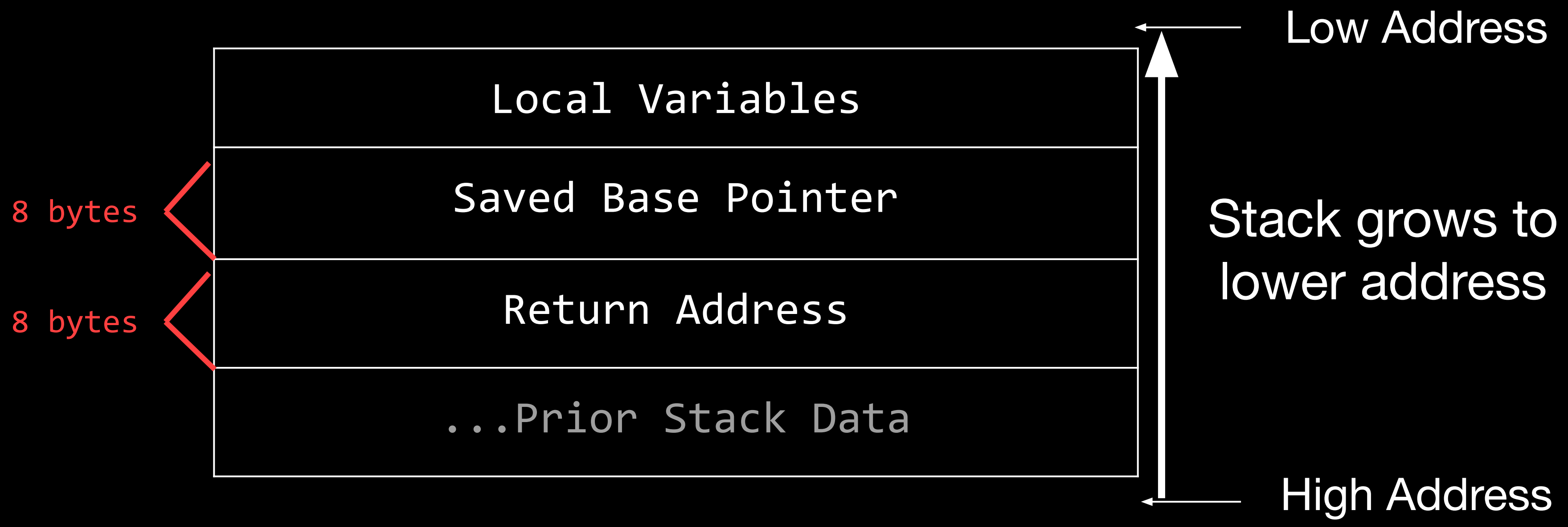
Primitive: Stack Read

- %p ‘pointer’ format specifier
  - `printf("%p", 0x13371337);`
    - Prints ‘`0x13371337`’
- `printf("%p");`

Review: Calling Functions

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

%rdi → "%p"	%rsi = 0x1234

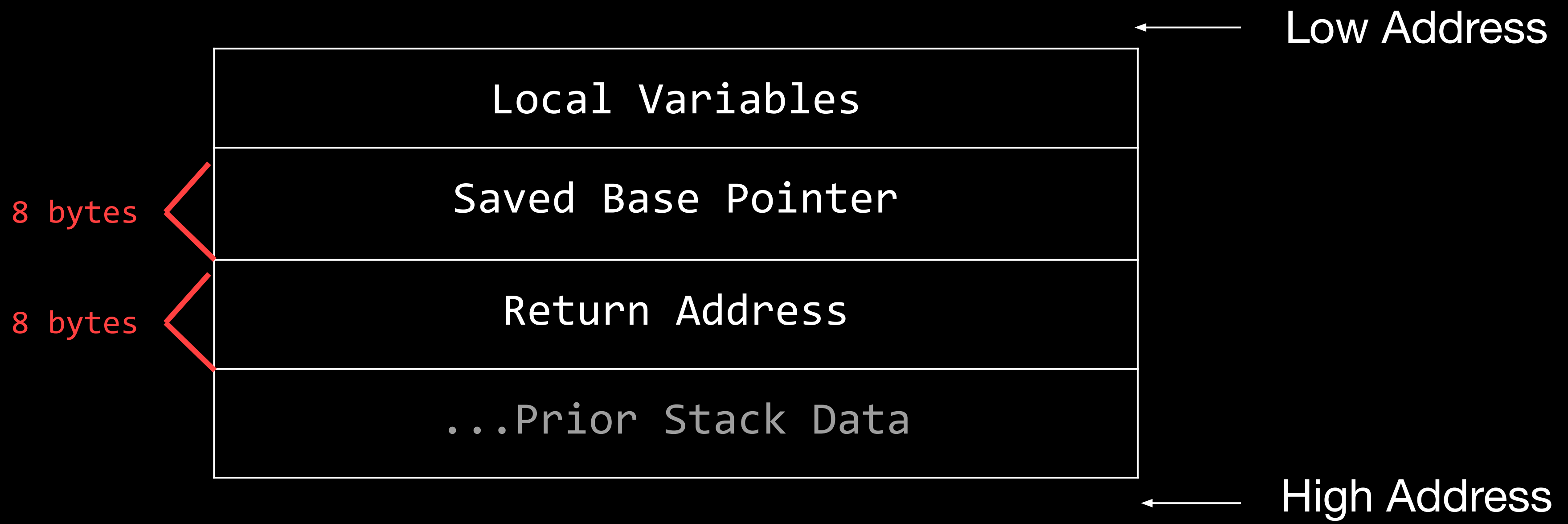


The stack

New: Calling Functions

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

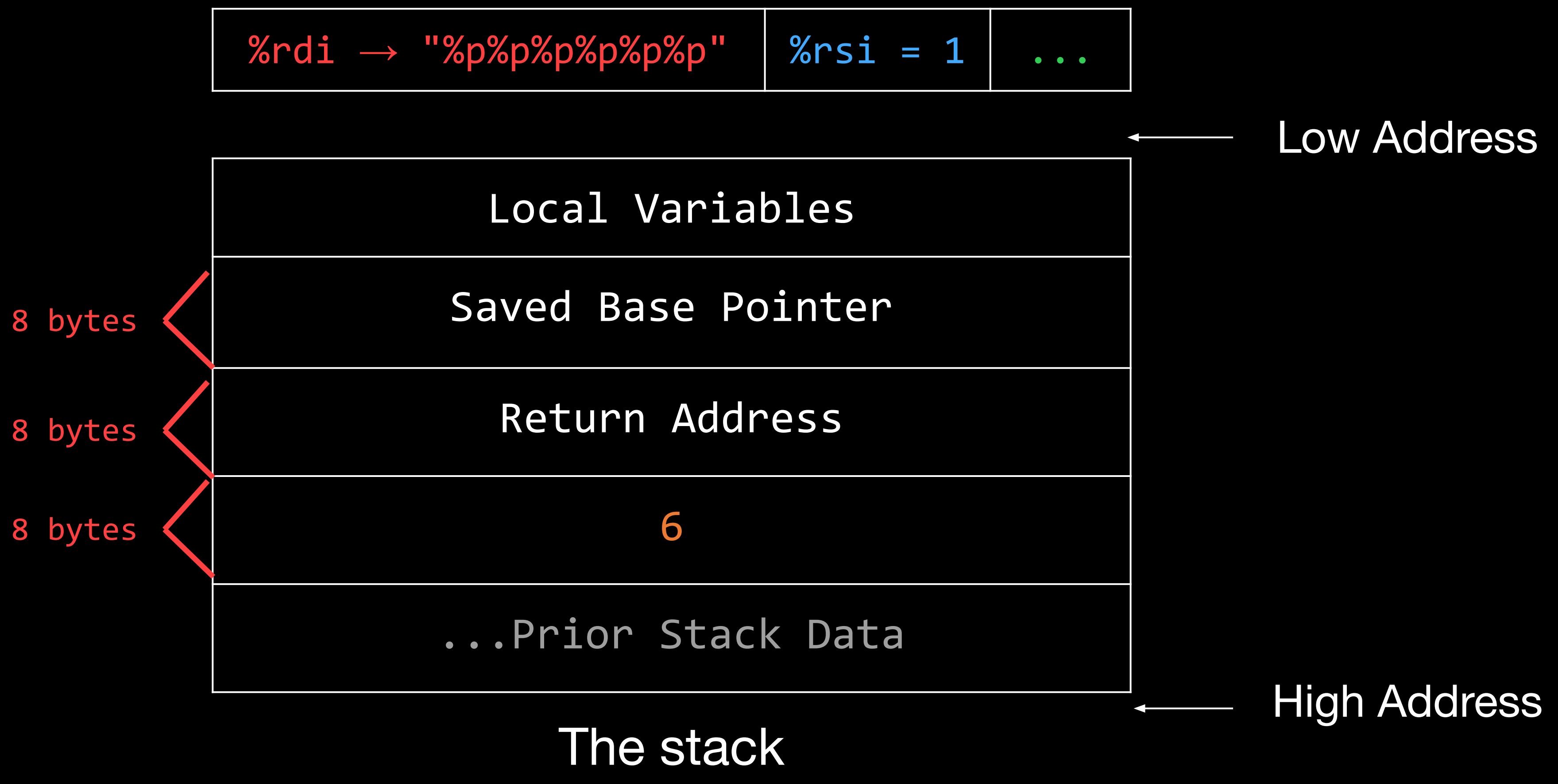
%rdi → "%p%p%p%p%p"	%rsi = 1	...	%r9 = 5



The stack

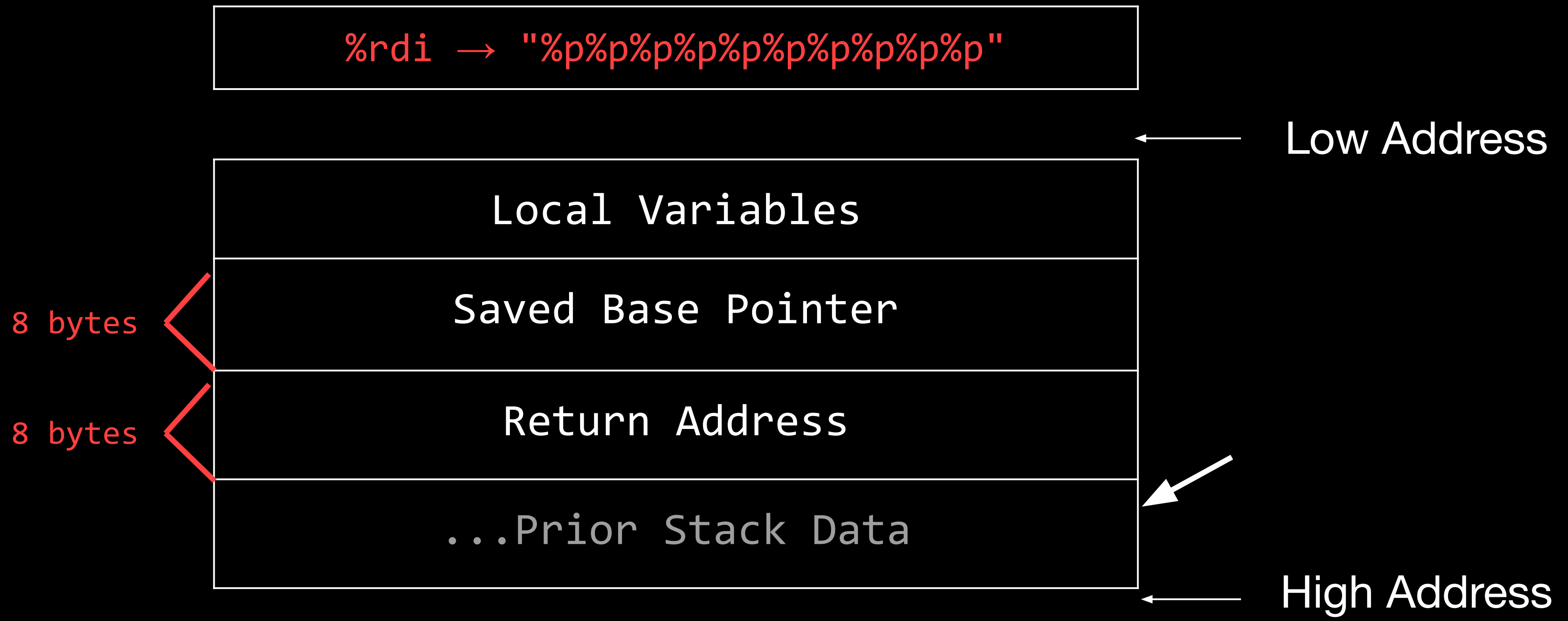
New: Calling Functions

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



printf Exploitation

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



The stack

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");`
    - 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

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’ if 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
```

Bistro Demo

Next Meetings

2024-11-07 • This Thursday

- Pyjails with Cameron and Louis
- Escape limited Python environments

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

ctf.sigpwny.com

sigpwny{%200c%n%15$p%+d}

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

SIGPwny