Intro to Binary Exploitation

Lucas Polidori - lucas.polidori@tufts.edu

Agenda

- Project Setup
- Challenges:
  - Challenge 1: Basic Buffer Overflow
  - Challenge 2: ret2win
  - Challenge 2a: Intro to ROP
  - Challenge 3: ret2libc
- Concluding Notes

Project Setup:

- Log into Halligan
- Enter these two commands:
  - `git clone https://github.com/Polidori-112/Ieee_Lab`
  - `cd Ieee_Lab`

Challenge 1: Basic Buffer Overflow

What chal1 Does:

- Initiates long int
- Receives Input
- Checks if int was manipulated
- INPUT DOES NOT AFFECT INT

```
18 int main() {
19
20     char buf[20];
21     long int secret_number = 0;
22
23
24     printf("What is your name?\n");
25     gets(buf);
26
27     printf("Hi %s, can you figure out how to win?\n", buf);
28
29     if (secret_number == 42) {
30         printf("Congrats, you found it!!\n");
31         win();
32     }
33
34     else if (secret_number != 0) {
35         printf("Huh, how did you do that? My secret number is now %ld (0x%lx)\n",
36     }
37
38     return 0;
39 }
40
```

Challenge 1: Basic Buffer Overflow

The Bug: gets(buf)

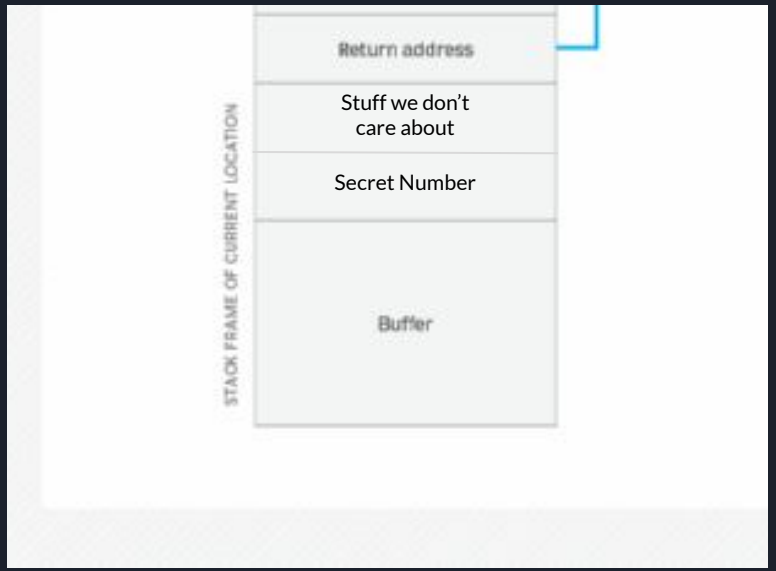
- Receives Input
- Places input on stack
- Does not sanity check input

```
18 int main() {
19
20     char buf[20];
21     long int secret_number = 0;
22
23
24     printf("What is your name?\n");
25     gets(buf);
26
27     printf("Hi %s, can you figure out how to win?\n", buf);
28
29     if (secret_number == 42) {
30         printf("Congrats, you found it!!\n");
31         win();
32     }
33
34     else if (secret_number != 0) {
35         printf("Huh, how did you do that? My secret number is now %ld (0x%lx)\n",
36     }
37
38     return 0;
39 }
40
```

Challenge 1: Basic Buffer Overflow

Exploiting the Bug:

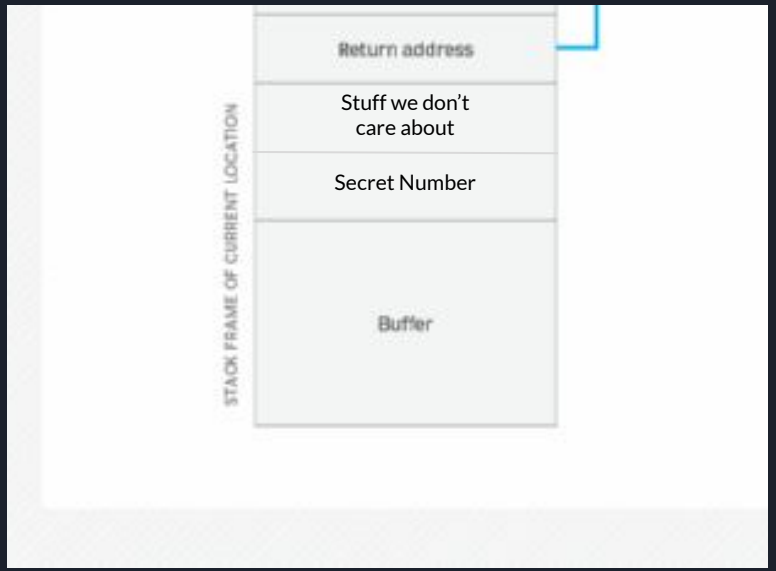
- Buffer and int are placed on stack
- gets() does not limit input
- ?



Challenge 1: Basic Buffer Overflow

Exploiting the Bug:

- Buffer and int are placed on stack
- gets() does not limit input
- Inputting long string + new number will overwrite secret number



Exploit Dev Time

Useful Commands:

- Info <>: print information on parts of the program
  - Functions: gives functions and their offsets
  - Registers: gives all registers and their values
- Disassemble <>: show the assembly code for a given function
- Break <>: set a breakpoint at a given location in the code
- x/<#>gx <>: print out ‘#’ amount of 8 byte hexadecimal values at given address

What’s the Big Deal?

- Nobody Uses gets() Function, right?
  - Poorly written code can elicit these attacks. gets() is just the simplest example that allows it
  - Many other vulnerabilities allow stack manipulation
- Manipulating values in a stack frame is rarely useful, right?
  - The fun is just beginning

Challenge 2: Ret2Win

What chal2 Does:

- Receives Input
- Has win function that prints flag
- NO WAY TO CALL WIN

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

void win() {
    //Following code prints out file flag.txt
    FILE *file = fopen("flag.txt", "r"); // Open the file in read mode
    if (file == NULL) {
        printf("Error: Could not open flag.txt\n");
        return;
    }
    char ch;
    while ((ch = fgetc(file)) != EOF) { // Read characters one by one
            putchar(ch); // Print the character
    }
    fclose(file); // Close the file
}

int main() {

    char buf[20];


    printf("What is your name?\n");
    gets(buf);

    printf("Hi %s, can you figure out how to win?\n", buf);

    return 0;
}
```

C Calling Conventions Note 1

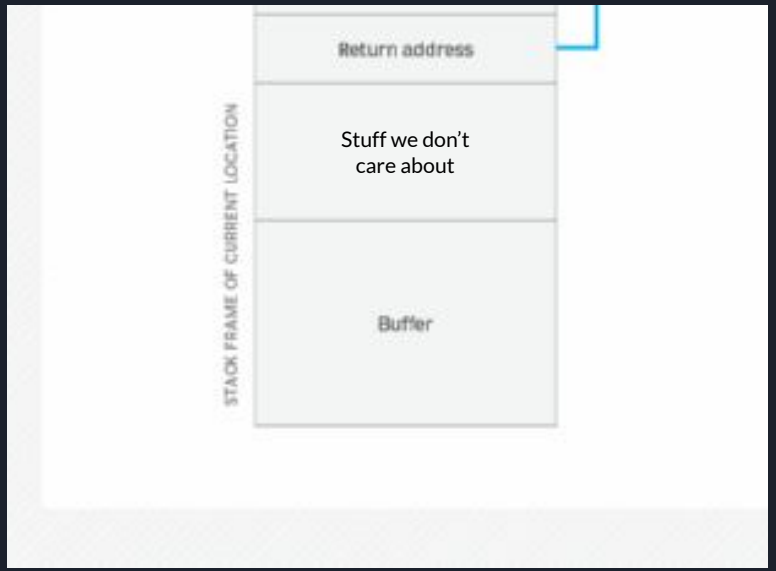
C Calling Conventions: Set of rules programmers and compilers abide by to ensure functions interact seamlessly

- Notable Rule: return pointer
  - Place the address you wish to return to directly after the stack frame of your function
  - When calling ‘ret’, move the instruction pointer to the value on the top of the stack

Challenge 2: Ret2Win

Exploiting the Bug:

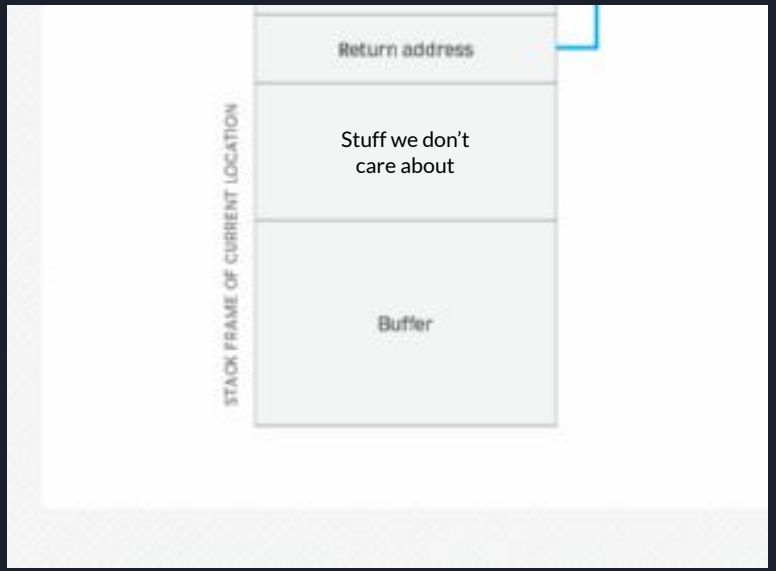
- Buffer and int are placed on stack
- gets() does not limit input
- ?



Challenge 2: Ret2Win

Exploiting the Bug:

- Buffer and int are placed on stack
- gets() does not limit input
- Write address of win() after long string of characters



Exploit Dev Time

Useful Commands:

- Info <>: print information on parts of the program
  - Functions: gives functions and their offsets
  - Registers: gives all registers and their values
- Disassemble <>: show the assembly code for a given function
- Break <>: set a breakpoint at a given location in the code
- x/<#>gx <>: print out ‘#’ amount of 8 byte hexadecimal values at given address

Challenge 2a: Intro to ROP

What Changed:

- Win has parameter
- Parameter must equal 42
- NO WAY TO EDIT INPUT PARAMETER

```c
void win(long int secret_number) {
    if (secret_number != 42) {
        printf("Close, but my secret number is %ld (0x%lx)\n", secret_
        return;
    }
    //Following code prints out file flag.txt
    FILE *file = fopen("flag.txt", "r"); // Open the file in read mode
    if (file == NULL) {
        printf("Error: Could not open flag.txt\n");
        return;
    }
    char ch;
    while ((ch = fgetc(file)) != EOF) { // Read characters one by one
        if (secret_number == 42) // Extra check to avoid shenanigans
            putchar(ch); // Print the character
    }
    fclose(file); // Close the file
}

int main() {

    char buf[20];


    printf("What is your name?\n");
    gets(buf);

    printf("Hi %s, can you figure out how to win?\n", buf);

    return 0;
}
```

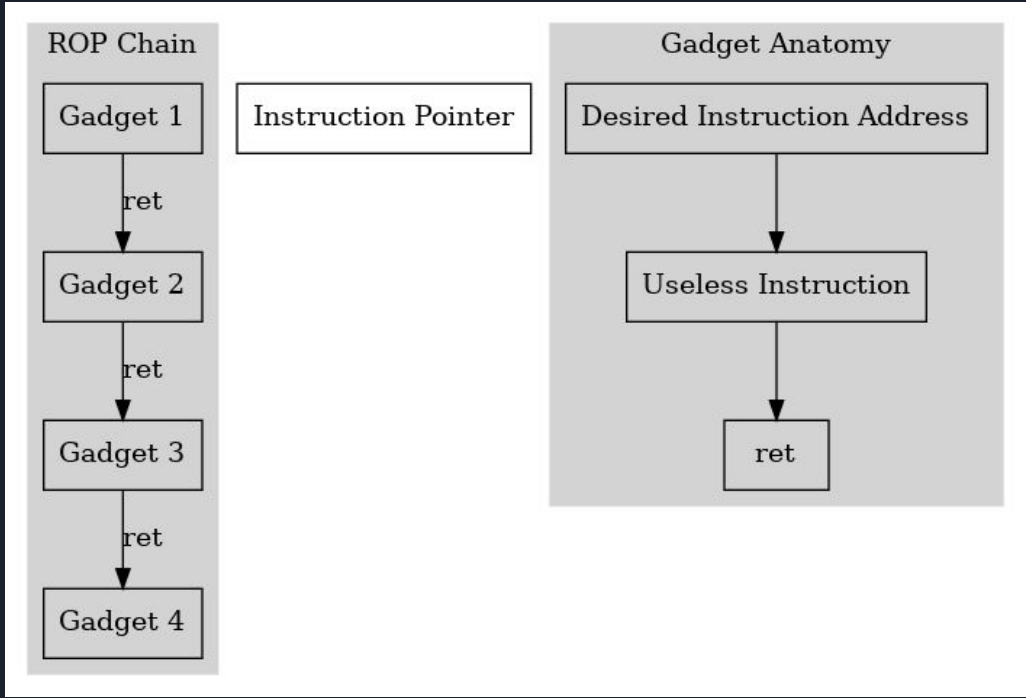
C Calling Conventions Note 2

C Calling Conventions: Set of rules programmers and compilers abide by to ensure functions interact seamlessly

- Notable Rule: Passing Parameters
  - When calling a function with parameters, place the parameters in `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` respectively
  - When writing a function with parameters, pull the values from `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` respectively

ROP: Return Oriented Programming

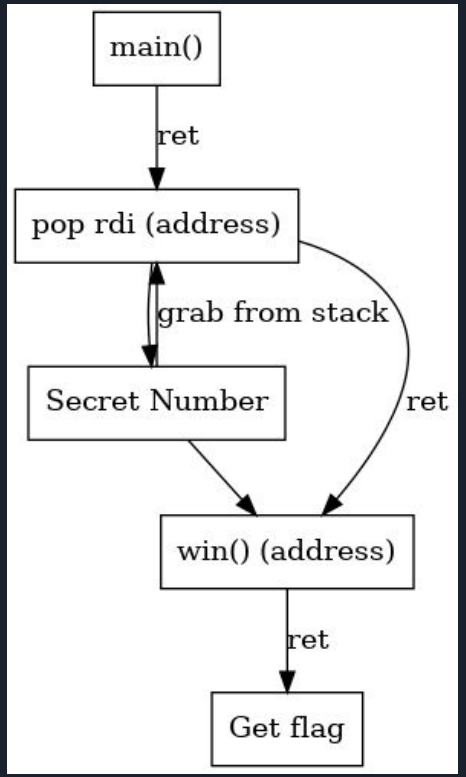
- Method of using instructions within binary to execute malicious task
- ‘Chain’ together desired instructions via ‘ret’ instruction



ROP: Return Oriented Programming

For chal2a:

- Must create a ‘chain’ of instructions and variables
- Instructions must be followed by ‘ret’ instruction
- After ‘ret’ is called, rip loads from top value of the stack
  - Next link of rop chain is executed



Exploit Dev Time

Useful Commands:

- Info <>: print information on parts of the program
  - Functions: gives functions and their offsets
  - Registers: gives all registers and their values
- Disassemble <>: show the assembly code for a given function
- Break <>: set a breakpoint at a given location in the code
- x/<#>gx <>: print out ‘#’ amount of 8 byte hexadecimal values at given address

What’s the Big Deal?

- You’re still limited to instructions and functions within the binary, right?
  - Not Exactly

Challenge 3: Ret2Libc

What Changed:

- No more win function
- Program has no way to print flag.txt

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

int main() {

    char buf[20];


    printf("What is your name?\n");
    gets(buf);

    printf("Hi %s, can you figure out how to win?\n", buf);

    return 0;
}

```

C Conventions Note 3:

- C functions are (most often) dynamically linked to binaries
  - The source is stored on the system, not in the program
- It is possible to ‘leak’ the addresses within the libc binary
  - All function offsets from the base address are equal for equivalent libc versions
- Reminder: all functions end with ret instruction…
  - Any idea on how to write remotely executed code?

Quick Libc Note:

- Noteable Lib_C values:
  - system(char *param): execute a system command
  - “/bin/sh”: string that is located in lib_c

Exploit Dev Time

Useful Commands:

- Info <>: print information on parts of the program
  - Functions: gives functions and their offsets
  - Registers: gives all registers and their values
- Disassemble <>: show the assembly code for a given function
- Break <>: set a breakpoint at a given location in the code
- x/<#>gx <>: print out ‘#’ amount of 8 byte hexadecimal values at given address

Quick Note:

When doing this not on local machine, there are a few more steps:

- Finding base address of lib_c must be done differently
- ASLR (mitigation) makes this much more difficult (but often not impossible)
    - Address Space Layout Randomization: Randomizes addresses on the kernel level (ie. Libc)

Notes:

- Limitations:
  - Permissions of Binary
  - Available ROP gadgets/memory space
  - Mitigations: Canary, ALSR, PIE, RELRO
- Large amount of unmentioned exploits and methods exist
- Vulnerability often found in backend code of system or library

Examples:

- CVE 2023-5474: Heap-Based Buffer Overflow (not covered) in Chrome PDF reader: 10/11/23
  - 2 other Heap Exploits found in same week
- CVE 2023-35662: Buffer Overflow in Android Source Code: 10/11/23
  - Potentially leads to remote code execution without user input
- CVE 2023-45199: Buffer Overflow in ARM mbed_tls (C library): 10/7/23
  - Leads to remote code execution
- CVE 2023-26370: Uninitialized Pointer (Not Covered) in Photoshop: 10/11/23
  - Can lead to remote code execution if user opens malicious file

Source: https://www.cisa.gov/news-events/bulletins/sb23-289

How to Prevent These Attacks

- Ensure GLIBC and packages and systems are up to date
- Large amount of unmentioned exploits and methods exist
  - For stack exploits: Avoid practices that let variable-sized values copy to a static allocation
  - Sanity check inputs:
    - strcpy, strcat, sprintf, gets -> strncpy, strncat, snprintf, fgets
  - For heap exploits:
    - Limit user's ability to malloc, free, and write to chunks.
    - Practice good memory management

Resources to Learn More

- **Learn:**
  - **CryptoCat (Youtube):** simple challenge walkthroughs focussing on exploiting Stack Smashing Mitigations
  - **Ir0nStone Git Blog:** Great blog overviewing many ROP tactics and mitigations
- **Practice:**
  - **PicoCTF:** Large amount of CTF Binex (and other) Challenges
  - **ROPEmporium**: 8 Excellent challenges teaching ROP

Questions?