

# Exploitation and Reverse Engineering

Lecture 6 - ECS198F SQ26



# Plan of the Day



## Exploring Binaries

- How do programs interact with memory when executed?
- What does it mean to “exploit” a binary?

## Program Exploitation

- What are various techniques of exploiting a program, and ways to protect a binary?
- What tools can we use to assist in the process?

## Reverse Engineering

- How do you exploit a binary if you don’t know the source code?
- What reverse engineering tools exist, and how can we use them?



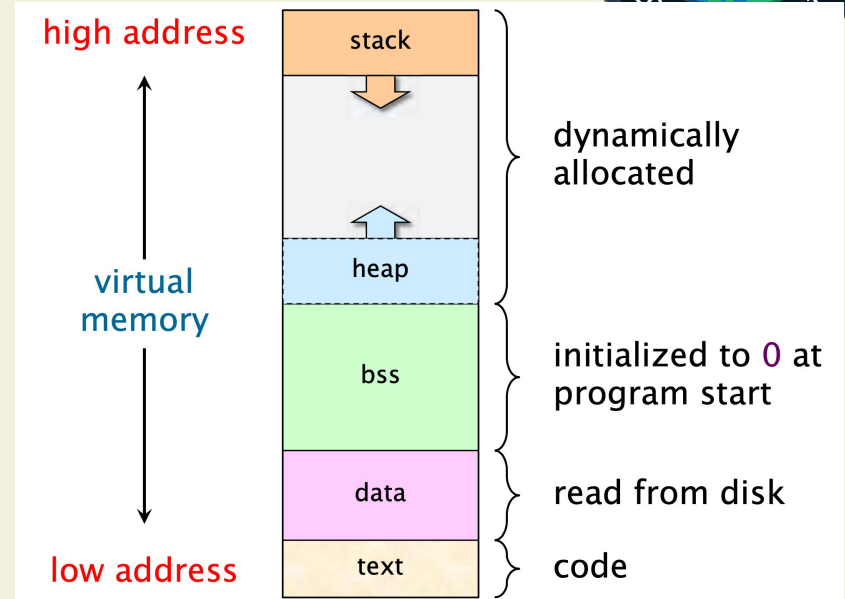
# **1. Program Exploitation Concepts**

# Quick Review of ECS 50



*What happens when a program is run?*

- Computer loads code and variables into memory, initializes some variables
- Allocates the stack and the heap
  - **Stack** - Stores information about the currently running function (local vars, args, etc.)
  - **Heap** - Dynamically allocated memory



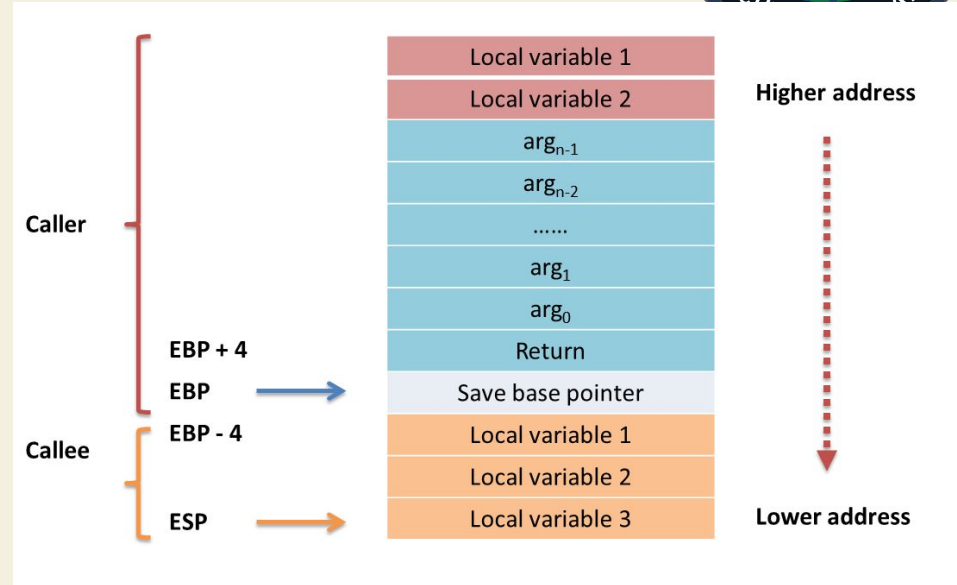
<https://martinlwx.github.io/en/wh-at-is-the-heap-and-stack/>

# Quick Review of ECS 50



What does the stack do and look like?

- Stores your local variables and arguments
- Stores information about the function that called the current one
  - Current function is everything between *ebp* and *esp*
  - Store location to jump to after execution is done!



[https://nobelsharanyan.wordpress.com/wp-content/uploads/2015/05/function\\_stack1.png](https://nobelsharanyan.wordpress.com/wp-content/uploads/2015/05/function_stack1.png)

# So What Is Program Exploitation?



## Basic Functions of a Computer

- Take Inputs
- Read/write from memory
- Perform operations
- Produce outputs

## What can we do?

- Give a program funny inputs
- **Read/Alter a program's memory**
- **Change the control flow or run custom code!**
- Achieve some desired output!

# relevant tiktok strikes again



<https://www.youtube.com/watch?v=Nz8ssH7LiB0>



## **2. The Process of pwning**

# Tooling



## Useful Tools in your Toolkit

- *gdb* - Yes, you can use a plain old debugger
  - [pwndbg](#) - Plugin for *gdb* that includes QoL features for exploit development
- *file*, *checksec* - Linux utilities that provide info about binaries
- [pwntools](#) - Python library that assists in exploit development

```
Breakpoint 2, main () at vuln.c:30
30  printf("Hello, %s\n", buf);
(gdb) x/80x $sp
0x7fffffffda0: 0x41414141      0x41414141      0x00000000      0x00000000
0x7fffffffdb0: 0x00000000      0x00000000      0x77c2a575      0x00000000
0x7fffffffdb8: 0xffffd1e8      0x00000000      0xffffd1e8      0x00000000
0x7fffffffdbf: 0xffffd120      0x00000001      0x00000000      0x00000000
0x7fffffffdf0: 0x00000000      0x00000000      0xcd4a82c2      0x87de2788
0x7fffffffdf8: 0xffffd1e8      0x00000000      0x00000000      0x00000000
0x7fffffffdf9: 0x7ffdf900      0x00000000      0x00000000      0x00000000
0x7fffffffdfb: 0x7ffdf900      0x00000000      0x00000000      0x00000000
0x7fffffffdfc: 0x7ffdf900      0x00000000      0x25d482c2      0x87de37f2
0x7fffffffdfd: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdfe: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdf7: 0xffffd1e8      0x00000000      0x25aee000      0x3b50486b
0x7fffffffdf8: 0xffffd1c0      0x00000000      0xf7c2a628      0x00000000
0x7fffffffdf9: 0xffffd1f8      0x00000000      0x00000000      0x00000000
0x7fffffffdfa: 0x00401304      0x00000000      0xffffd1f8      0x00000000
0x7fffffffdfb: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdfc: 0x00401170      0x00000000      0xffffd1e0      0x00000000
0x7fffffffdfd: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdf0: 0x00000000      0x00000000      0x00401195      0x00000000
0x7fffffffdf1: 0xffffd1d8      0x00000000      0x00000038      0x00000000
[ SOURCE CODE ]
In file: /mnt/windows/Users/nuclе/CTF/CSCDevelopment/picocf/binexp/echo_escape/1/vuln.c:30
21 int main() {
22   char buf[32];
23
24   printf("Welcome to the secure echo service!\n");
25   printf("Please enter your name: ");
26   fflush(stdout);
27
28   read(0, buf, 128);
29
30   printf("Hello, %s\n", buf);
31   printf("Thank you for using our service.\n");
32
33   return 0;
34 }
[ STACK ]
00:0000 rsi rsp 0x7fffffffce70 {buf} ← 'AAAAAAAAAAAAA\n'
01:0008 -018 0x7fffffffce78 {buf+0x8} ← 0xa4141414141 /* 'AAAAA\n' */
02:0010 -010 0x7fffffffce80 {buf+0x10} ← 0
03:0018 -008 0x7fffffffce88 {buf+0x18} → 0x7ffff7fe2560 (dl_main) ← endbr64
04:0020 rbp 0x7fffffffce90 → 0x7ffffffcf30 → 0x7ffffffcf90 ← 0
05:0028 +008 0x7fffffffce98 → 0x7ffff7c2a575 (__libc_start_call_main+117) ← mov edi, eax
06:0030 +010 0x7fffffffcea0 → 0x7ffff7f0000 ← 0x30102464457f
07:0038 +018 0x7fffffffcea8 → 0x7ffffffcfb8 → 0x7ffffffd49f ← '/mnt/windows/Users/nuclе/CTF/C
[ BACKTRACE ]
0 0x401358 main+84
1 0x7ffff7c2a575 __libc_start_call_main+117
2 0x7ffff7c2a628 __libc_start_main+136
3 0x401195 _start+37
pwndbg>
```

# Demo #1: Buffer Overflows



<https://play.picoctf.org/practice/challenge/438>

<https://play.picoctf.org/practice/challenge/439>

# Demo #2: Overflow the Stack



If stack variables are vulnerable to buffer overflow, what can we do with that control?

<https://play.picoctf.org/practice/challenge/75>

5

# Protection Against Overflows?



## Possible Solution:

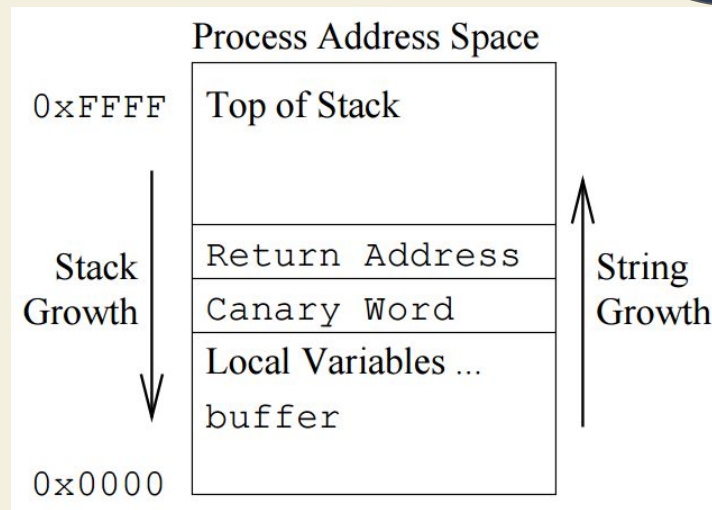
- Don't use "unsafe functions..."
  - Could work!
- "Just don't use C/C++"
  - I mean a lot of languages DO protect you (Python, Rust, etc.)
  - Not always about "exploitability", the ability to make a program crash is just as important!
- Make the operating system protect you against some attacks!

# Operating System Protections



## Enter stack canary!

- Some random piece of data inserted at runtime to check if “stack smashing” is occurring
  - If it is, kill the program
- Can be turned off; or, if you have the right exploit, leaked
  - Use *checksec* to see if it’s enabled!



[https://tianocore-docs.github.io/ATB/B-Mitigate\\_Buffer\\_Overflow\\_in\\_UEFI/draft/media/image2.png](https://tianocore-docs.github.io/ATB/B-Mitigate_Buffer_Overflow_in_UEFI/draft/media/image2.png)

# Format String Vulnerabilities



```
// Assume some user input in a variable user_input

int main() {
    char user_input[50];

    printf(user_input);

    printf("%s", user_input);

    return 0;
}
```

Which one of these printf's is vulnerable, and what is not?

# Format String Vulnerabilities



**printf(variable)** allows users to pass in arbitrary format strings to read!

- Enables “stack walking” by sending a format string saying you have a lot more variables to print! (%x, %p)
- Arbitrary writes with space padding chained with %n!

```
// Assume some user input in a variable user_input  
  
int main() {  
    char user_input[50];  
    printf(user_input);  
    printf("%s", user_input);  
    return 0;  
}
```

# Demo #3: Format String Vulns



<https://play.picoctf.org/practice/challenge/434>

# Arbitrary Code Execution Techniques



- **Shellcode** - Inject assembly instructions into memory, and tell computer to run it
- **Return-Oriented Programming (ROP)** - Jump execution around existing code to chain together a program (avoids protections)

## Protections Against ACE

- **PIE / ASLR** - Randomize memory addresses of programs AND segments to make execution less predictable
- **DEP** - Don't allow execution from certain memory segments (stack, etc.)

<https://ctf101.org/binary-exploitation/return-oriented-programming/>



## **3. Reverse Engineering**

# Ideas of Reverse Engineering



## Objectives

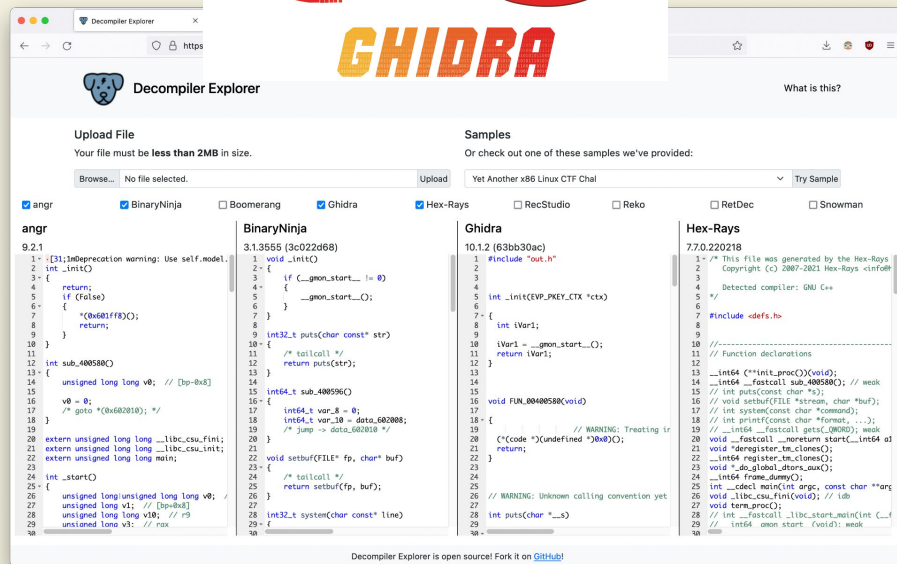
- 1) Understand how a program works (source code usually not provided)
- 2) Check it for anything of interest (vulnerabilities, functionality, etc.)

## Challenges

- **You gotta figure out what all the variables and functions do**
  - If debug symbols / mapping not provided, you have to make sense of the noise
- Interpretation may vary! Some tools represent the same thing different ways!

# Tooling

- **Decompiler** - Program that attempts to map assembly instructions back into code
  - Dogbolt (Decompiler Explorer), Ghidra, IDA/Binary Ninja
- **Disassembler** - Get the assembly instructions from the machine code



# Last Demo



<https://github.com/SunshineCTF/SunshineCTF-2025-Public/tree/main/i95/Jupiter>