Introduction to Binary Exploitation

Andrew Haberlandt

Feburary 23, 2021

- **Don't forget to start recording**
- Slides are on https://wiki.osucyber.club
- Some content adapted from: Nathan Peercy (Purdue)

Announcements

- **Next Week:** Group solve time focusing on pwn

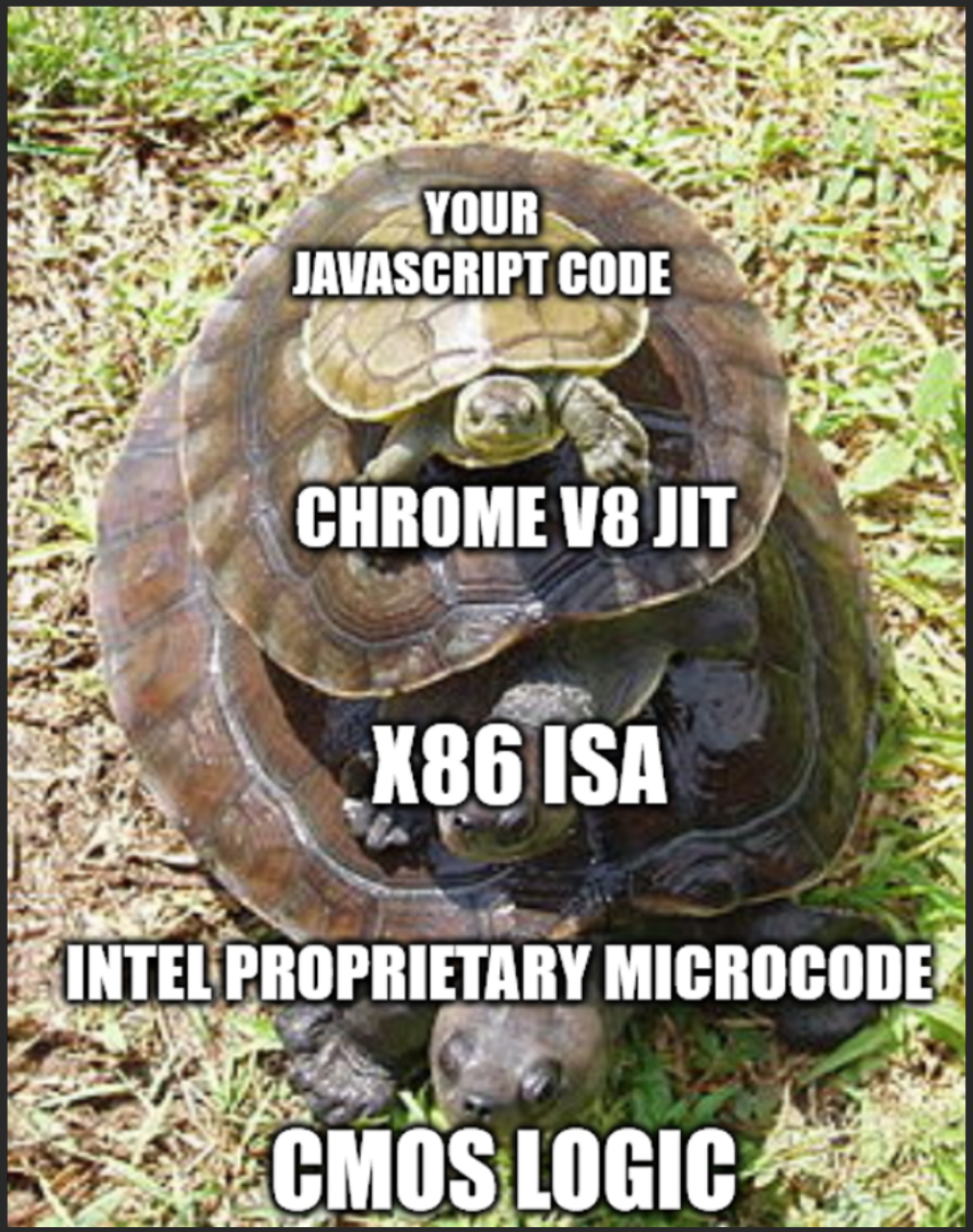
Overview

What is binary exploitation?

- Make a [compiled/binary] program do something it wasn't intended to do
- Usually this means the goal is arbitrary code execution
- pwn is awesome but pwn is hard. You'll eventually need a good understanding of:
  - Assembly (we'll use x86)
  - C
  - Operating systems

Why

- Native code is everywhere
- No really, EVERY APPLICATION EVER ULTIMATELY ENDS UP RUNNING MACHINE CODE


YOUR
JAVASCRIPT CODE
CHROME V8 JIT
X86 ISA
INTEL PROPRIETARY MICROCODE
CMOS LOGIC

Why

- GOAL: Take over ('pwn') a machine
  - We want **Arbitrary Code Execution**. If you can run whatever code you want, you can do anything the OS allows that process to do (usually anything the user can do)
  - After you have code execution, **privilege escalation** concerns exploiting (usually) the operating system to gain additional privileges (i.e. normal user root access, or sometimes directly to kernel code execution)
  - Some special programs (ie. sudo) are usually given extra privileges to run as root ("suid programs"); exploit one of these and your code runs as root
  - Examples: bug bounties like Valve, Tesla, many others

What does it look like to 'pwn' a machine?

- **Real-life examples**
  - Buffer overflow in CSGO/Half-Life/TF2 client, allowing a malicious server to run code just by viewing the server's info https://hackerone.com/reports/470520
  - Read an arbitrary file on a Half-Life server: https://hackerone.com/reports/590279
- **Arbitrary code execution**
  - Often demonstrated by: getting a shell
  - "popping calc"
  - Sometimes in CTF problems: opening and reading a special 'flag' file is sufficient (can do with shell, or not)
  - After you have code execution, you're often done unless you're the NSA or CIA

Demo 1: Reverse shell in Bash

Common vulnerabilities

- Command injection
- Memory corruption: a broad category
- Off-by-one errors
- Race conditions
- (lack of) Input validation
- Miscellaneous logic bugs

Command Injection

- Programmers are lazy and often 'shell out' to run another program
- If an attacker controls part of the command being run, they can use features of the shell (usually bash) to end the previous command and inject their own

```
#include <stdio.h>
void get_log_file(char *name) {
  char cmd[50];
  sprintf(cmd, "cat log_file_%s", name);
  system(cmd);
}
```

- ie. if you can cause the program to call get_log_file with something like
  `; cat /etc/passwd`
  then you can run other commands

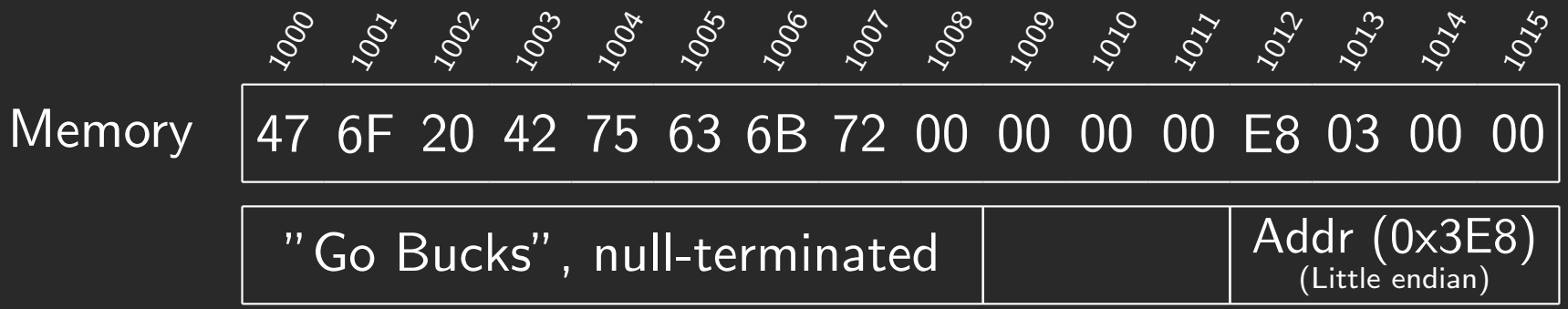
Demo 2: Command injection

Logic Bugs

- Most are unintentional, Intentional logic bugs are essentially "backdoors".
- Leaving in debug options: ex. bypass access control by adding debug=1
- Program gets confused about state when receiving packets/commands out-of-order
- An example: "[steam client] Opening a specific steam:// url overwrites files at an arbitrary location" https://hackerone.com/reports/667242

Memory Corruption Bugs

- Memory is a byte-addressable array of bytes
- If a user can write somewhere they are not intended to, it can be the cause of a memory corruption bug
- Everything has a finite size

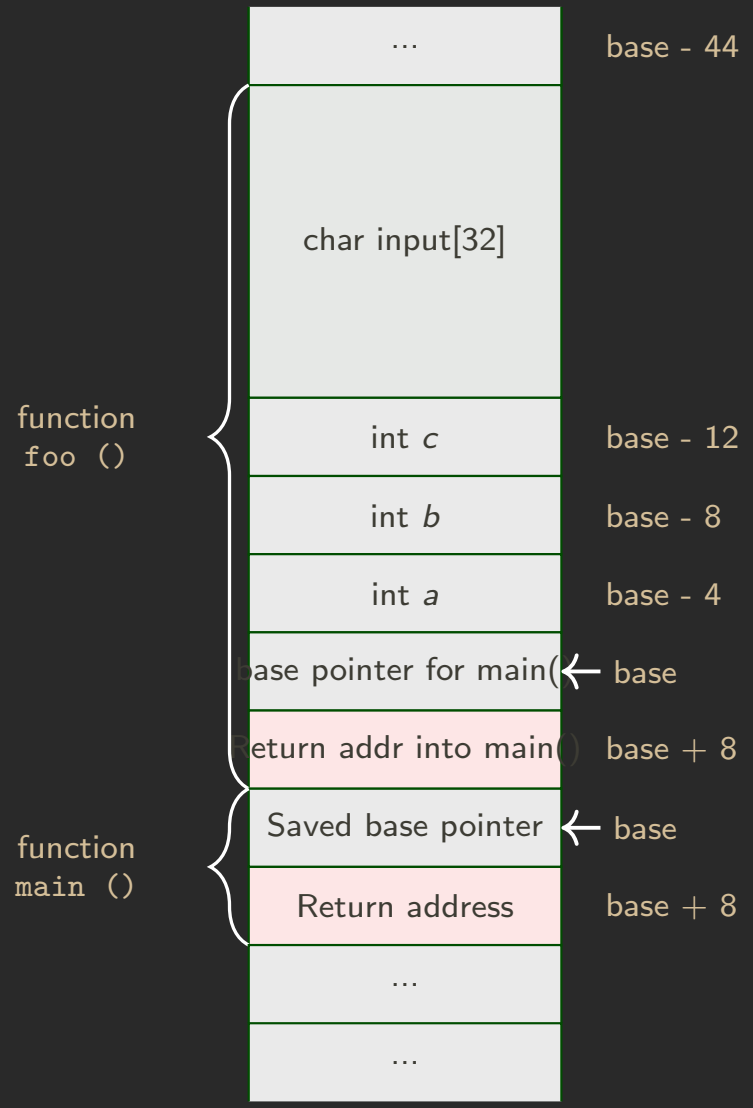


Memory Corruption Bugs

- String Functions and length-checks
  - **strcpy / strncpy:** Arguments are two pointers, copies all characters from one location to the other until it reaches a null byte
  - **gets / fgets:** Reads a string from the user (gets uses stdin, fgets can use any file)
  - **others:** Most other string functions rely on the strings being terminated by a null byte and many don't check length

Memory Corruption Bugs

- The important question is "can we overwrite anything important?"
  - To answer, need to know some things about where stuff is in memory
- Every function starts by making room for it's own local variables on the stack

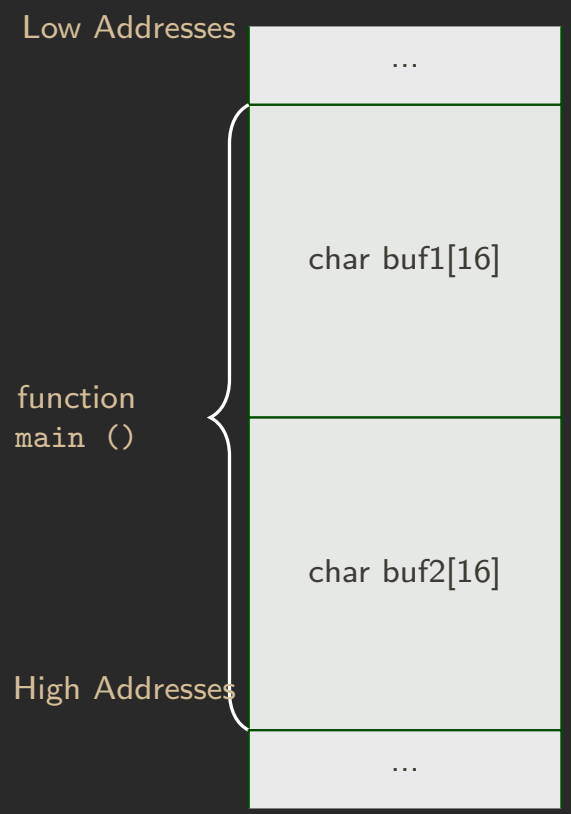


Simplest example: Stack buffer overflow into another variable

```c
int main() {
  char buf1[16];
  char buf2[16];

  // get a (arbitrary-length) string from the user
  gets(buf1);

  // print the strings back to the user
  printf("buf1: %s\n", buf1);
  printf("buf2: %s\n", buf2);

  // check for win condition
  if (strcmp(buf2, "win") == 0) {
    system("/bin/sh");
  }
}
```

Simplest example: Stack buffer overflow into another variable

- We can change the value of other variables on the stack!

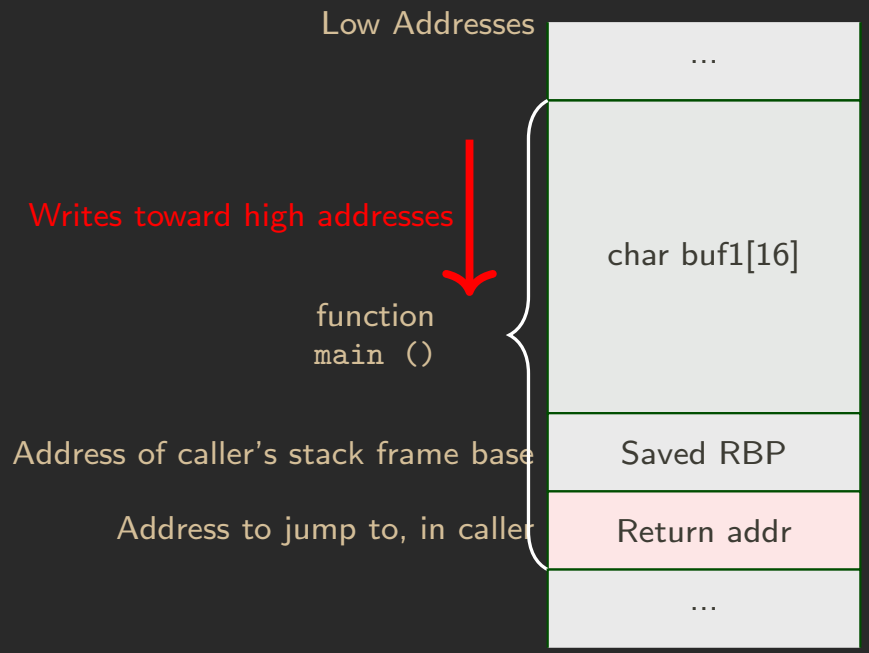


Demo: Stack buffer overflow and pwntools

OK, but we want code execution

Stack buffer overflow into return address

```
int main() {
    char buf1[16];

    // get a (arbitrary-length)
    // string from user
    gets(buf1);
}
```



1990s exploitation

- You control where the program goes when the current function returns
- Just put your code somewhere you know the address of, and then overwrite the return address so that it holds the address of your code
- When the function returns, your code runs

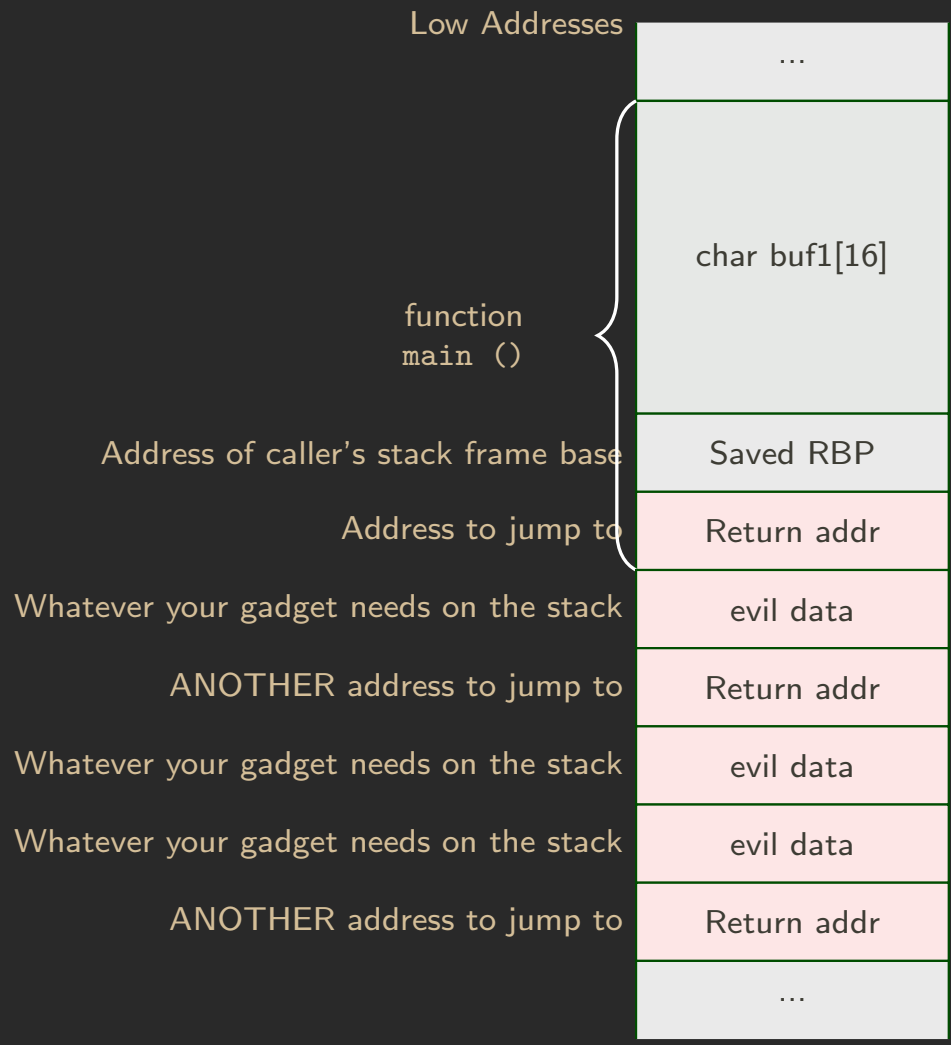
Demo 3: Return to the 1990s + pwntools

Modern memory protections have entered the chat

- You can't just write your code onto the stack and jump to it. The stack is not normally executable.
- We need to be more creative

ROP

- We don't just control the return address, we can keep writing and we control the whole stack
- We can return to specific locations that do useful things and then return again ('gadgets')
  - pop rdi; ret;
  - pop rdx; pop rax; ret;
- Chain these together to control all registers and you can eventually make syscalls, etc.



More on ROP

- https://docs.pwntools.com/en/dev/rop/rop.html
- If there is interest we may do another ROP talk

More examples of memory corruption

- Some memory bugs can lead to arbitrary read/write
  - Can you overwrite an entry in the Global Offset Table (GOT) - addresses of code in dynamically-loaded libraries
  - Can you leak the address of something else important?
- Controlled array index
  - You can read/write beyond the bounds of an array, without corrupting everything in between.

Common Defenses

- ASLR - Address Space Layout Randomization
- PIE - Position Independent Executable (DEMO?)
- NX - Non-eXecutable memory
- Stack Canaries
- Many more...

Useful Tools

- GDB with GEF, pwndbg
- Ghidra and RE tools
- python3 with pwntools
- checksec, ROPGadget, Ropper

Recommended Challenges

- **speedrun series**: 8-100 points, starts from basics and gradually becomes more difficult/more modern
- **speedrun0, 0.25, 0.5, and 1** are *very* similar to what was demoed today, it progresses from there in small steps

Next Week

- **Highly Recommended:** Attempt some pwn challenges and have questions ready to go
- https://bootcamp.osucyber.club/