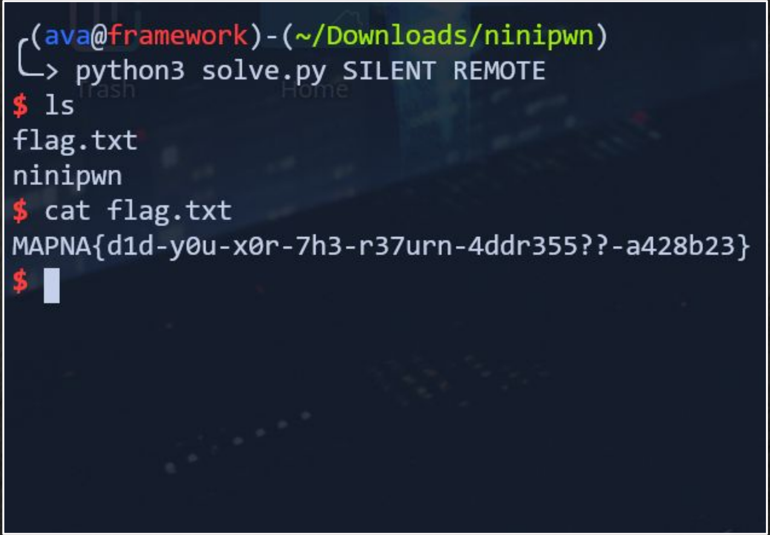
INTRODUCTION TO PWN

Cyberkickstart

WHAT IS PWN?

- pwn a.k.a. **binary exploitation**
- takes advantage of vulnerabilities in executables
    - think compiled binaries
- e.g. buffer overflows
- probably the hardest CTF category to get into
    - requires an understanding of the underlying assembly code
    - gets complicated due to protections and mitigations



SCENARIO

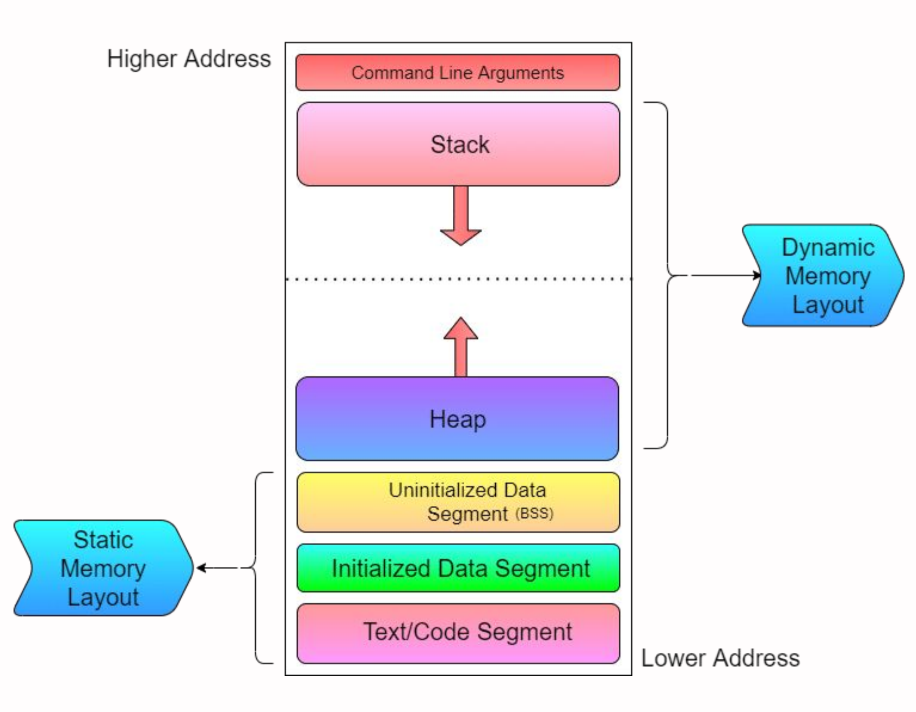
- Pwn CTF problems will typically give you two things:
  1. Compiled binary (typically C or C++)
  2. Remote netcat connection (IP address and port)
- You can run the binary locally, but in order to get the flag, you must be able to exploit the binary running on the **remote** connection
- Goal: send a very specific input to the remote binary which will exploit it to either **print the flag** or give you a **shell**
- There are multiple ways to do this (depends on the challenge), but today I'll cover simple **buffer overflows** to achieve <u>variable overwrite</u> and <u>ret2win</u>

C PROGRAMS

- C is a **compiled** language
  - in order to run it, you have to compile it down to assembly language first
  - compiled program a.k.a. binary, executable, **ELF**
- ELF = Executable and Linkable Format
- Challenges will give you a binary, but not usually the source code for it
- Unless you want to read the raw assembly, you'll want to decompile it (like we did for rev last week)

MEMORY LAYOUT

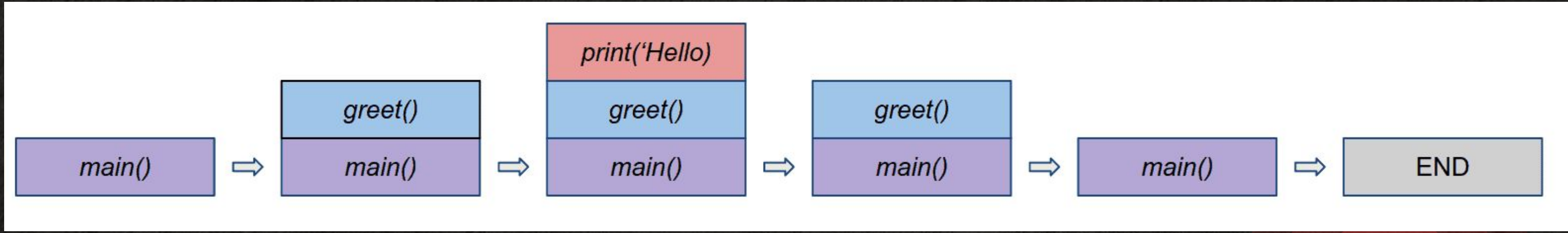
- ELF files define a program's memory layout
- There are different sections that correspond to different things
  - text = actual code
  - heap = dynamically allocated memory (malloc)
  - stack = local variables, function parameters, return addresses
- The stack grows from higher → **lower** addresses
  - top of stack = lowest address



CALLING CONVENTION

- The **stack** stores function information
  - variables allocated inside the function
  - return address of the function
- Every time you call a function, it adds a new stack frame
- Example:

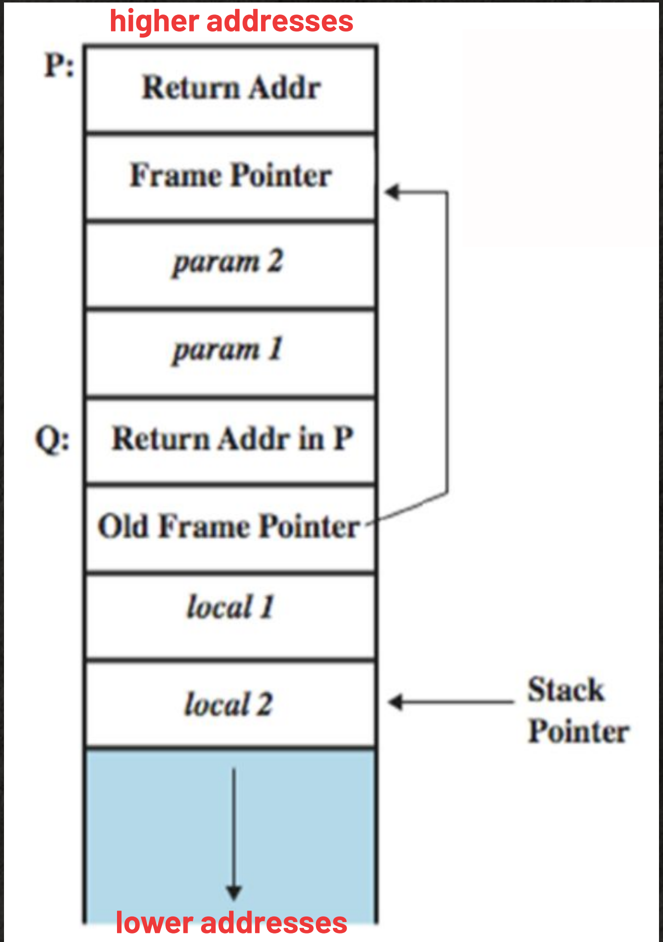
```
function Greet():
    print 'Hello'

function Main():
    Greet()
```



STACK FRAMES

Example:

If function P( ) calls function Q( )



EXAMPLE

```c
#include <stdio.h>

int numbers() {
    int a = 1;
    int b = 2;
    printf("a + b = %d\n", a + b);
}

int main() {
    numbers();
    return 0;
}
```

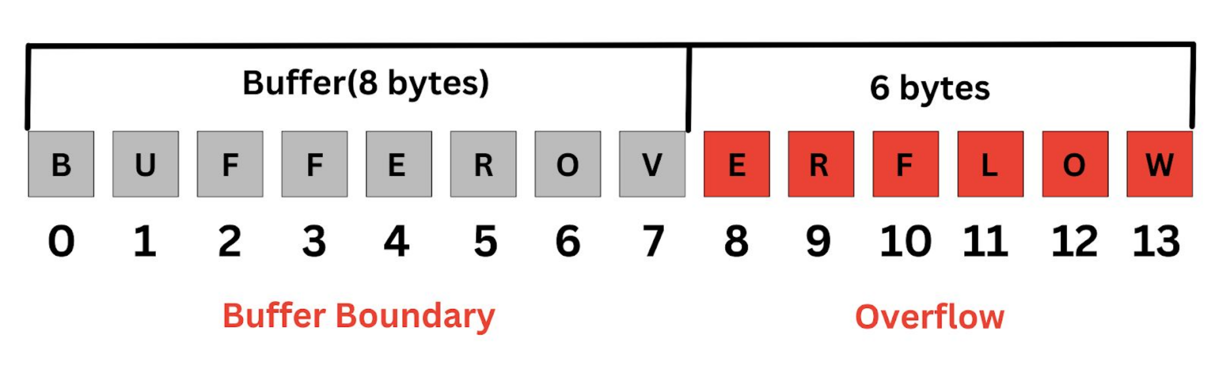
HIGHER ADDRESSES

Stack frame for *main*
Return address to inside of *main*
Saved frame pointer (rbp)
Local variable *a* (4 bytes)
Local variable *b* (4 bytes)

LOWER ADDRESSES

MEMORY MANAGEMENT

- Who has heard of a buffer overflow attack before?
- C requires **manual** memory management
- If you don't allocate your buffer size and handle input correctly, you can **overflow** it



EXAMPLE

```c
void vuln() {
    char name[32];
    printf("What's your name? ");
    gets(name);
    printf("Hi!\n");
}
```

What’s wrong with this code?

UNSAFE CODE

- Certain C functions are **unsafe** because they do not do bounds checking
    - e.g. gets
    - This means that you can write beyond the bounds of the buffer
- Other C functions are only safe if used correctly
    - e.g. a function might ask you to specify a maximum size, but you specify a size greater than the size of the buffer (fgets)
- Write more data than the buffer can hold = **buffer overflow**
- Buffer overflows allow you to modify the value of things contained **higher** up on the stack
    - anyone see where this is going?

```
❯ objdump -M intel --source win | grep -i win
win:     file format elf64-x86-64
000000000040123d <win>:
  401274:       75 0e                   jne    401284 <win+0x47>
```

```c
void win() {
  puts("Nice job!");
  char flag[256];

  FILE *flagfile = fopen("/ctf/flag.txt", "r");

  if (flagfile == NULL) {
    puts("Cannot read flag.txt.");
  } else {
    fgets(flag, 256, flagfile);
    flag[strcspn(flag, "\n")] = '\0';
    puts(flag);
  }
}

void vuln() {
    char name[32];
    printf("What's your name? ");
    gets(name);
    printf("Hi!\n");
}
```

HIGHER ADDRESSES

Stack frame for *main*
3d 12 40 00 00 00 00 00 (8 bytes)
41 41 41 41 41 41 41 41 (8 bytes)
41 41 41 41 41 41 … 41 (32 bytes)

LOWER ADDRESSES

Input: b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=\x12@\x00\x00\x00\x00\x00'

NOTES

- Overflows often require you to send unprintable characters, which isn't easy to send manually
  - pwntools Python library
- x86 uses something called **little-endian** format
  - endianness describes the order a computer stores/reads bytes in
  - little-endian means least-significant (lowest) byte is stored first
  - e.g. storing the hex number 0x12345678 looks like 78 56 34 12 in memory
  - this is important for overwriting values like variables or return addresses

DEMO

PRACTICE

"Overflow" and "Ret2Win" from BYU Fall End-of-Semester CTF 2022
https://github.com/BYU-CSA/old-ctf-challenges/tree/master/pwn/overflow
https://github.com/BYU-CSA/old-ctf-challenges/tree/master/pwn/ret2win

"ret2win" at ROP Emporium
https://ropemporium.com/challenge/ret2win.html

Take IT&C 515R- Vulnerabilities, Exploitation, and Reverse Engineering (VERE)