CS 5600
Computer Systems

Lecture 13: Exploits and Exploit Prevention

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP
- Kernel Exploits and Rootkits

Setting the Stage

Game executable is setuid

```
[cbw@ ifight game] ls -lh
-rw- ---- 1 amislove faculty  180 Jan 23 11:25 secrets.txt
-rwsr-sr-x 4 amislove faculty 8.5K Jan 23 11:25 guessinggame
```

- Suppose I really want to see the secret answers
  - But I’m not willing to play the game
- How can I run arbitrary code as amislove?
  - If I could run code as amislove, I could read secrets.txt
  - Example: execvp(“/bin/sh”, 0);

Looking for Vulnerabilities

- Code snippet for *guessinggame*

```
char buf[8];
for (int x = 1; x < argc; ++x) {
     strcpy(buf, argv[x]);
     num = atoi(buf);
     check_for_secret(num);
}
```

Stack buffer overflow

Confirmation

```
[cbw@finalfight game] ls -lh
-rw------- 1 amislove faculty  180 Jan 23 11:25 secrets.txt
-rwsr-sr-x 4 amislove faculty 8.5K Jan 23 11:25 guessinggame
[cbw@finalfight game] ./guessinggame 1 2 3
Sorry, none of those number are correct :(
[cbw@finalfight game] ./guessinggame AAAAAAAAAAAAAAAAAAAAAAA
Sorry, none of those number are correct :(
Segmentation fault (core dumped)
```

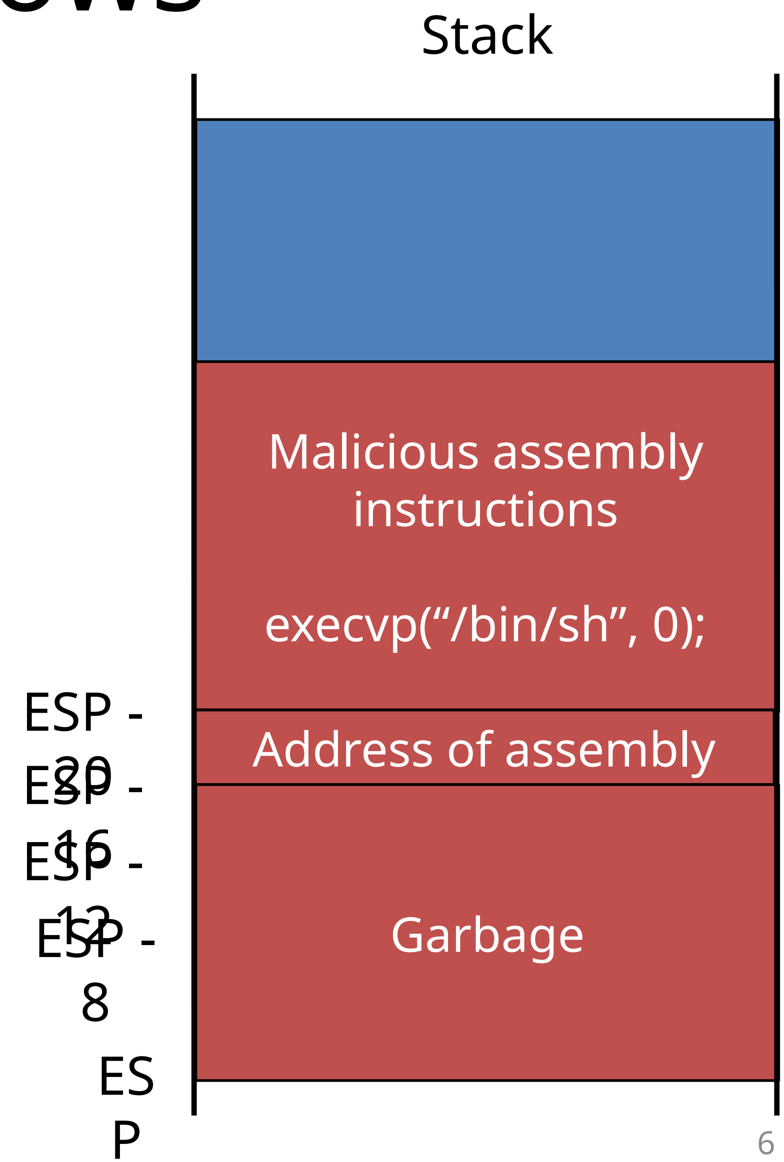
```
(gdb) bt
#0  0x0000000000400514 in myfunc ()
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x0000004141414141 in ?? ()
```

‘A’ = 0x41 in ASCII

Exploiting Stack Buffer Overflows

- Preconditions for a successful exploit

1. Overflow is able to overwrite the return address
2. Contents of the buffer are under the attackers control



Exploitation, Try #1

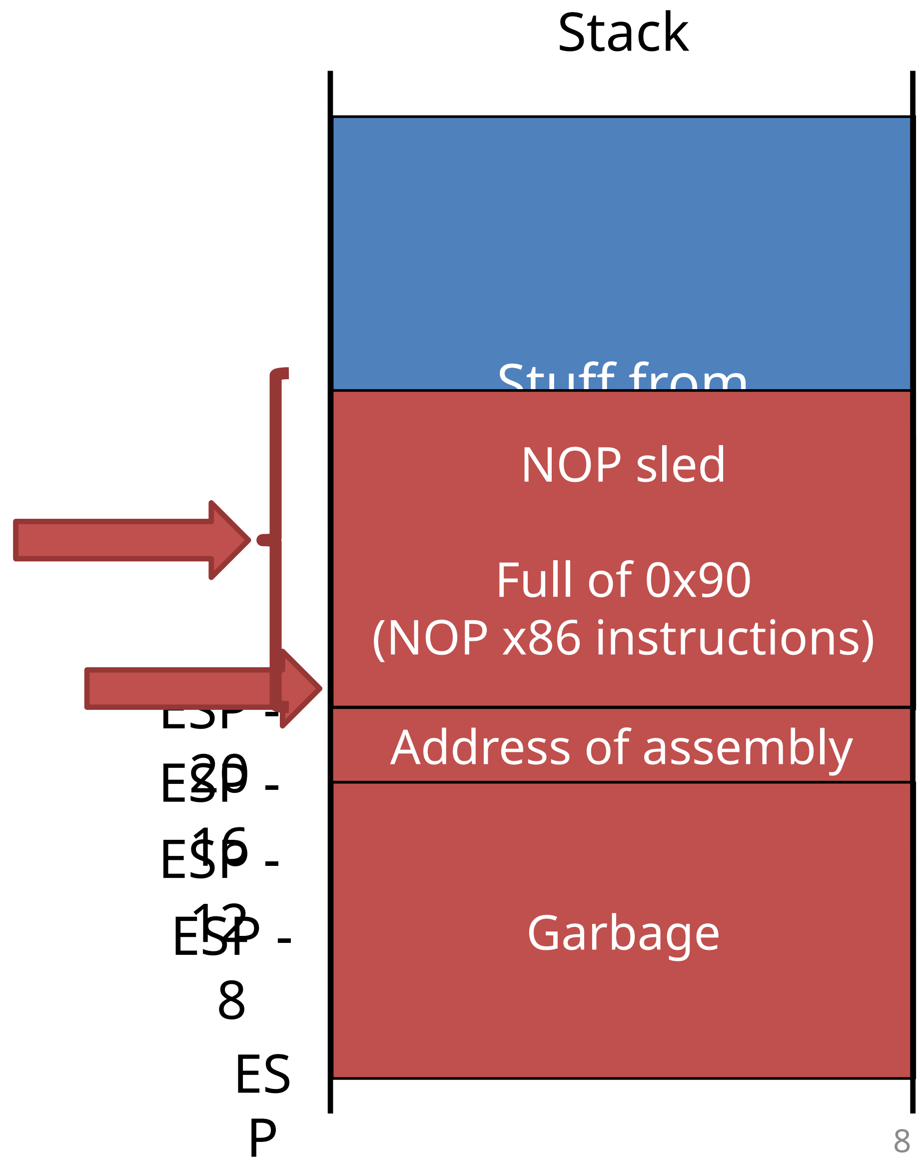
```
[cbw@finalfight game] ./guessinggame [16-bytes of garbage]
[4-byte stack pointer][evil shellcode assembly]
Segmentation fault (core dumped)
```

This is not what we want :(

- Problem: how do you know the address of the shellcode on the stack?
  - To execute the shellcode, you have to return to its exact start address
  - This is a small target

NOP Sled

- To execute the shellcode, you have to return to its exact start address
- You can increase the size of the target using a NOP sled (a.k.a. slide, ramp)



on, Try #2

./guessinggame ran the shellcode, turned into /bin/sh

```
[c    nalfight game] ./guessinggame [16 bytes of garbage]
[     yte stack pointer][2048 bytes of 0x90][evil shellcode
a  sembly]
$
```

- There is a lot more to writing a successful exploits
  - Depending on the type of flaw, compiler countermeasures, and OS countermeasures
  - If you like this stuff, take a security course

Types of Exploitable Flaws

- Stack overflow
- Heap overflow

```
char * buf = malloc(100);
strcpy(buf, argv[1]);
```

- Double free

```
free(buf);
free(buf);
```

- Format string

```
printf(argv[1]);
```

- Off-by-one

```
int vectors[100];
for (i = 0; i <= 100; i++)      vector[i] = x;
```

- … and many more

Triggering Exploitable Flaws

- Local vulnerabilities:
  - Command line argument
  - Environment variables
  - Data read from a file
  - Date from shared memory pipes
- Remote vulnerabilities
  - Data read from a socket
- Basically, any place where an attacker can give input to your process

Attacker can inject code into your machine via the Internet

Leveraging an Exploit

- After a successful exploit, what can the attacker do?
  - Anything the exploited process could do
  - The shellcode has full API access
- Typical shellcode payload is to open a shell
  - Remote exploit: open a shell and bind STDIN/STDOUT to a socket (remote shell)
- If process is uid=root or setuid=root, exploitation results in privilege escalation
- If the process is the kernel, the exploit also results in privilege escalation

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP
- Kernel Exploits and Rootkits

Defending Against Stack Exploits

- Exploits leverage programmer bugs
  - Programmers are never going to write code that is 100% bug-free
- What can the system do to help prevent processes from being exploited?
- Mechanisms that prevent stack-based exploits
  - Stack canaries
  - Non-executable stack pages (NX-bit)

The Canary in the Coal Mine

- Miners used to take canaries down into mines
- The birds are very sensitive to poisonous gases
- If the bird dies, it means something is very wrong!
- The bird is an early warning system

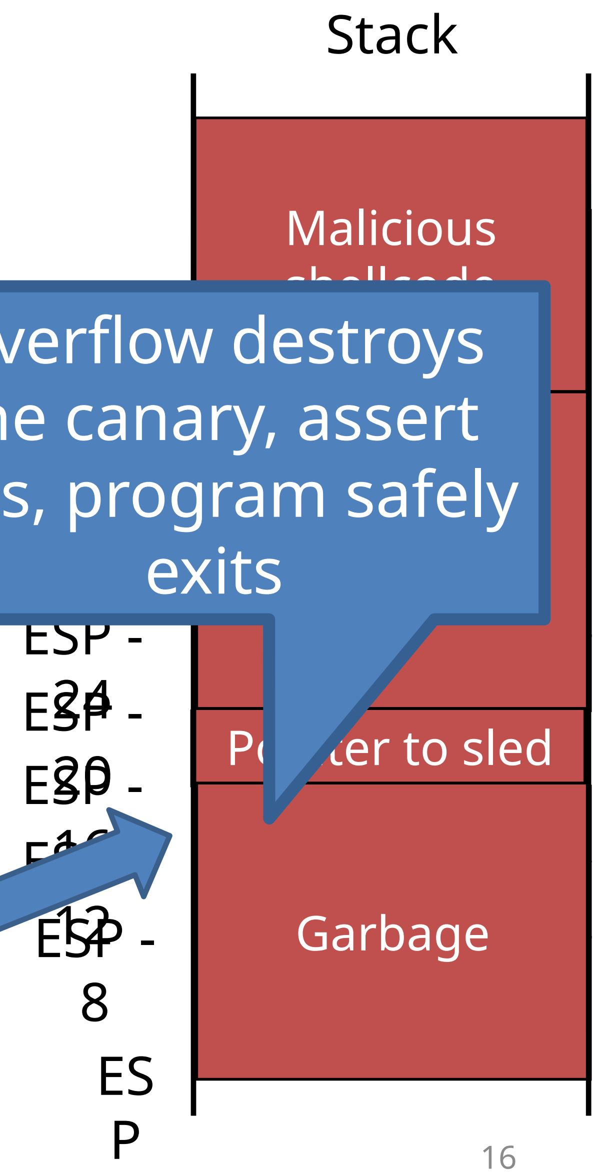
Stack Canaries

- A stack canary is an early warning system that alerts you to stack overflows

Automatically added by the compiler

```
int canary = secret_canary;
char buf[8];
for (x = 1; x < argc; ++x) {
        strcpy(buf, argv[x]);
        num = atoi(buf);
        check_for_secret(num);
}
...
assert(canary==secret_canary);
return 0;
```

Overflow destroys the canary, assert fails, program safely exits



Canary Implementation

- Canary code and data are inserted by the compiler
  - gcc supports canaries
  - Disable using the –fno-stack-protector argument
- Canary secret must be random
  - Otherwise the attacker could guess it
- Canary secret is stored on its own page at semi-random location in virtual memory
  - Makes it difficult to locate and read from memory

Canaries in Action

```
[cbw@finalfight game] ./guessinggame AAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./guessinggame terminated
Segmentation fault (core dumped)
```

- Note: canaries do not prevent the buffer overflow
- The canary prevents the overflow from being exploited

When Canaries Fail

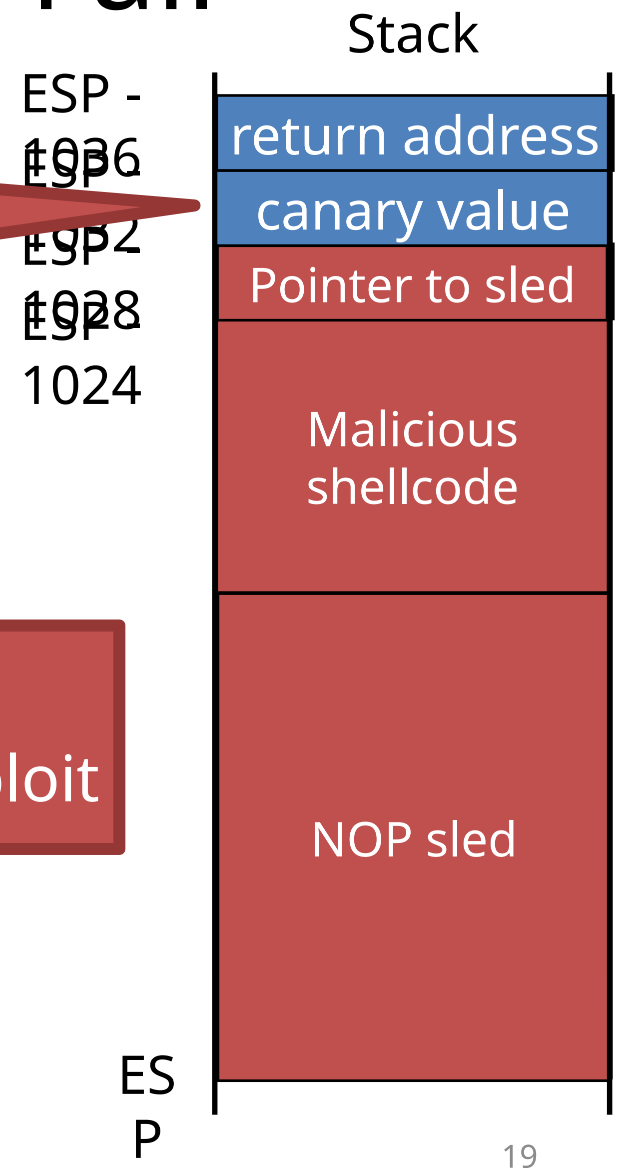
```
void my_func() { ... }

int canary = secret_canary;
void (*fptr)(void);
char buf[1024];
fptr = &my_func;
strcpy(buf, argv[1]);
fptr();
assert(canary==secret_canary);
return 0;
```

Function pointer

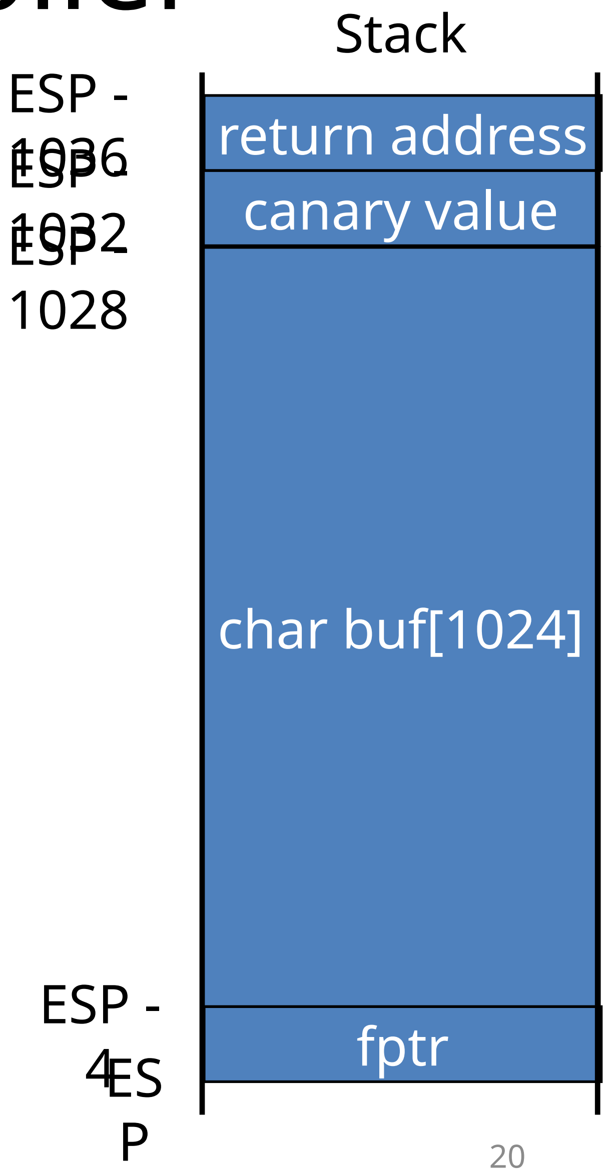
Canary is left intact

Calling fptr triggers the exploit



ProPolice Compiler

- Security oriented compiler technique
- Attempts to place arrays above other local variables on the stack
- Integrated into gcc



When ProPolice Fails

```c
void my_func() { ... }

struct my_stuff {
      void (*fptr)(void);
      char buf[1024];
};

int canary = secret_canary;
struct my_stuff stuff;
stuff.fptr = &my_func;
strcpy(stuff.buf, argv[1]);
stuff.fptr();
assert(canary==secret_canary);
return 0;
```

- The C specification states that the fields of a struct cannot be reordered by the compiler

Non-Executable Stack

- Problem: compiler techniques cannot prevent all stack-based exploits
- Key insight: many exploits require placing code in the stack and executing it
  - Code doesn’t typically go on stack pages
- Solution: make stack pages non-executable
  - Compiler marks stack segment as non-executable
  - Loader sets the corresponding page as non-executable

x86 Page Table Entry, Again

- On x86, page table entries (PTE) are 4 bytes

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

- W bit determines writeable status
- … but there is no bit for executable/non-executable
- On x86-64, the most significant bit of each PTE (bit 63) determines if a page is executable
  - AMD calls it the NX bit: No-eXecute
  - Intel calls it the XD bit: eXecute Disable

When NX bits Fail

- NX prevents shellcode from being placed on the stack
  - NX must be enabled by the process
  - NX must be supported by the OS
- Can exploit writers get around NX?
  - Of course ;)
  - Return-to-libc
  - Return-oriented programming (ROP)

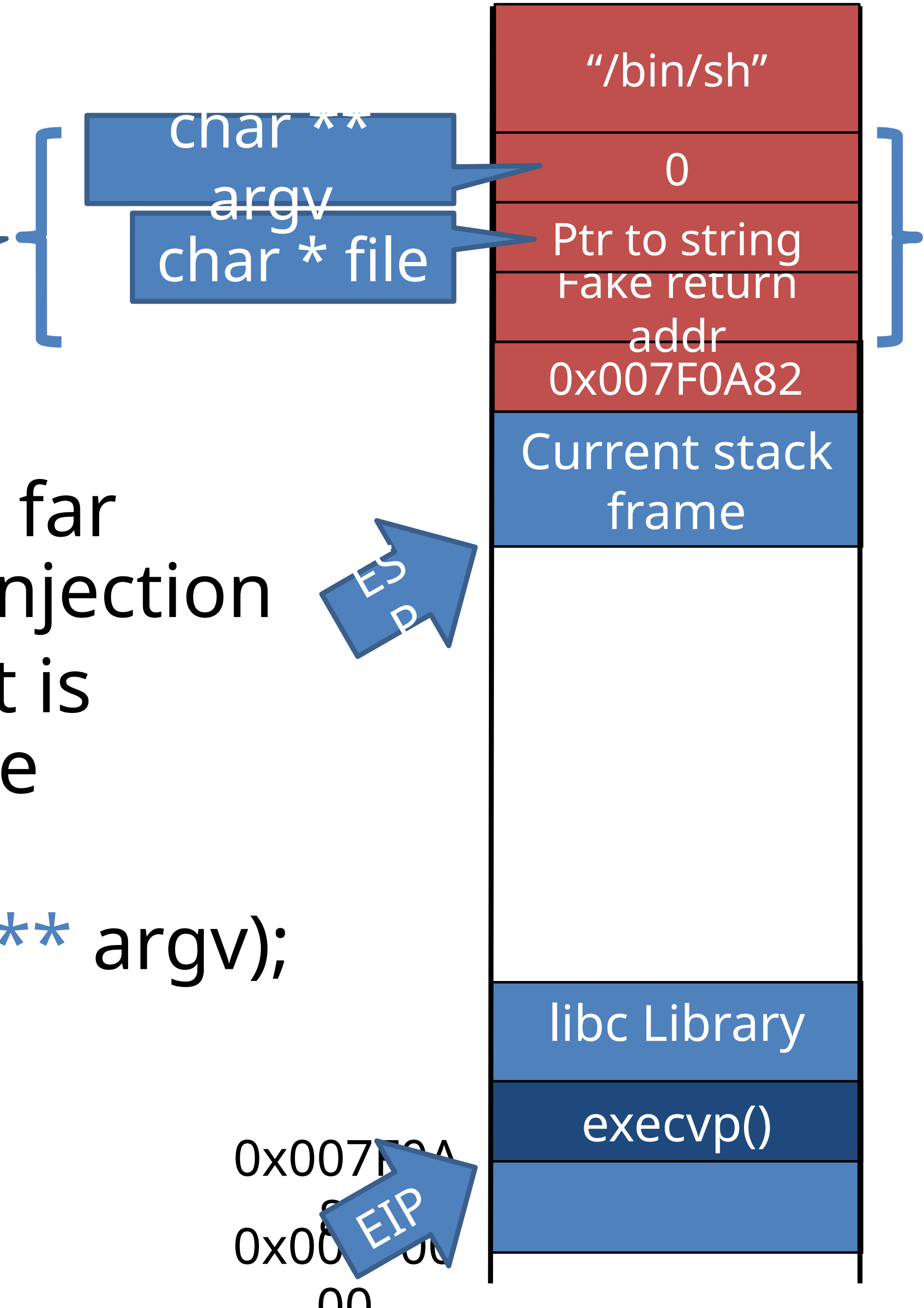
- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP
- Kernel Exploits and Rootkits

Return to libc

Parameters for a call to execvp()

- Example exploits thus far have leveraged code injection
- Why not use code that is already available in the process?

execvp(char * file, char ** argv);



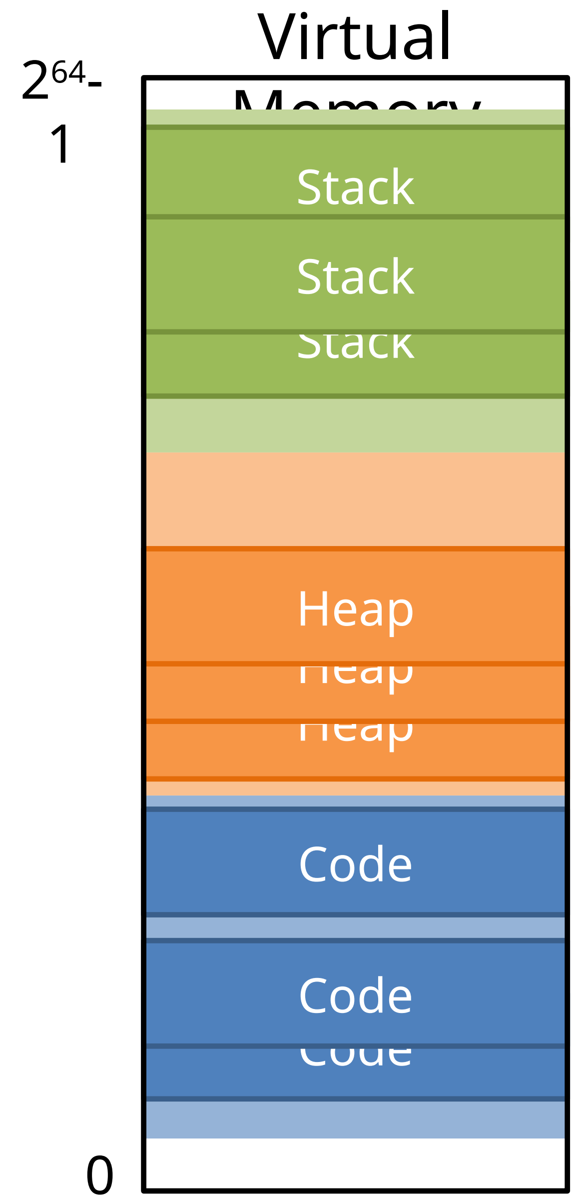
Stack Control = Program Control

- Return to libc works by crafting special stack frames and using existing library code
  - No need to inject code, just data onto the stack
- Return-oriented programming (ROP) is a generalization of return to libc
  - Why only jump to existing functions?
  - You can jump to code **anywhere** in the program
  - Gadgets are snippets of assembly that form a Turing complete language
  - Gadgets + control of the stack = arbitrary code execution power

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP
- Kernel Exploits and Rootkits

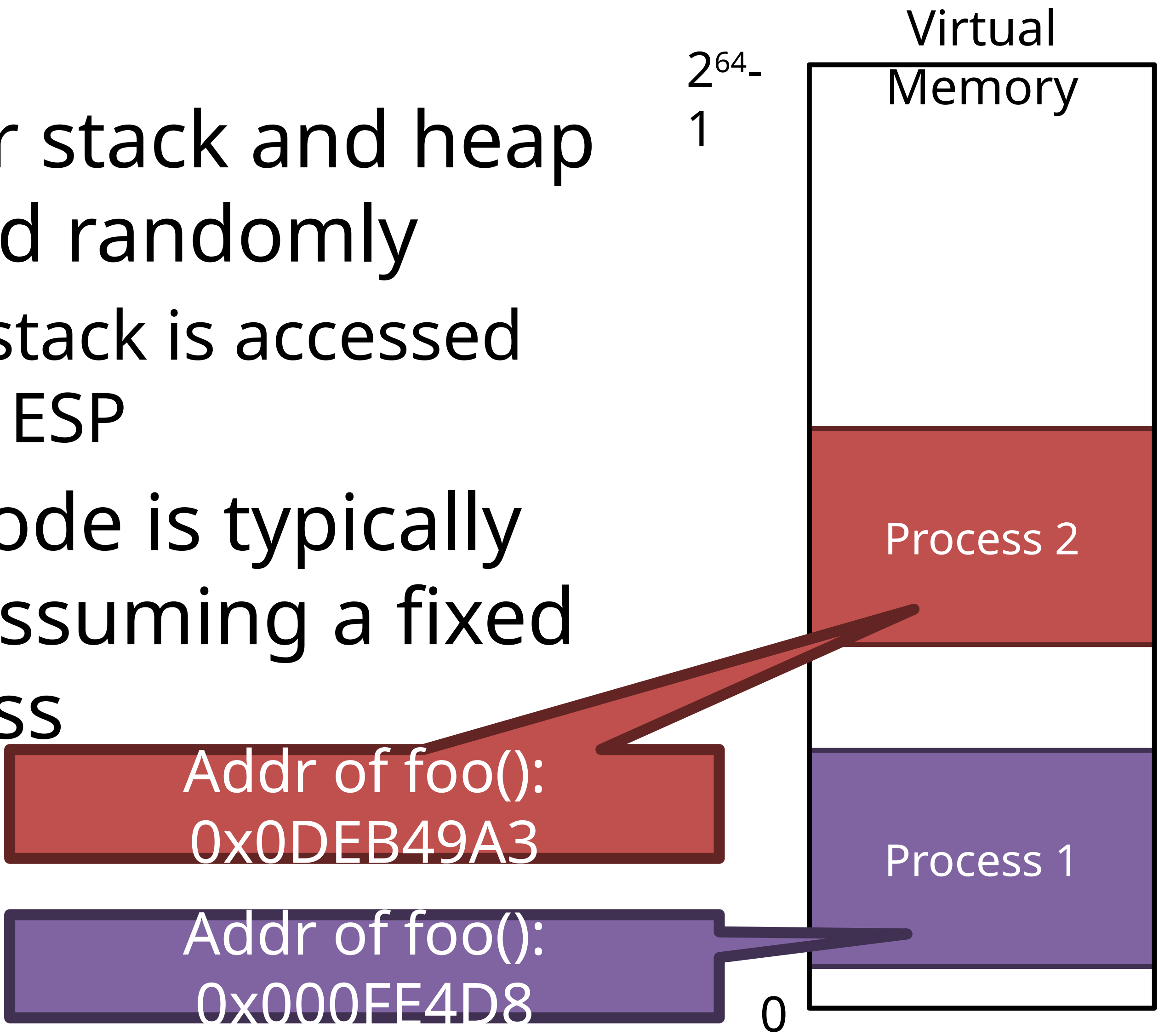
Defending Against Return to libc

- Return to libc and ROP work by repeatedly returning to known pieces of code
  - This assumes the attacker knows the addresses of this code in memory
- Key idea: place code and data at **random** places in memory
  - Address Space Layout Randomization (ASLR)
  - Supported by all modern OSes



Randomizing Code Placement

- It's okay for stack and heap to be placed randomly
  - Example: stack is accessed relative to ESP
- Problem: code is typically compiled assuming a fixed load address



Position Independent Code Example

- Modern c
  Independ
  - Also call
    Executable (PIE)

```
int global_var = 20;

int func() { return 30; }

int main() {
  int x = func();
  global_var = 10;
  return 0;
}
```

- function call
- Address is calculated as EIP + given value
- Example: 0x4004cc + 0xffffffe8 = 0x4004b4

Global data is accessed relative to EIP

```
4004c0:
mov    ebp, esp
4004c3:
sub    esp, 0x10
4004c7:
call  4004b4 <func>
4004cc:
mov [ebp-0x4], eax
4004cf:                         c7 05 3f 0b 20 00 10
mov
[eip+0x200b3f], 0x10
4004d6:                                  00 00 00
4004d9:                                  b8 00 00 00 00
mov    eax, 0x0
4004de:                                  c9
leave
4004df:                                  c3
ret
```

Tradeoffs with PIC/PIE

- Pro
  - Enables the OS to place the code and data segments at a random place in memory (ASLR)
- Con
  - Code is slightly less efficient
  - Some addresses must be calculated
- In general, the security benefits of ASLR far outweigh the cost

When ASLR Fails

- ASLR is much less effective on 32-bit architectures
  - Less ability to move pages around randomly
  - May allow the attacker to brute-force the exploit
- Use a huge NOP sled
  - If the sled is enormous, even a random jump will hit it
- Use heap spraying
  - Technique that creates many, many, many copies of shellcode in memory
  - Attempts to fill all available heap memory
  - Jump to a random address is likely to hit a copy

Exploitation Prevention Wrap-up

- Modern OSes and compilers implement many strategies to prevent exploitation
  - More advanced techniques exist and are under development
- Exploitation strategies are also becoming more sophisticated
  - Just scratched the surface of attack strategies
- Bottom line: **don’t write buggy code**
  - Compiler and OS techniques don’t fix bugs, they just try to prevent exploitation
  - Even minor flaws can be exploited

Strategies for Writing Secure Code

- **Assume all external data is under the control of an attacker**
- Avoid unsafe library calls
  - strcpy(), memcpy(), gets(), etc.
  - Use bounded versions instead, i.e. strncpy()
- Use static analysis tools, e.g. Valgrind
- Use a fuzzer
  - Runs your program repeatedly with crafted inputs
  - Designed to trigger flaws
- Use security best-practices
  - Drop privileges, use chroot jails, etc.

- Basic Program Exploitation
- Protecting the Stack
- Advanced Program Exploitation
- Defenses Against ROP
- Kernel Exploits and Rootkits