Unit: Software Security & Trusted Systems

B.Tech VIII

NETWORK AND SYSTEM SECURITY (CORE ELECTIVE - 5) (CS424)

Book :

Computer Security: Principles and Practice
By William Stallings

Chapter 10: Buffer Overflow

Chapter 11 Software Security

Chapter 12 Operating System Security

Chapter 13 Trusted Computing and Multilevel Security

Contents

- **Chapter 10: Buffer Overflow**
  - **Buffer Overflow**
  - Stack Overflows
  - Shellcode
  - Various defenses against buffer overflow attacks
  - A range of other types of buffer overflow attacks

Software Security

- **NIST Definition of Computer Security**
- "The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)"

Software Security

- Any compromise to integrity, authentication and availability makes a software unsecure.
- Software systems can be attacked to steal information, monitor content, introduce vulnerabilities and damage the behavior of software.
- Software security is the concept of implementing mechanisms in the construction of security to help it remain functional (or resistant) to attacks.
- i.e. before a piece of software is released to the public, it is subjected to software security testing to determine its ability to withstand hacking attempts.

Software Security

- The goal of software security is to create secure software from the start without requiring additional security components to add additional layers of protection.
- Programmers and engineers in the development stage must put up time and effort to accomplish software security before software launched into the market.

Software Security vs. Application Security

- The terms – **software security and application security** are very near to each other.
- However, the **distinction** between them is :
  - Software security vulnerabilities must be taken care of before the software is deployed and sent to the end-users. This requires effort and commitment from programmers and engineers in the development stage.
  - Application security is the additional tools require to provide security to application once it is deployed to market.

Software Security – Some common attacks

- **Buffer overflow, stack overflow:**
  - Buffer and stack overflow attacks overwrite the contents of the heap or stack respectively by writing extra bytes.
- **Command injection:**
  - It can be achieved on the software code when system commands are used predominantly.
  - New system commands are appended to existing commands by the malicious attack.
  - Sometimes it may stop services and cause DoS.
- **SQL injection:**
  - It uses malicious SQL code to retrieve or modify important information from database servers.
  - It can be used to bypass login credentials.
  - It fetches important information from a database or delete all important data from a database.

Buffer Overflow

- A **buffer overflow**, is also known as a **buffer overrun** or **buffer overwrite.**
- **NIST defines buffer overflow as**
  - "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."

Buffer Overflow

- A very common attack mechanism
  - Started from 1988 Morris Worm to Code Red, Slammer, Sasser and many others..
  - It is due to **careless programming in applications**.

Table 10.1 **A Brief History of Some 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).

Buffer Overflow

- A very common attack mechanism
  - Started from 1988 Morris Worm to Code Red, Slammer, Sasser and many others..
  - It is due to **careless programming in applications.**
- Techniques for preventing buffer overflow occurrence are well known and documented.
- Still of major concern to security practitioners due to
  - Legacy of buggy code in widely deployed operating systems and applications.
  - Continued careless programming techniques by programmers.

Buffer Overflow Basics

- Caused by **programming error**
- When a process attempts to store data beyond the limits of a fixed-sized buffer, and consequently overwrites adjacent memory locations.
  - Buffer can be on stack, heap, global data
- **Consequences** of overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

Buffer Overflow Basics

- Sometimes it is done deliberately
  - To attack on a system
  - To transfer the control of execution to the attacker's chosen location of the code segment, resulting in the ability to execute arbitrary code with the privileges of the attacked process.

Buffer Overflow Example

- C main function containing three variables
  - valid, str1, and str2 – Assume these values will be saved in adjacent memory locations.
  - next_tag(str1) - to copy some expected tag value(Assume "START") into str1

```
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);
}
```

(a) Basic buffer overflow C code

Buffer Overflow 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);
}
```

(a) Basic buffer overflow C code

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

Figure 10.2 Basic Buffer Overflow Stack Values

Buffer Overflow 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);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs

Buffer Overflow Example

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

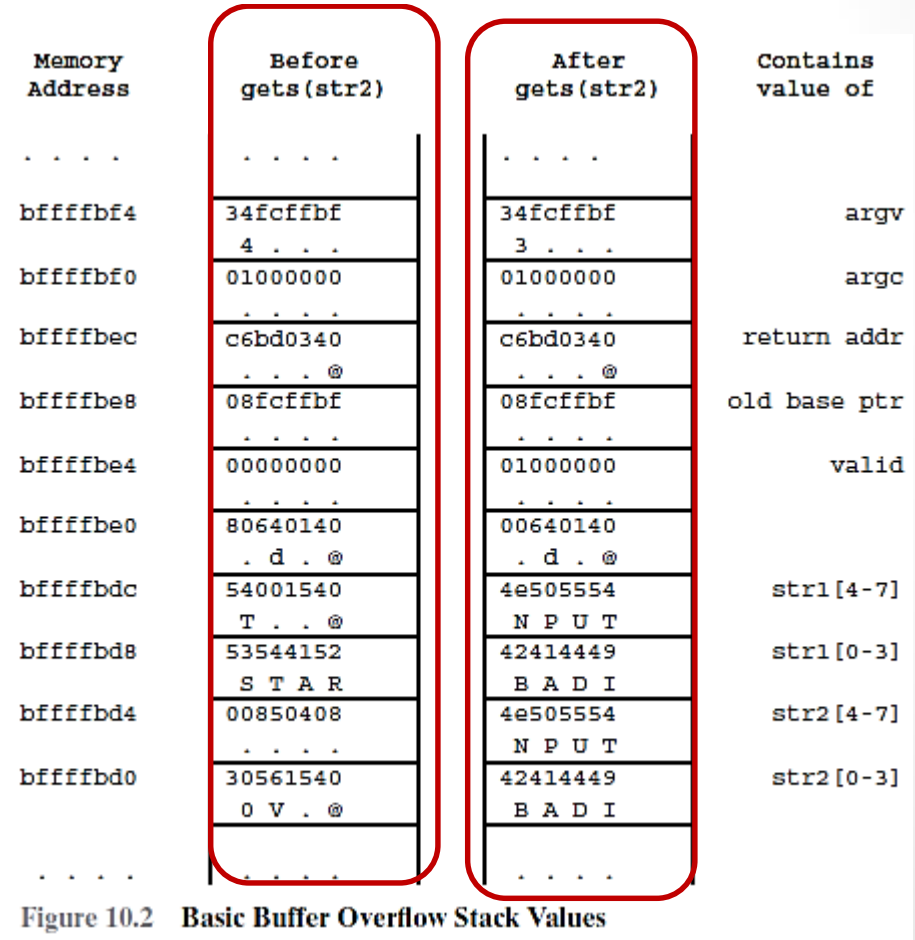


Figure 10.2 **Basic Buffer Overflow Stack Values**

Buffer Overflow Example

- **What is the Problem with 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);
}
```

- C library `gets()` function does not include any checking on the amount of data copied.
- If more than 7 characters are present on the input line, they require more room than is available in the `str2` buffer.
- Consequently the extra characters will proceed to overwrite the values of the adjacent variable, `str1` in this case.

Buffer Overflow Example

- **What is the Problem with example?**

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

- C library `gets()` function does not include any checking on the amount of data copied.
- If more than 7 characters are present on the input line, they require more room than is available in the `str2` buffer.
- Consequently the extra characters will proceed to overwrite the values of the adjacent variable, `str1` in this case.

Buffer Overflow Example

- **How attack is possible ?**

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

- Assume that a valid password is stored in the str1 buffer, instead of "START" tag.
- By supplying a long string an attacker performs buffer overflow and overwrites a valid password.
- Strncmp() generates Valid=1 and so the attacker can access privileged features without knowing a valid password.

Buffer Overflow Attacks

- To exploit a buffer overflow an attacker
  - must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - understand how buffer is stored in memory and determine potential for corruption

A Little Programming Language History

- At machine level all data an array of bytes
  - interpretation depends on instructions used
- Modern high-level languages have a strong notion of type and valid operations
  - not vulnerable to buffer overflows
  - does incur overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
  - hence are vulnerable to buffer overflow
  - have a large legacy of widely used, unsafe, and hence vulnerable code

Function Calls and Stack Frames

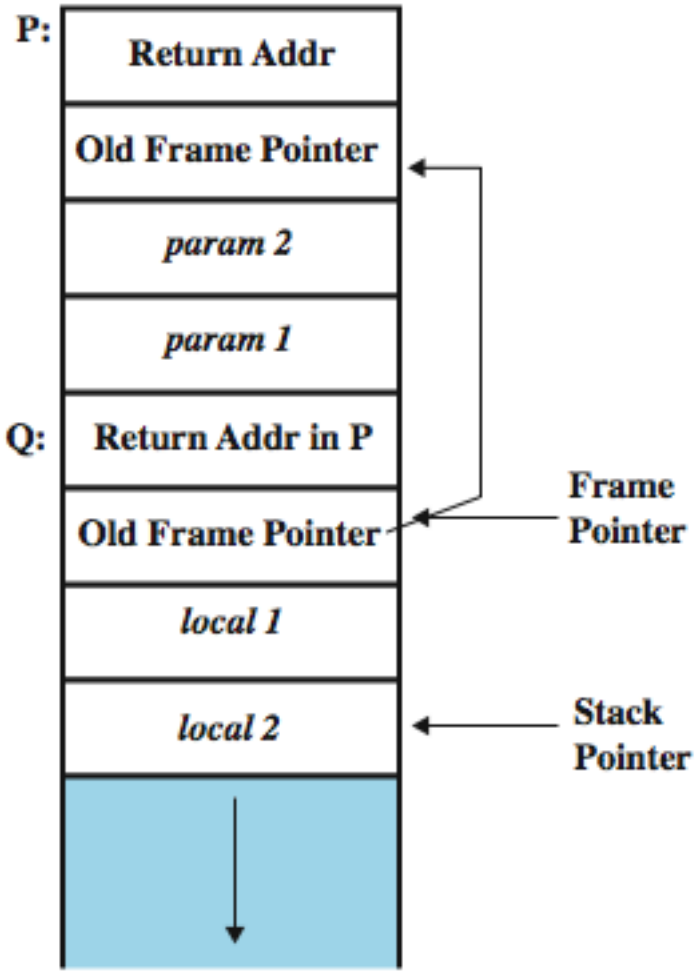
- Some important Concepts
  - When one function calls another, it needs to save the return address at some safe place so the called function can return control when it finishes.
  - It also needs locations to save the parameters to be passed in to the called function, and also possibly to save register values that it wishes to continue using when the called function returns.
  - All of this data is usually saved on the stack in a structure known as a **stack frame.**
  - Also there is need of chaining these frames together.

Function Calls and Stack Frames

Assume a function P calls Q
Stack frame: two frames

Calling function P: needs a data structure to store the "return" address and parameters to be Passed.

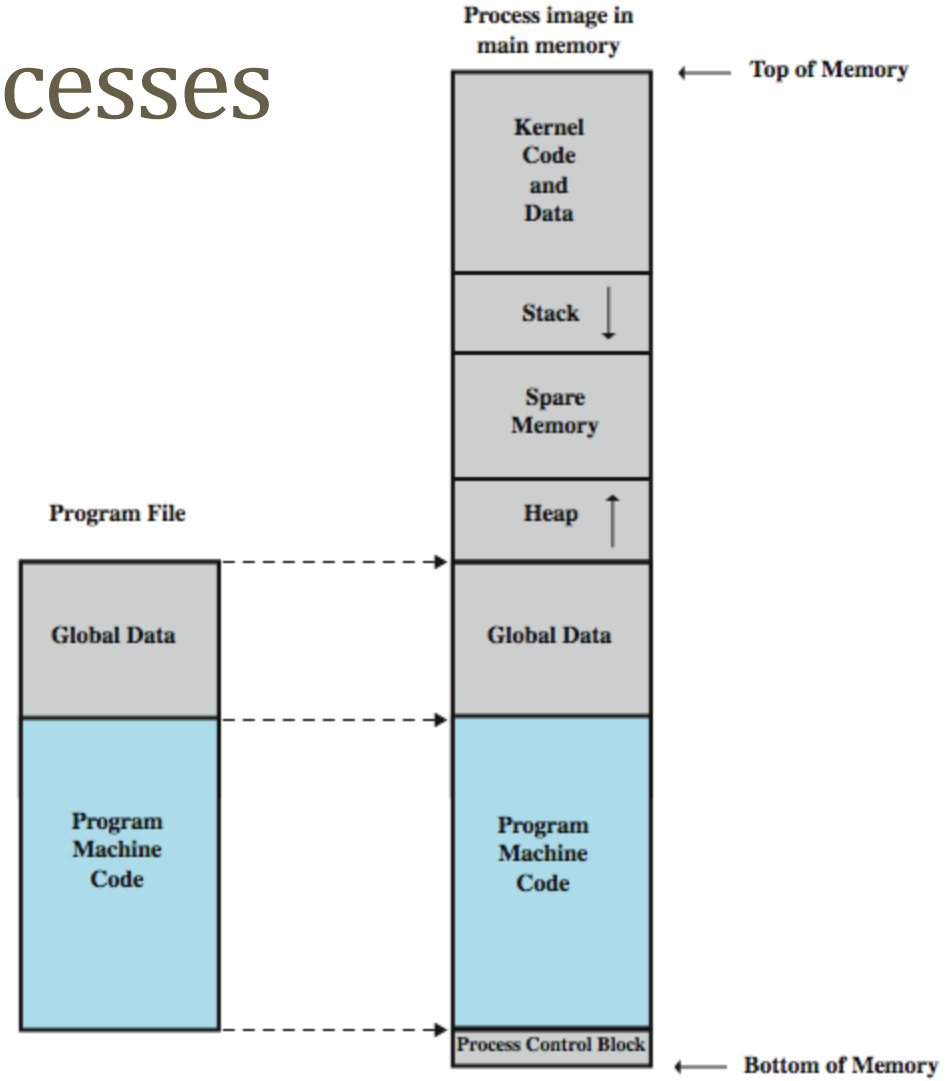
Called function Q: needs a place to store its local variables somewhere different for every call.



Stack Buffer Overflow

- Occurs when buffer is located on stack
  - also referred to as **stack smashing**
  - used by Morris Worm
  - "Smashing the Stack" paper popularized it
- Have local variables below saved frame pointer and return address
  - hence overflow of a local buffer can potentially overwrite these key control items
- Attacker overwrites return address with address of desired code
  - program, system library or loaded in buffer

Programs and Processes



Stack Buffer Overflow Example

- Consider an example
- It contains a single local variable, the buffer "inp".

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

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

- Output for input "Bill and Lawrie" where length <=16.

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

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

Stack Buffer Overflow Example

- Consider an example
- It contains a single local variable, the buffer "inp".

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

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

- Output for input "XXXXXX……." where length >16.
- It generates 'segmentation fault' as it overwrites the saved frame pointer and return address with garbage values

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

Stack Buffer Overflow Example

- Consider an example
- It contains a single local variable, the buffer “inp”.

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

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

- Rather than this, the attacker could be interested in transfer control to a location and code of the attacker’s choosing, rather than immediately crashing the program.
- The simplest way of doing this is for the input causing the buffer overflow to contain the desired target address at the point where it will overwrite the saved return address in the stack frame.
- Then when the attacked function finishes and executes the return instruction, instead of returning to the calling function, it will jump to the supplied address instead and execute instructions from there.

Stack Buffer Overflow Example

- Consider an example
- It contains a single local variable, the buffer "inp".

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

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

- Here, return address is "0x08048394"

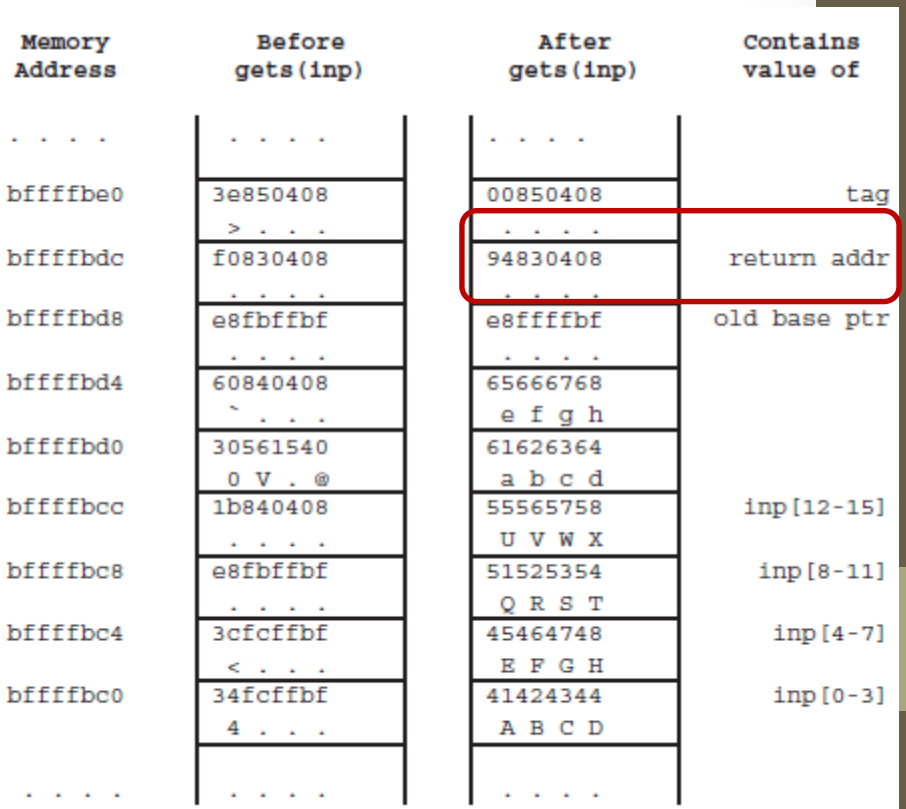


Figure 10.6 **Basic Stack Overflow Stack Values**

Stack Buffer Overflow Example

- Consider an example
- It contains a single local variable, the buffer “inp”.

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

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

- Now consider the input of size 24 “ABCDEFGHQRSTUVWXabcdefgyuy” for value of “inp” buffer.
- To overwrite the return address, frame ptr is also need to be overwritten wit some valid address. Suppose that valid address is “0x8fcffbf”.
- Because the aim of this attack is to cause the hello function to be called again, a second line of input is included for it to read on the second run, namely the string NNNN, along with newline characters at the end of each line.

Stack Buffer Overflow Example

- Consider an example

```
$ 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)
```

- Here, the example of perl command is used where `pack()` function is used to convert a hexadecimal string into its binary equivalent. This output is then piped into the targeted buffer2 program.

Another Stack Overflow

```
void getinp(char *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");
}
```

Safe input function; output may still overwrite part of the stack frame (sprintf creates formatted value for a var)

Another Stack Overflow

```
$ cc -o buffer3 buffer3.c

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

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

Safe input function; output may still overwrite part of the stack frame

Common Unsafe C Functions

`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

Unix Shellcode

- In Windows terms: `command.exe`

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

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

```
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
```

Unix Shellcode

```
        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) t0 %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
```

Shellcode

- code supplied by attacker
  - often saved in buffer being overflowed
  - traditionally transferred control to a shell
- machine code
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently have automated sites/tools

Buffer Overflow Defenses

- Buffer overflows are widely exploited
- Large amount of vulnerable code in use
  - despite cause and countermeasures known
- Two broad defense approaches
  - compile-time - harden new programs
  - run-time - handle attacks on existing programs

Compile-Time Defenses: Programming Language

- Use a modern high-level languages with strong typing
  - not vulnerable to buffer overflow
  - compiler enforces range checks and permissible operations on variables
- Do have cost in resource use
- And restrictions on access to hardware
  - so still need some code in C like languages

Compile-Time Defenses: Safe Coding Techniques

- If using potentially unsafe languages eg C
- Programmer must explicitly write safe code
  - by design with new code
  - ***extensive after code review*** of existing code, (e.g., OpenBSD)
- Buffer overflow safety a subset of general safe coding techniques
- Allow for graceful failure ***(know how things may go wrong)***
  - check for sufficient space in any buffer

Compile-Time Defenses: Language Extension, Safe Libraries

- Proposals for safety extensions (library replacements) to C
  - performance penalties
  - must compile programs with special compiler
- Several safer standard library variants
  - new functions, e.g. strlcpy()
  - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe

Compile-Time Defenses: Stack Protection

- Stackgaurd: add function entry and exit code to check stack for signs of corruption
  - Use random canary
  - e.g. Stackguard, Win/GS, GCC
  - check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support
- Or save/check safe copy of return address (in a safe, non-corruptible memory area), e.g. Stackshield, RAD

Run-Time Defenses: Non Executable Address Space

- Many BO attacks copy machine code into buffer and xfer ctrl to it
- Use virtual memory support to make some regions of memory non-executable (to avoid exec of attacker's code)
  - e.g. stack, heap, global data
  - need h/w support in MMU
  - long existed on SPARC/Solaris systems
  - recent on x86 Linux/Unix/Windows systems
- Issues: support for executable stack code

Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - stack, heap, global data: change address by 1 MB
  - using random shift for each process
  - have large address range on modern systems means wasting some has negligible impact
- Randomize location of heap buffers and location of standard library functions

Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory (or between stack frames)
  - flagged in MMU (mem mgmt unit) as illegal addresses
  - any access aborts process
- Can even place between stack frames and heap buffers
  - at execution time and space cost

Other Overflow Attacks

- have a range of other attack variants
  - stack overflow variants
  - heap overflow
  - global data overflow
  - format string overflow
  - integer overflow
- more likely to be discovered in future
- some cannot be prevented except by coding to prevent originally

Summary

- Introduced basic buffer overflow attacks
- Stack buffer overflow details
- Shellcode
- Defenses
  - compile-time, run-time
- Other related forms of attack (not covered)
  - replacement stack frame, return to system call, heap overflow, global data overflow