Information Security
CS 3002

Dr. Haroon Mahmood

Assistant Professor

NUCES Lahore

NIST’s Definition: Buffer overflow

“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 Basics

- Caused by programming error
- Allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- Overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

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", st  r1, 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)
```

Buffer Overflow Example

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]
. . . .	. . . .	. . . .	

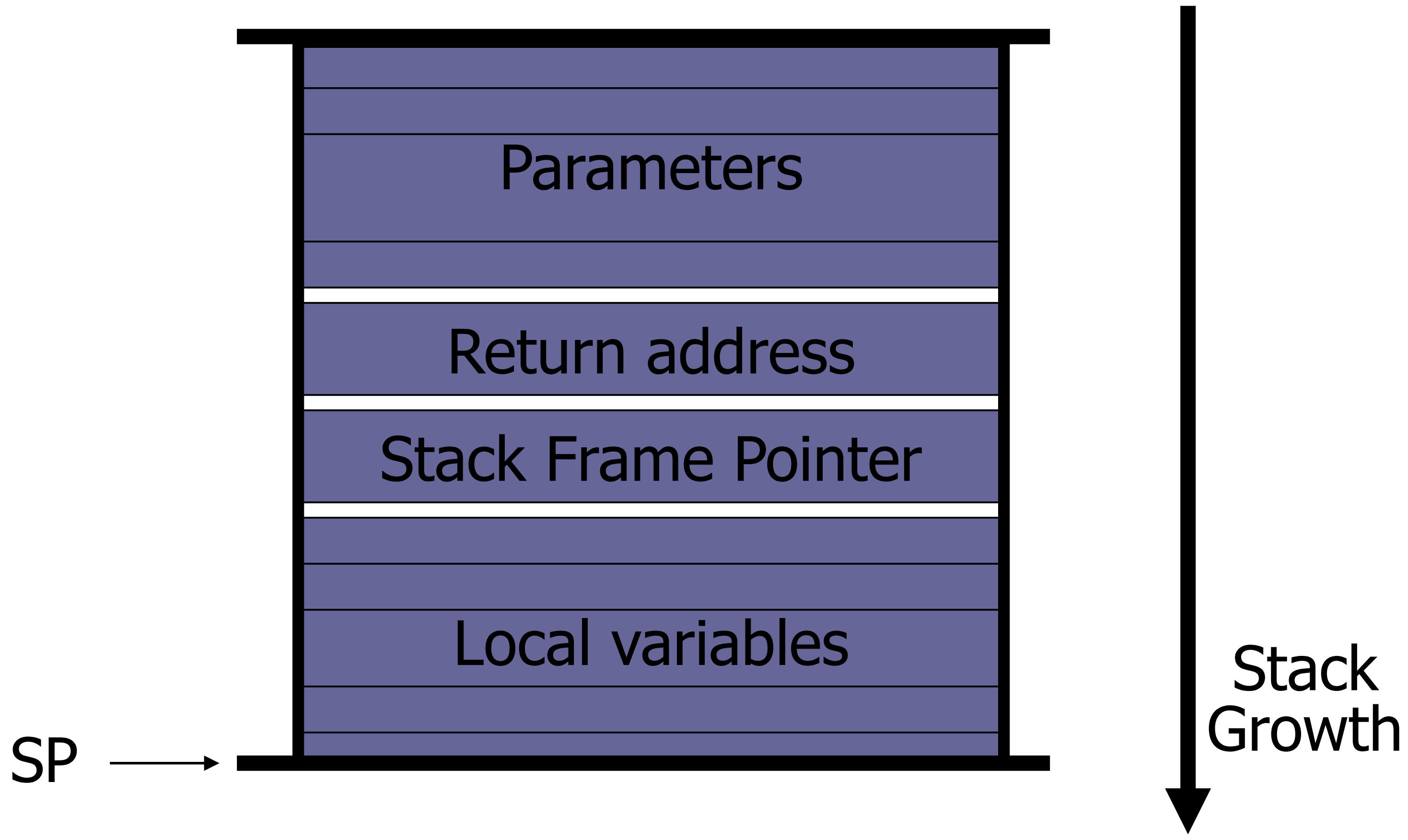
Buffer Overflow Attacks

- 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

- At machine level, all data is 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

Stack Growth

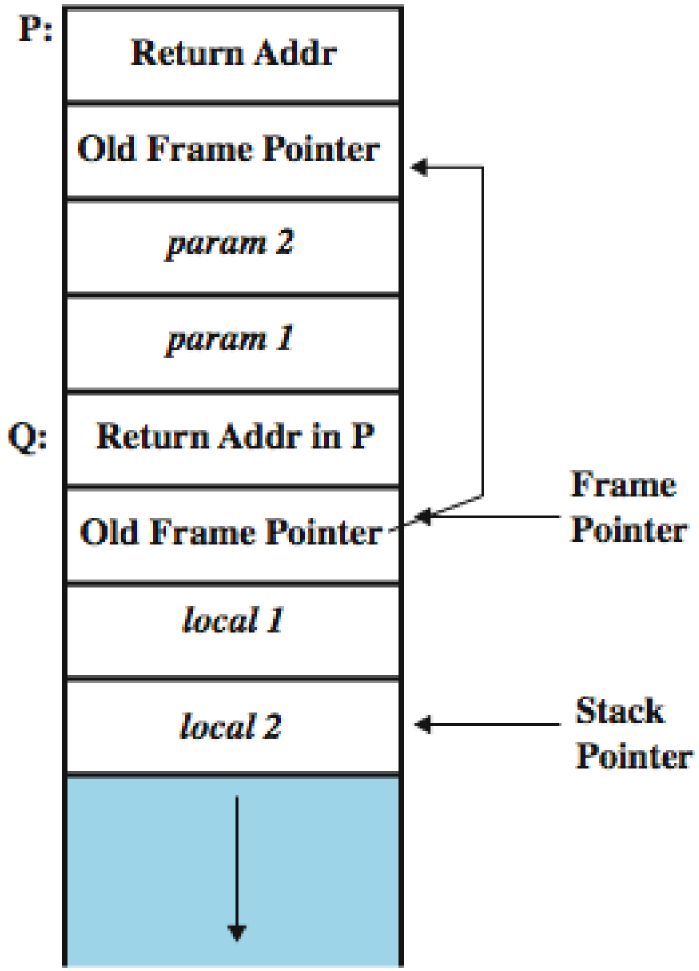


Function Calls and Stack Frames

Stack frame:

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

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



Stack Buffer Overflow

- Occurs when buffer is located on stack
  - 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

What Happens in a Function Call?

```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    ….
}
int main() {
   func("abc");
}
```

- Before main() calls func()
  - Push pointer to “abc” onto stack
  - Use “call func” assembly, which pushes current IP on stack
- Upon entering func()
  - Push stack frame pointer register (bp) on stack
  - Update sp to leave space for local variable.
- Upon leaving func()
  - Update sp to just below saved bp
  - Pop stack to bp, restore bp
  - Use “ret” assembly, which pop stack to IP

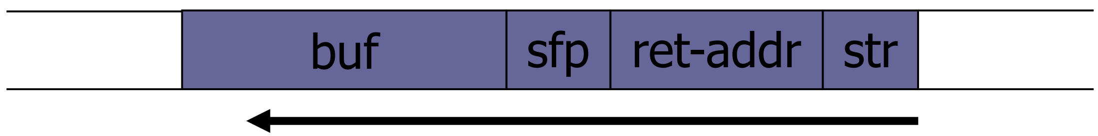
What are buffer overflows?

- Suppose a web server contains a function:

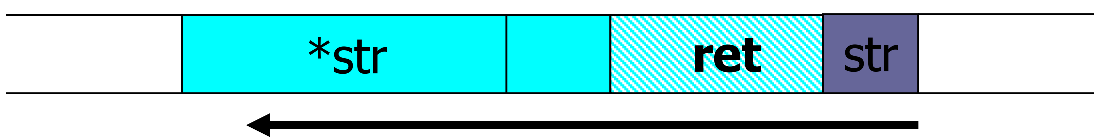
```
void func(char *str) {
  char buf[128];

    strcpy(buf, str);
  do-something(buf);
}
```

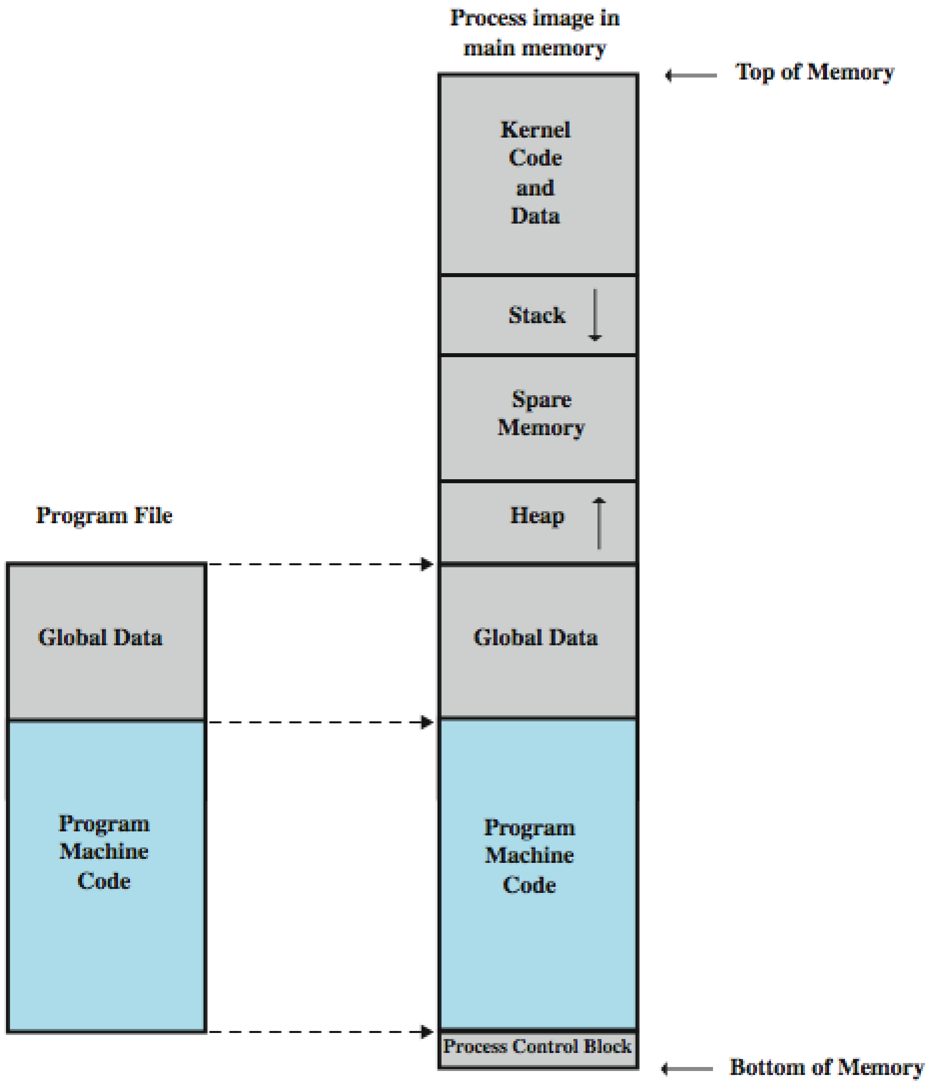
- When the function is invoked the stack looks like:



- What if `*str` is 136 bytes long? After `strcpy`:



Programs and Processes



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
buffer3getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

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

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 e.g. 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 transfer 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
- some cannot be prevented except by coding to prevent originally

Integer Overflow

- **Integer overflow: an arithmetic operation attempts to create a numeric value that is larger than can be represented within the available storage space.**

- **Example:**

Test 1:

```
short x = 30000;
short y = 30000;
printf("%d\n", x+y);
```

Test 2:

```
short x = 30000;
short y = 30000;
short z = x + y;
printf("%d\n", z);
```

Will two programs output the same?
Assuming short uses 16 bits.
What will they output?

Where Does Integer Overflow Matter?

- **Allocating spaces using calculation.**
- **Calculating indexes into arrays**
- **Checking whether an overflow could occur**

- **Direct causes:**
  - **Truncation; Integer casting**

C Data Types

short int	16bits	[-32,768; 32,767]
unsigned short int	16bits	[0; 65,535]
unsigned int	16bits	[0; 4,294,967,295]
Int	32bits	[-2,147,483,648; 2,147,483,647]
long int	32 bits	[-2,147,483,648; 2,147,483,647]
char 8 bits	[0; 255]	

Integer Overflow Vulnerabilities

- **Example:**

```
const long MAX_LEN = 20K;
Char   buf[MAX_LEN];
short len = strlen(input);
if (len < MAX_LEN)   strcpy(buf, input);
```

Can a buffer overflow attack occur?

If so, how long does input needs to be?

Another Integer Overflow

```
int copy_something(char *buf, int len){
  char kbuf[800];
  if(len > sizeof(kbuf)){ /* [1] */
    return -1;
  }
  return memcpy(kbuf, buf, len); /* [2] */
}
```

What could go wrong?

Format string attack

```
int  func(char *user)  {
   fprintf( stdout, user);
}
```

Problem: what if user = “%s%s%s%s%s%s%s” ??

- Most likely program will crash: DoS.
- If not, program will print memory contents. Privacy?
- Full exploit using user = “%n”

Correct form:

```
int  func(char *user)  {
   fprintf( stdout, "%s", user);
}
```

Vulnerable functions

Any function using a format string.

Printing:

printf, fprintf, sprintf, …

vprintf, vfprintf, vsprintf, …

Logging:

syslog, err, warn

Malicious format strings

- Interesting inputs for the string str to attack printf(str)
- %x%x%x%x%x%x%x%x
  will print bytes from the top of the stack
- %s
  will interpret the top bytes of the stack as an address X, and then prints the string starting at that address A in memory, ie. it dumps all memory from A up to the next null terminator
- %n
  will interpret the top bytes of the stack as an address X, and then writes the number of characters output so far to that address

Malicious format strings

- Leaking data from stack

```
int main( int argc, char** argv)
{
int pincode = 1234;
 printf(argv[1]);
}
```

- How can an attacker learn the value of pincode ?

- Supplying %x%x%x as input will dump top 12 bytes of the stack

Malicious format strings

- printf(”str has the value %s ” , str);
- // %s to print a string, ie a char*

- Any guess what printf(”str has the value %s ”); does?

- It interprets the top of the stack as a pointer (an address) and prints the string allocated in memory at that address
- Of course, there might not be a string allocated at that address.
- printf simply prints whatever is in memory up to the next null terminator

Malicious format strings

```
int j;
char* msg; ...
printf( ”how long is this? %n”, &j);
```

- %n causes the number of characters printed to be written to j.
- Here it will give j the value 14
- Any guess what printf(”how long is this? %n”, msg”); will do?
- It interprets the top of the stack as an address, and writes the value 14 to it