Computer Security: Principles and Practice

Chapter 10: Buffer Overflow

NIST’s Definition

- “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 Well-Known Problem

- A very common attack mechanism
  - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- Prevention techniques known
- Still of major concern due to
  - legacy of widely deployed buggy
  - continued careless programming techniques

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

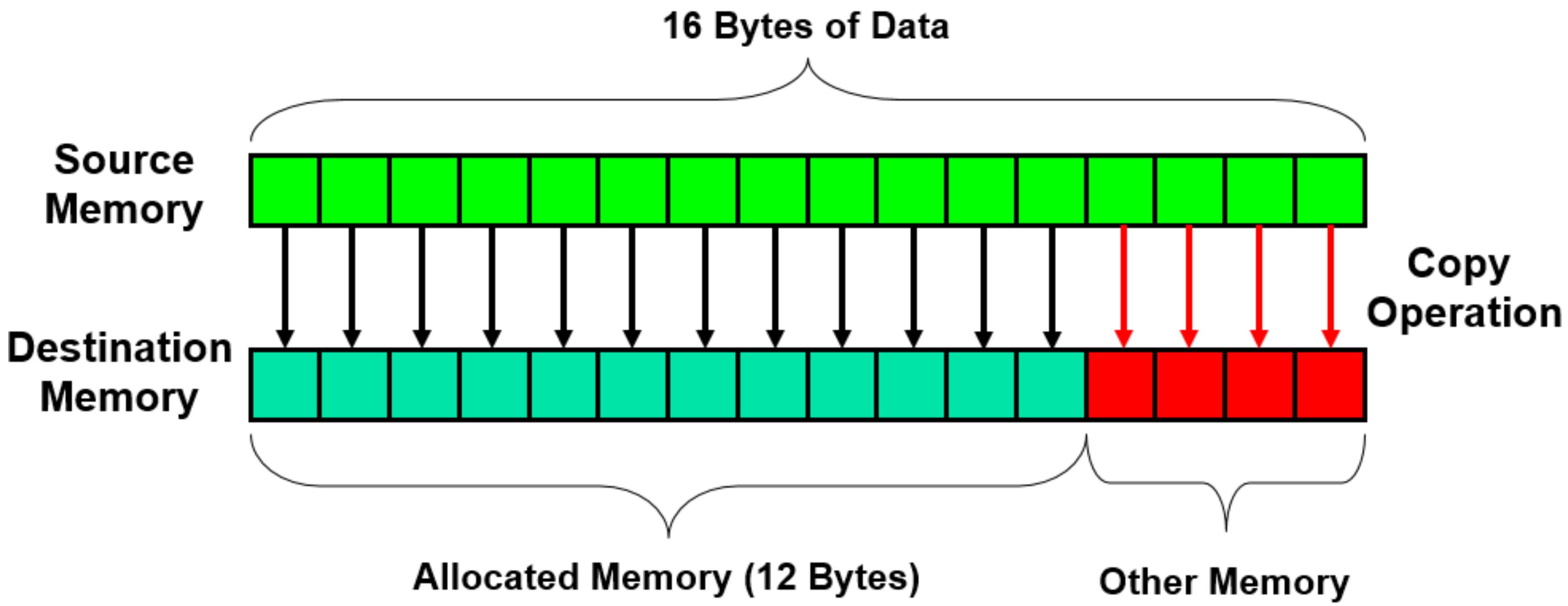
```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

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

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

Another illustration



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

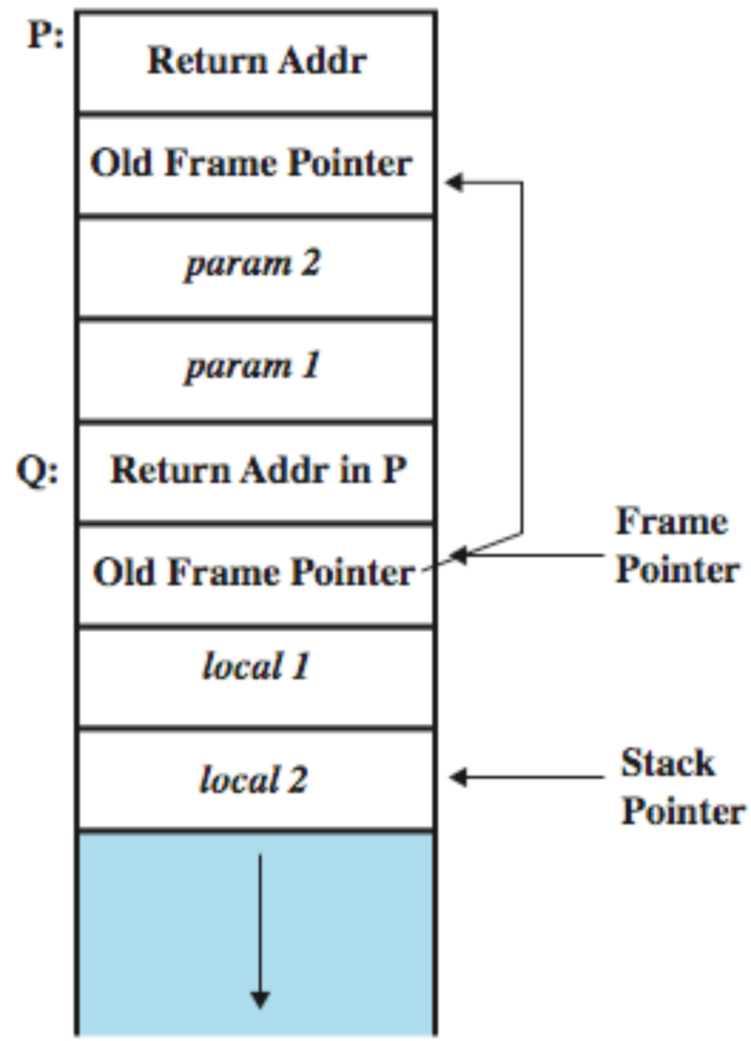
- At machine level all data an array of bytes
  - interpretation depends on instructions used
- Modern high-level languages have a strong notion of type and valid operations
  - not vulnerable to buffer overflows
  - does incur overhead, some limits on use
- C and related languages have high-level control 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

Stack frame:

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

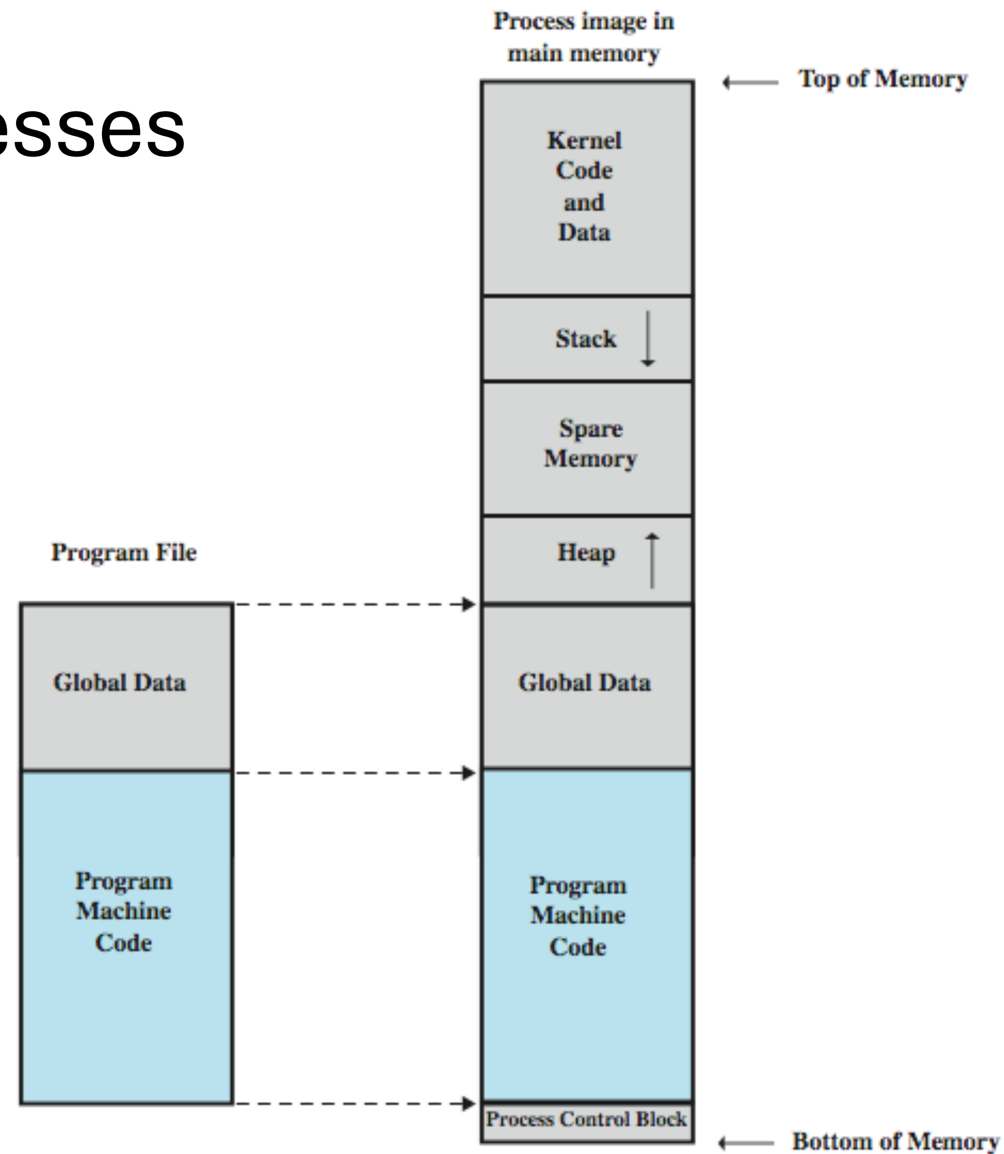
Called function: needs a place to store its local variables somewhere different for every call



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

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
buffer3     getinp read SAFE
read   val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
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