CS 6V81-05: System Security and Malicious Code Analysis
Buffer Overflow, Integer and Heap Overflow

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

April 11th, 2012
Outline

1. Background
2. Buffer Overflow
3. Integer and Heap Overflow
4. Summary
Outline

1. Background

2. Buffer Overflow

3. Integer and Heap Overflow

4. Summary
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest stack address
  - address of top element

![IA32 Stack Diagram]
Pushing

- `pushl src`
- Fetch operand at `Src`
- Decrement `%esp` by 4
- Write operand at address given by `%esp`
Popping

- **popl dest**
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to Dest
Procedure Call Example

```
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax
```

%eip is program counter
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax

%eip is program counter
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax

%eip is program counter
**Procedure Call Example**

```assembly
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax
```

- `%esp` is program counter
- `%eip` is program counter

```
0x104
0x108
0x10c
0x110

call 8048b90
```
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax

%esp = 0x108
%eip = 0x804854e

%esp = 0x108
%eip = 0x8048553

%esp = 0x104
%eip = 0x8048b90

%eip is program counter
Procedure Return Example

8048591: c3  
ret

%esp 0x104  
%eip 0x8048591

%esp 0x108  
%eip 0x8048553

%eip is program counter
Call Chain Example

- **Code Structure**

```c
yoo(…)
{
  
  who();
  
}

who(…)
{
  
  amI();
  
}

amI(…)
{
  
  amI();
  
}
```

- **Procedure amI recursive**

- **Call Chain**

```plaintext
yoo
  who
    amI
      amI
        amI
```

Carnegie Mellon

10
Call Chain Example

Contents
- Local variables
- Return information
- Temporary space

Management
- Space allocated when enter procedure
  - “Set-up” code
- Deallocated when return
  - “Finish” code

Pointers
- Stack pointer %esp indicates stack top
- Frame pointer %ebp indicates start of current frame
Stack Operation

```c
yoo(...) {
  .
  .
  who();
  .
  .
}
```

Frame Pointer %ebp
Stack Pointer %esp

Call Chain

Stack Operation
Stack Operation

```c
who(...) {
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

Call Chain

- `yoo`
- `%ebp`
- `%esp`

Frame

- `yoo`
- `who`

Stack Pointer

- `who`
- `yoo`
Stack Operation

```c
amI(...) {
    .
    .
    amI();
    .
    .
}
```

Call Chain

- Frame Pointer
  - %ebp
- Stack Pointer
  - %esp

- yoo
- who
- amI
Stack Operation

```
amI (...) {
    .
    .
    amI ();
    .
    .
}
```

Call Chain

- yoo
- who
- amI
- amI

Stack Operation Diagram:

- Frame Pointer %ebp
- Stack Pointer %esp
Stack Operation

```c
amI(...) {
    •
    •
    •
    amI();
    •
    •
}
```

Call Chain

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`
- `amI`

Frame Pointer
- `%ebp`

Stack Pointer
- `%esp`
Stack Operation

```
amI(...) 
{ 
  .
  .
  amI();
  .
}
```

Call Chain

```
yoo
who
amI
amI
amI
```

Frame Pointer
%ebp
Stack Pointer
%esp

```
.
.
.
yoo
who
amI
amI
```
Stack Operation

```c
amI(...) {
    
    amI();
    
}
```

Call Chain

```
ymoo
who
amI
amI
Stack Pointer
%esp
Frame Pointer
%ebp
```

Summary

Background
Buffer Overflow
Integer and Heap Overflow
Stack Operation

```c
who(...) {
    ... 
    amI();
    ... 
    amI();
    ... 
}
```

Call Chain

- Frame Pointer: `%ebp`
- Stack Pointer: `%esp`

Diagram showing the stack operation with a call chain.
Stack Operation

```
amI (...) {
  ...
  ...
  ...
}
```

Call Chain

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`

Frame Pointer
- `%ebp`

Stack Pointer
- `%esp`
Stack Operation

```c
who(...) {
    ...;
    amI();
    ...;
    amI();
}
```

Call Chain

```
yoo
  ↓
who
  ↓
amI
  ↓
amI
  ↓
amI
  ↓
...;
  ↓
amI
  ↓
amI
  ↓
ymoo
```

Frame Pointer %ebp
Stack Pointer %esp
Stack Operation

```
yoo(...) 
{ 
  • 
  • 
  who(); 
  • 
  • 
} 
```

Call Chain

- Frame Pointer: `%ebp`
- Stack Pointer: `%esp`

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`
IA32/Linux Stack Frame

- Current Stack Frame (“Top” to Bottom)
  - Parameters for function about to call
    - “Argument build”
  - Local variables
    - If can’t keep in registers
  - Saved register context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by call instruction
  - Arguments for this call
IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E.g., local variables

- **Heap**
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()

- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code

- **Text**
  - Executable machine instructions
  - Read-only
Memory Allocation Example

```c
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB*/

int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    p1 = malloc(1 <<28); /* 256 MB */
    p2 = malloc(1 << 8); /* 256 B */
    p3 = malloc(1 <<28); /* 256 MB */
    p4 = malloc(1 << 8); /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?
**IA32 Example Addresses**

- $esp: 0xffffbcd0
- p3: 0x65586008
- p1: 0x55585008
- p4: 0x1904a110
- p2: 0x1904a008
- &p2: 0x18049760
- &beyond: 0x08049744
- big_array: 0x18049780
- huge_array: 0x08049760
- main(): 0x080483c6
- useless(): 0x08049744
- final malloc(): 0x006be166

**address range ~2^{32}**

`malloc()` is dynamically linked  
Address determined at runtime

[Diagram showing stack, text, data, and heap sections with memory addresses]
Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
- July, 1999
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers
August 1999

- Mysteriously, Messenger clients can no longer access AIM servers.
- Microsoft and AOL begin the IM war:
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes.
  - At least 13 such skirmishes.

How did it happen?

The Internet Worm and AOL/Microsoft War were both based on stack buffer overflow exploits!

- many library functions do not check argument sizes.
- allows target buffers to overflow.
String Library Code

- Implementation of Unix function `gets()`

```c
/* Get string from stdin */
char *gets(char *dest) {
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- No way to specify limit on number of characters to read
- Similar problems with other library functions
  - `strcpy`, `strcat`: Copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification
Vulnerable Buffer Code

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo() {
    echo();
}
```

unix>./bufdemo
Type a string:1234567
1234567

unix>./bufdemo
Type a string:12345678
Segmentation Fault

unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
Buffer Overflow Disassembly

**echo**

```
80485c5:  55    push %ebp
80485c6:  e5    mov %esp,%ebp
80485c8:  53    push %ebx
80485c9:  ec    sub $0x14,%esp
80485cc:  5d    lea 0xffffffff8(%ebp),%ebx
80485cf:  1c    mov %ebx,(%esp)
80485d2:  ff    call 8048575 <gets>
80485d7:  ff    mov %ebx,(%esp)
80485da:  ff    call 80483e4 <puts@plt>
80485df:  14    add $0x14,%esp
80485e2:  5b    pop %ebx
80485e3:  5d    pop %ebp
80485e4:  c3    ret
```

**call echo**

```
80485eb: e8    call 80485c5 <echo>
80485f0: c9    leave
80485f1: c3    ret
```
/* Echo Line */

void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
[3] [2] [1] [0]
Stack Frame for echo

echo:
    pushl %ebp # Save %ebp on stack
    movl %esp, %ebp
    pushl %ebx # Save %ebx
    subl $20, %esp # Allocate stack space
    leal -8(%ebp),%ebx # Compute buf as %ebp-8
    movl %ebx, (%esp) # Push buf on stack
    call gets # Call gets
    ...

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
Buffer Overflow Stack Example

```plaintext
unix> gdb bufdemo  
(gdb) break echo  
Breakpoint 1 at 0x80485c9  
(gdb) run  
Breakpoint 1, 0x80485c9 in echo ()  
(gdb) print /x $ebp  
$1 = 0xffffd678  
(gdb) print /x *(unsigned *)$ebp  
$2 = 0xffffd688  
(gdb) print /x *((unsigned *)$ebp + 1)  
$3 = 0x80485f0
```

```
<table>
<thead>
<tr>
<th>Before call to gets</th>
<th>Before call to gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Frame for main</td>
<td>Stack Frame for main</td>
</tr>
<tr>
<td>Return Address</td>
<td>0x80485eb: e8 d5 ff ff ff</td>
</tr>
<tr>
<td>Saved %ebp</td>
<td>call 80485c5 &lt;echo&gt;</td>
</tr>
<tr>
<td>Saved %ebx</td>
<td>leave</td>
</tr>
<tr>
<td>[3][2][1][0]</td>
<td>08 04 85 f0</td>
</tr>
<tr>
<td>Stack Frame for echo</td>
<td>0xfffffd688</td>
</tr>
<tr>
<td>buf</td>
<td>0xfffffd678</td>
</tr>
<tr>
<td>buf</td>
<td>xx xx xx xx</td>
</tr>
</tbody>
</table>
```

Carnegie Mellon

35

Buffer Overflow

Stack Example
Buffer Overflow Stack Example #1

Before call to gets

Stack Frame for main

Stack Frame for echo

Input 1234567

Stack Frame for main

Overflow buf, and corrupt %ebx, but no problem
Buffer Overflow Stack Example #2

Before call to gets

Stack Frame for main

08 04 85 f0
ff ff d6 88
Saved %ebx

Stack Frame for echo

xx xx xx xx buf

Input 12345678

Stack Frame for main

08 04 85 f0
ff ff d6 00

Stack Frame for echo

38 37 36 35
34 33 32 31 buf

Base pointer corrupted

... 80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Set %ebp to corrupted value
80485f1: c3 ret
Buffer Overflow Stack Example #3

Before call to gets

Stack Frame for main

```
<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>
```

Saved %ebx

```
|   xx |   xx |   xx |   xx |
```

buf

Stack Frame for echo

```
<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>
```

Input 123456789!”#

Stack Frame for main

```
|   08 |   04 |   85 |   00 |
```

```
<table>
<thead>
<tr>
<th>43</th>
<th>42</th>
<th>41</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
</tr>
</tbody>
</table>
```

```
|   34 |   33 |   32 |   31 |
```

buf

Return address corrupted

```
80485eb:   e8 d5 ff ff ff    call 80485c5 <echo>
80485f0:   c9                 leave  # Desired return point
```
Buffer Overflow Stack Example #3

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When bar() executes ret, will jump to exploit code

```c
void foo()
{
    bar();
    ...
}

int bar()
{
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```
Exploits Based on Buffer Overflows

- Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines
- Internet worm
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
  - Worm attacked fingerd server by sending phony argument:
    - `finger "exploit-code padding new-return-address"`
    - Exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
Exploits Based on Buffer Overflows

- Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines

- IM War
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
  - When Microsoft changed code to match signature, AOL changed signature location.
Code Red Exploit Code

- Starts 100 threads running
- Spread self
  - Generate random IP addresses & send attack string
  - Between 1st & 19th of month
- Attack www.whitehouse.gov
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
  - Between 21st & 27th of month
- Deface server’s home page
  - After waiting 2 hours
Avoiding Overflow Vulnerability

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

- Use library routines that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
  - Don’t use scanf with %s conversion specification
    - Use fgets to read the string
    - Or use %ns where n is a suitable integer
Randomized stack offsets
- At start of program, allocate random amount of space on stack
- Makes it difficult for hacker to predict beginning of inserted code

Nonexecutable code segments
- In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
- X86-64 added explicit “execute” permission

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffffff638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffffff6a8
```
Stack Canaries

**Idea**
- Place special value ("canary") on stack just beyond buffer
- Check for corruption before exiting function

**GCC Implementation**
- `-fstack-protector`
- `-fstack-protector-all`

```bash
unix> ./bufdemo-protected
Type a string:1234
1234
unix> ./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```
Protected Buffer Disassembly

```
echo
804864d: 55 push %ebp
804864e: 89 e5 mov %esp,%ebp
8048650: 53 push %ebx
8048651: 83 ec 14 sub $0x14,%esp
8048654: 65 a1 14 00 00 00 mov %gs:0x14,%eax
804865a: 89 45 f8 mov %eax,0xfffffff8(%ebp)
804865d: 31 c0 xor %eax,%eax
804865f: 8d 5d f4 lea 0xfffffff4(%ebp),%ebx
8048662: 89 1c 24 mov %ebx,(%esp)
8048665: e8 77 ff ff ff call 80485e1 <gets>
804866a: 89 1c 24 mov %ebx,(%esp)
804866d: e8 ca fd ff ff call 804843c <puts@plt>
8048672: 8b 45 f8 mov 0xfffffff8(%ebp),%eax
8048675: 65 33 05 14 00 00 00 xor %gs:0x14,%eax
804867c: 74 05 je 8048683 <echo+0x36>
804867e: e8 a9 fd ff ff call 8048683 <echo+0x36>
8048683: 83 c4 14 add $0x14,%esp
8048686: 5b pop %ebx
8048687: 5d pop %ebp
8048688: c3 ret
```
Setting Up Canary

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary
[3] [2] [1] [0]
Stack Frame for echo

Get canary
Put on stack
Erase canary

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

echo:
    ...
    movl %gs:20, %eax # Get canary
    movl %eax, -8(%ebp) # Put on stack
    xorl %eax, %eax # Erase canary
    ...

Checking Canary

Before call to gets

Stack Frame for main

Return Address

Saved %ebp
Saved %ebx
Canary
buf

Stack Frame for echo

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

echo: 

movl -8(%ebp), %eax # Retrieve from stack
xorl %gs:20, %eax # Compare with Canary
je .L24 # Same: skip ahead
call __stack_chk_fail # ERROR
.L24:

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

echo: 

movl -8(%ebp), %eax # Retrieve from stack
xorl %gs:20, %eax # Compare with Canary
je .L24 # Same: skip ahead
call __stack_chk_fail # ERROR
.L24:

Canary Example

**Before call to gets**

Stack Frame for **main**

- Return Address
- Saved `%ebp`
- Saved `%ebx`

```
03 e3 7d 00
[3][2][1][0]
```

Stack Frame for **echo**

```
buf
```

**Input 1234**

Stack Frame for **main**

- Return Address
- Saved `%ebp`
- Saved `%ebx`

```
03 e3 7d 00
```

```
34 33 32 31
```

Stack Frame for **echo**

```
buf
```

(gdb) `break echo`
(gdb) `run`
(gdb) `stepi 3`
(gdb) `print /x *((unsigned *) $ebp - 2)`

$1 = 0x3e37d00

Benign corruption!
(allow programmers to make silent off-by-one errors)
What are the common features of integer overflow vulnerabilities?

```c
unsigned int x = read_int();
if ( x > 0x7fffffff )
    abort();
unsigned int s = x*sizeof(int);
char* p = malloc(s);
read_int_into_buf(p, x);
```

- an untrusted source
- an incomplete check
- an integer overflow
- a heap overflow followed
- a sensitive operation
... if (version == 4) {
    const uint16_t sps = _X_BE_16 (this->header+44) ? : 1;
    this->w = _X_BE_16 (this->header+42);
    this->h = _X_BE_16 (this->header+40);
    this->cfs = _X_BE_32 (this->header+24);
    this->frame_len = this->w * this->h;
    this->frame_size = this->frame_len * sps;
    this->frame_buffer = calloc(this->frame_size, 1);
...
CVE-2008-1722 (CUPS)

Buffer Overflow

Integer and Heap Overflow

An untrusted source

An incomplete check

An integer overflow

An sensitive operation

Background

Buffer Overflow

Integer and Heap Overflow

Summary
CVE-2008-2430(VLC)

......
if( ChunkFind( p_demux, "fmt ", &i_size ) )
{
    msg_Err( p_demux, "cannot find 'fmt ' chunk" );
goto error;
}
if( i_size < sizeof( WAVEFORMATEX ) - 2 )
{
    msg_Err( p_demux, "invalid 'fmt ' chunk" );
goto error;
}
stream_Read( p_demux->s, NULL, 8 ); /* Can
/* load waveformex */
p_wf_ext = malloc( __EVEN( i_size ) + 2 );
......

an untrusted source
an incomplete check
an integer overflow
a sensitive operation
What’s the essential feature of integer overflow vulnerabilities?

unsigned int x = read_int();
if (x > 0x7fffffff)
    abort();
unsigned int s = x*sizeof(int);
char* p = malloc(s);
read_int_into_buf(p, x);
Summary

Software vulnerabilities

1. Cannot be avoided (software complexity)
2. Memory bugs (buffer overflow, integer overflow) are dangerous
3. Tons of research has been carried out to stop memory bugs
4. Stack, and integer and heap overflow can be stopped.
References

- Smashing the Stack for Fun and Profit by Aleph One
- A Comparison of Buffer Overflow Prevention Implementations and Weaknesses (Blackhat 2004)
- ...