Outline

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

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

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 Pushing

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

- **Stack "Bottom"**
  - `%esp`
  - `%eip`

- **Stack "Top"**
  - `%esp`
  - `%eip`

### IA32 Stack Popping

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

- **Stack "Bottom"**
  - `%esp`
  - `%eip`

### Procedure Call Example

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

### Procedure Return Example

```
8048591: c3 ret
```

### Background

**Buffer Overflow**

- Integer and Heap Overflow

**Summary**

- Procedure Call Example
- Procedure Return Example
### Call Chain Example

**Code Structure**
- `yoo(…)
  
  {  
    
    who();  
    
  }

- `who(…)
  
  {  
    
    amI();  
    
  }

Procedure `amI` recursive

**Call Chain**

- `amI(…)
  
  {  
    
    who();  
    
  }

- `who(…)
  
  {  
    
    amI();  
    
  }

- `amI(…)
  
  {  
    
    who();  
    
  }

**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

**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

- Frame Pointer `%ebp`
- Return Address
- Old `%ebp`
- Arguments
- Saved Registers + Local Variables
- Argument Build
- Frame Pointer `%esp`
- Stack Pointer `%esp`
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0xffffbcd0</td>
</tr>
<tr>
<td>p3</td>
<td>0x65586008</td>
</tr>
<tr>
<td>p1</td>
<td>0x55585008</td>
</tr>
<tr>
<td>p4</td>
<td>0x1904a110</td>
</tr>
<tr>
<td>p2</td>
<td>0x1904a008</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x18049760</td>
</tr>
<tr>
<td>&amp;beyond</td>
<td>0x08049744</td>
</tr>
<tr>
<td>big_array</td>
<td>0x18049780</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x08049760</td>
</tr>
<tr>
<td>main()</td>
<td>0x08048306</td>
</tr>
<tr>
<td>useless()</td>
<td>0x08049744</td>
</tr>
<tr>
<td>final malloc()</td>
<td>0x006be166</td>
</tr>
</tbody>
</table>

malloc() is dynamically linked
address determined at runtime
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!
  - 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

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;
}
```
**Vulnerable Buffer Code**

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

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

```bash
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: 89 e5 mov %esp,%ebp
80485c8: 53 push %ebx
80485cc: 8d 5d f8 lea 0xfffffff8(%ebp),%ebx
80485cf: 89 1c 24 mov %ebx,(%esp)
80485d2: e8 9e ff ff ff call 8048575 <gets>
80485d7: 89 1c 24 mov %ebx,(%esp)
80485da: e8 05 fe ff ff call 80483e4 <puts@plt>
80485df: 83 c4 14 add $0x14,%esp
80485e2: 5b pop %ebx
80485e3: 5d pop %ebp
80485e4: c3 ret
```

call echo

```
80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave
80485f1: c3 ret
```

**Buffer Overflow Stack Example**

```
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
```

Before call to gets:

```
stack frame for main:
Return Address
Saved %ebp
Saved %ebx
buf

stack frame for echo:
```

Before call to gets:

```
stack frame for main:
Return Address
Oxfffff688
Saved %ebp
Saved %ebx
buf

stack frame for echo:
```

Carnegie Mellon
Buffer Overflow Stack Example #1

Before call to `gets`

Stack Frame for `main`

Input 12345678

Stack Frame for `main`

Before call to `gets`

Stack Frame for `echo`

Overflow buf, and corrupt %ebx, but no problem

Base pointer corrupted

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Desired return point

80485f1: c3 ret # Set %ebp to corrupted value

Buffer Overflow Stack Example #2

Before call to `gets`

Stack Frame for `main`

Input 12345678

Stack Frame for `main`

Before call to `gets`

Stack Frame for `echo`

Return address corrupted

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Desired return point

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

Buffer Overflow Stack Example #3

Before call to `gets`

Stack Frame for `main`

Input 12345678!"#

Stack Frame for `main`

Stack Frame for `echo`

Stack after call to `gets()`

void foo()
{
    return address A
}

int bar()
{
    char buf[64];
    gets(buf);
    ... return ...
}

foo stack frame

data written by `gets()`

pad

exploit code

bar stack frame
### 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.

- **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

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);  /* Use fgets to read the string */
    puts(buf);  /* Or use %ns where n is a suitable integer */
}
```

- **Code Red**
  - 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

- 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
System-Level Protections

- 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

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`

Protected Buffer Disassembly

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

Setting Up Canary

- Before call to `gets`
  - Saved `%ebp`
  - Saved `%ebx`
  - Canary
  - Stack Frame for `echo`

```
push %ebp
mov %esp,%ebp
push %ebx
sub $0x14,%esp
mov %gs:0x14,%eax
lea 0xffffffff4(%ebp),%ebx
xorl %eax,%eax
...```

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

```
push %ebp
mov %esp,%ebp
push %ebx
sub $0x14,%esp
mov %gs:0x14,%eax
lea 0xffffffff4(%ebp),%ebx
xorl %eax,%eax
...```

```
push %ebp
mov %esp,%ebp
push %ebx
sub $0x14,%esp
mov %gs:0x14,%eax
lea 0xffffffff4(%ebp),%ebx
xorl %eax,%eax
...```

```
push %ebp
mov %esp,%ebp
push %ebx
sub $0x14,%esp
mov %gs:0x14,%eax
lea 0xffffffff4(%ebp),%ebx
xorl %eax,%eax
...```

Background Buffer Overflow Integer and Heap Overflow Summary

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

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

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

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

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

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

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

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

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

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

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

```
unix> ./bufdemo-protected
Type a string:1234
1234
unix> ./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```
### Background

Buffer Overflow

Integer and Heap Overflow

Summary

---

### Buffer Overflow

#### Integer and Heap Overflow

#### Summary

---

### Checking Canary

**Before call to gets**

Stack Frame for `main`

Return Address

Saved `%ebp`

Saved `%ebx`

Canary

[3] [2] [1] [0]

`buf`

**Before call to gets**

Stack Frame for `echo`

Return Address

Saved `%ebp`

Saved `%ebx`

Canary

[3] [2] [1] [0]

'buf'

---

### Outline

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

---

### What are the common features of integer overflow vulnerabilities?

- An untrusted source
- An incomplete check
- An integer overflow
- A sensitive operation
- A heap overflow followed

---

### Input 1234

Before call to gets

Stack Frame for `main`

Return Address

Saved `%ebp`

Saved `%ebx`

buf

canary

Stack Frame for `echo`

Return Address

Saved `%ebp`

Saved `%ebx`

buf

---

### Canary Example

#### Carnegie Mellon

50

Canary Example

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

$1 = 0x3e37d00

Benign corruption!

(allows programmers to make silent off-by-one errors)
### CVE-2008-5238 (Xine)

```c
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);
    ....
```

#### Background

- **An untrusted source**
- **A sensitive operation**
- **An incomplete check**

### CVE-2008-1722 (CUPS)

```c
png_get_IHDR(pp, info, &width, &height, &bit_depth, &color_type, &interlace_type, img->ysize * 3);
```

#### Background

- **An untrusted source**
- **An incomplete check**
- **An integer overflow**
- **A sensitive operation**

### CVE-2008-2430 (VLC)

```c
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 read */
/* load waveformatex */
p_wf_ext = malloc (X_BE_32 (i_size ) + 2);
```

#### Background

- **An untrusted source**
- **An incomplete check**
- **An integer overflow**
- **A sensitive operation**
**Software vulnerabilities**

- Cannot be avoided (software complexity)
- Memory bugs (buffer overflow, integer overflow) are dangerous
- Tons of research has been carried out to stop memory bugs
- 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)
- ...