Machine-Level Programming V: Miscellaneous Topics
Sept. 24, 2002

Topics
- Linux Memory Layout
- Buffer Overflows
Administrivia

- **Lab 2 due tonight by midnight!**
  - If you checked out a bomb with your “real name”, then you don’t need to hand in anything else.
  - If you didn’t use your real name, then you need to let Jim know what your bomb number is.

- **Lab 3 info will be up after class**
  - Can work as a team of no more than 2 people!
  - You can work alone if you like.
  - Start early – especially if you had difficulties with Lab 2.
**Linux Memory Layout**

- **Stack**
  - Runtime stack (8MB limit)

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

- **DLLs**
  - Dynamically Linked Libraries
  - Library routines (e.g., printf, malloc)
  - Linked into object code when first executed

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

- **Text**
  - Executable machine instructions
  - Read-only

**Upper 2 hex digits of address**

Red Hat v. 6.2 ~1920MB memory limit
Linux Memory Allocation

Initially

Linked

Some Heap

More Heap

Initially

Linked

Some Heap

More Heap

Initially

Linked

Some Heap

More Heap
Text & Stack Example

(gdb) break main
(gdb) run
    Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
    $3 = (void *) 0xbffffffc78

Main

- Address 0x804856f should be read
  0x0804856f

Stack

- Address 0xbffffffc78
Dynamic Linking Example

Initially
- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x08048454

Final
- Code in DLL region
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 ... */
}
```
## Example Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$esp</td>
<td>0xbfffff7c8</td>
</tr>
<tr>
<td>p3</td>
<td>0x500b5008</td>
</tr>
<tr>
<td>p1</td>
<td>0x400b4008</td>
</tr>
<tr>
<td>Final malloc</td>
<td>0x40006240</td>
</tr>
<tr>
<td>p4</td>
<td>0x1904a640</td>
</tr>
<tr>
<td>p2</td>
<td>0x1904a538</td>
</tr>
<tr>
<td>beyond</td>
<td>0x1904a524</td>
</tr>
<tr>
<td>big_array</td>
<td>0x1804a520</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0804a510</td>
</tr>
<tr>
<td>main()</td>
<td>0x0804856f</td>
</tr>
<tr>
<td>useless()</td>
<td>0x08048560</td>
</tr>
<tr>
<td>Initial malloc</td>
<td>0x08048454</td>
</tr>
</tbody>
</table>

![Diagram](image)
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
Internet Worm and IM War (cont.)

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

- Implementation of Unix function `gets`
  - No way to specify limit on number of characters to read

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

- Similar problems with other Unix functions
  - `strcpy`: Copies string 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);
}

int main()
{
    printf("Type a string:");
    echo();
    return 0;
}
```
Buffer Overflow Executions

unix>./bufdemo
Type a string: 123
123

unix>./bufdemo
Type a string: 12345
Segmentation Fault

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

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

```
echo:
    pushl %ebp  # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp  # Allocate space on stack
    pushl %ebx  # Save %ebx
    addl $-12,%esp  # Allocate space on stack
    leal -4(%ebp),%ebx  # Compute buf as %ebp-4
    pushl %ebx
    call gets
    . . .
```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d

8048648: call 804857c <echo>
804864d: mov 0xfffffffff8e8(%ebp),%ebx # Return Point
Buffer Overflow Example #1

Before Call to `gets`

Input = “123”

Stack Frame for `main`

Return Address
Saved `%ebp`
[3][2][1][0]

Stack Frame for `echo`

Stack Frame for `main`

Return Address
Saved `%ebp`
[3][2][1][0]

Stack Frame for `echo`

No Problem
Buffer Overflow Stack Example #2

Input = “12345”

Stack Frame for main

Stack Frame for echo

Saved value of %ebp set to 0xbfff0035

Bad news when later attempt to restore %ebp

8048592: push %ebx
8048593: call 80483e4 <_init+0x50> # gets
8048598: mov 0xffffffe8(%ebp),%ebx
804859b: mov %ebp,%esp
804859d: pop %ebp # %ebp gets set to invalid value
804859e: ret
Buffer Overflow Stack Example #3

Input = “12345678”

Stack Frame for main

Return Address
Saved %ebp
[3][2][1][0]

Stack Frame for main

Stack Frame for echo

%ebp and return address corrupted

Invalid address

No longer pointing to desired return point

8048648: call 804857c <echo>
804864d: mov 0xfffffffffe8(%ebp),%ebx # Return Point
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `bar()` executes `ret`, will jump to exploit code
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:
  - `finger droh@cs.cmu.edu`

- 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 Worm

## History

- **June 18, 2001.** Microsoft announces buffer overflow vulnerability in IIS Internet server
- **July 19, 2001.** Over 250,000 machines infected by new virus in 9 hours
- **White house must change its IP address.** Pentagon shut down public WWW servers for day

## When We Set Up CS:APP Web Site

- Received strings of form
  
  ```
  GET /default.ida?
  NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
  %u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858
  %ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff
  %u0078%u0000%u00=a
  HTTP/1.0" 400 325 "-" "-
  ```
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
Code Red Effects

Later Version Even More Malicious
- Code Red II
- As of April, 2002, over 18,000 machines infected
- Still spreading

Paved Way for NIMDA
- Variety of propagation methods
- One was to exploit vulnerabilities left behind by Code Red II
Avoiding Overflow Vulnerability

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    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
Static Code Analysis

http://spinroot.com/static/
MOSDEF Demo

http://www.immunitysec.com/resources-freesoftware.shtml

mosdef.py -f <filename>

Outputs byte codes and length