Topics
- Linux Memory Layout
- Understanding Pointers
- Buffer Overflow
- Floating Point Code

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

Linux Memory Allocation

Text & Stack Example
(gdb) break main
(gdb) run
Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
$3 = (void *) 0xbfffc78

Main
- Address 0x804856f should be read 0x0804856f

Stack
- Address 0xbfffc78
### Dynamic Linking Example

```c
(gdb) print malloc
$1 = {<text variable, no debug info>}
0x8048454 <malloc>
(gdb) run
Program exited normally.
(gdb) print malloc
$2 = {void *(unsigned int)}
0x40006240 <malloc>
```

#### Initially
- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x0

#### 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

```plaintext
$esp 0xbfffffc78
p3 0x500b5008
p1 0x400b4008
Final malloc 0x40006240
p4 0x1904a640
p2 0x1904a538
beyond 0x1904a524
big_array 0x1804a520
huge_array 0x0804a510
main() 0x0804856f
useless() 0x08048560
Initial malloc 0x08048454
```

---

### C operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- + - * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>/ %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; ^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>= += -= *= /= %= &amp;= ^= != &lt;&lt; &gt;&gt;</td>
<td>right to left</td>
</tr>
</tbody>
</table>

Note: Unary +, -, and * have higher precedence than binary forms
**C pointer declarations**

- `int *p`  
  p is a pointer to int

- `int *p[13]`  
  p is an array[13] of pointer to int

- `int *((p[13])`  
  p is an array[13] of pointer to int

- `int **p`  
  p is a pointer to a pointer to an int

- `int (*(p)[13])`  
  p is a pointer to an array[13] of int

- `int *f()`  
  f is a function returning a pointer to int

- `int (*)(f)()`  
  f is a pointer to a function returning int

- `int *((*f())[13])()`  
  f is a function returning ptr to an array[13] of pointers to functions returning int

- `int *((*x[3])())[5]`  
  x is an array[3] of pointers to functions returning pointers to array[5] of ints

**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

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';
  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
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

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

Buffer Overflow Stack Example

```
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
    # Push buf on stack
    call gets
    # Call gets
...```
Buffer Overflow Example #1

Before Call to `gets`

Input = “123”

No Problem

Buffer Overflow Stack Example #2

Input = “12345”

Bad news when later attempt to restore `%ebp`

Buffer Overflow Stack Example #3

Input = “123456 78”

Invalid address

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.`

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?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u8190%u00c3%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0" 400 325 "=" "="
  ```

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year. ...

It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.

... Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL’s friendly exterior they are nefariously compromising peoples’ security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

It was later determined that this email originated from within Microsoft!
**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**

```c
/* 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

---

**IA32 Floating Point**

**History**

- 8086: first computer to implement IEEE FP
  - separate 8087 FPU (floating point unit)
- 486: merged FPU and Integer Unit onto one chip

**Summary**

- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

**Floating Point Formats**

- single precision (C float): 32 bits
- double precision (C double): 64 bits
- extended precision (C long double): 80 bits
FPU Data Register Stack

FPU register format (extended precision)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>78</td>
<td>6463</td>
</tr>
</tbody>
</table>

FPU registers
- 8 registers
- Logically forms shallow stack
- Top called %st(0)
- When push too many, bottom values disappear

"Top" \[ \text{stack grows down} \]

FPU instructions

Large number of floating point instructions and formats
- \(~50\) basic instruction types
- \(\text{load, store, add, multiply}\)
- \(\text{sin, cos, tan, arctan, and log!}\)

Sample instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fldz</td>
<td>push 0.0</td>
<td>Load zero</td>
</tr>
<tr>
<td>flds Addr</td>
<td>push M[Addr]</td>
<td>Load single precision real</td>
</tr>
<tr>
<td>fmuls Addr</td>
<td>(%st(0) \leftarrow %st(0) \times M[\text{Addr}])</td>
<td>Multiply</td>
</tr>
<tr>
<td>faddp</td>
<td>(%st(1) \leftarrow %st(0) + %st(1)); pop</td>
<td>Add and pop</td>
</tr>
</tbody>
</table>

Floating Point Code Example

Compute Inner Product of Two Vectors
- Single precision arithmetic
- Common computation

Compute Inner Product Stack Trace

1. \(\text{fldz}\)

2. \(\text{flds }\%\text{ebx}\)

3. \(\text{fmuls }\%\text{ecx}\)

4. \(\text{faddp}\)

5. \(\text{flds }\%\text{ebx}\)

6. \(\text{fmuls }\%\text{ecx}\)

7. \(\text{faddp}\)

```
float ipf (float x[], float y[], int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

```
pushl %ebp  # setup
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%ebx  # %ebx=&x
movl 12(%ebp),%ecx  # %ecx=&y
movl 16(%ebp),%edx  # %edx=n
flds
xorl %eax,%eax  # i=0
cmpl %edx,%eax  # if i<n done
jge .L3
.L5:
    flds (%ebx,%eax,4)  # push x[i]
    fmuls (%ecx,%eax,4)  # st(0)=y[i]
    faddp  # st(1)=st(0); pop
    incl %eax  # i++
    cmpl %edx,%eax  # if i<n repeat
    jle .L5
.L3:
    movl -4(%ebp),%ebx  # finish
    movl %ebp, %esp
    ret  # st(0) = result
```
Final Observations

Memory Layout
- OS/machine dependent (including kernel version)
- Basic partitioning: stack/data/text/heap/DLL found in most machines

Type Declarations in C
- Notation obscure, but very systematic

Working with Strange Code
- Important to analyze nonstandard cases
  - E.g., what happens when stack corrupted due to buffer overflow
- Helps to step through with GDB

IA32 Floating Point
- Strange “shallow stack” architecture