Machine-Level Programming V: Wrap-up

Dr. Steve Goddard
goddard@cse.unl.edu

Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
I have modified them and added new slides.

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

Initially

Linked

Some Heap

More Heap

Text & Stack Example

Initially

(qdb) break main
(qdb) run
Breakpoint 1, 0x804856f in main ()
(qdb) print $esp
$3 = (void *) 0xbfffc78

Main
- Address 0x804856f should be read 0x804856f

Stack
- Address 0xbfffc78

Giving credit where credit is due

Mention credit where credit is due

Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
I have modified them and added new slides.

Topics
- Linux Memory Layout
- Understanding Pointers
- Buffer Overflow
- Floating Point Code
**Dynamic Linking Example**

(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
  0x8048454

Final
- Code in DLL region

**Memory Allocation Example**

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**

$esp 0xbbfffc78
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>&lt; &gt; &lt;= &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; ^</td>
<td></td>
</tr>
<tr>
<td>? : = + = - = * = / = % = &amp; =</td>
<td>right to left</td>
</tr>
<tr>
<td>^ = = + = - = * = / = % = &amp; =</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</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 *[13]` p is an array[13] of pointer to int
- `int **p` p is a pointer to a pointer to 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 function returning int
- `int *(*f()())[13]` f is a function returning ptr to an array[13] of pointers to functions returning int

**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

![Diagram of MSN server and AIM server connections]
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') {
        c = getc();
    }
    *p = '\'0';
    return dest;
}
```

- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given % conversion specification

**Vulnerable Buffer Code**

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

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

**Buffer Overflow Executions**

```c
unix> /buffdemo
Type a string:123
123
unix> /buffdemo
Type a string:12345
Segmentation Fault
unix> /buffdemo
Type a string:1234575
Segmentation Fault
```

**Buffer Overflow Stack Example**

```c
unix> gdb /buffdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
gdb> print/x ((unsigned *)%ebp + 1)
$1 = 0x80485e48
(gdb) print/x * ((unsigned *)%ebp + 1)
$2 = 0x80485e48
```

Before call to `gets`

```c
(gdb) print/x * %ebp
$3 = 0x8048583
```

Before call to `echo`

```c
(gdb) print/x %ebp
$4 = 0x8048589
```

**Buffer Overflow Stack**

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

```c
int main()
{
    printf("Type a string:");
    echo();
    return 0;
}
```
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 -s root@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.

Exploits Based on Buffer Overflows

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

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
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 overflow bug. By itself, this might not be the end of the world, as we surely have had ours. But AOL is now "exploiting these new buffer overflow bugs" to help in its efforts to block other instant messaging clients.

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 people's 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

---

**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

---

**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

---

**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>79 78</th>
<th>d43</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>frac</td>
<td></td>
</tr>
</tbody>
</table>

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

`Top` ➔ stack grows down

FPU instructions

Large number of floating point instructions and formats

- ~50 basic instruction types
- load, store, add, multiply
- 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>push 0.0</td>
<td></td>
<td>Load zero</td>
</tr>
<tr>
<td>push M[Add]</td>
<td></td>
<td>Load single precision real</td>
</tr>
<tr>
<td>\texttt{fmul} \texttt{Addr}</td>
<td>\texttt{st}(0) \texttt{=} \texttt{st}(0) \texttt{*} \texttt{M[Add]}</td>
<td>Multiply</td>
</tr>
<tr>
<td>\texttt{faddp}</td>
<td>\texttt{st}(1) \texttt{=} \texttt{st}(0) \texttt{*} \texttt{st}(1)</td>
<td>Add and pop</td>
</tr>
</tbody>
</table>

Floating Point Code Example

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

```c
float ipdf(float a[], float b[], int n);
{ int i; float result = 0.0;
  for (i = 0; i < n; i++) {
    result += a[i] * b[i];
  }
  return result;
}
```

Inner Product Stack Trace

Initialization

1. \texttt{fids}

| 0.0 | \texttt{st}(0) |

Iteration 0

2. \texttt{fids} \texttt{tobe, beax, 4}

| 0.0 | \texttt{tob}(0) | \texttt{st}(1) |

3. \texttt{fmls} \texttt{tobe, beax, 4}

| 0.0 | \texttt{tob}(0) | \texttt{tob}(1) |

4. \texttt{faddp} \texttt{0.0} \texttt{tob(tob(1)\*tob(0))} | \texttt{st}(0) |

5. \texttt{fids} \texttt{tobe, beax, 4}

| \texttt{tob}(0) | \texttt{tob}(1) |

6. \texttt{fmls} \texttt{tobe, beax, 4}

| \texttt{tob}(0) | \texttt{tob}(1) |

7. \texttt{faddp} \texttt{0.0} \texttt{tob(tob(1)\*tob(0))} | \texttt{tob}(0) |

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