Machine-Level Programming V: Memory Layout and Buffer Overflows

CSci 2021: Machine Architecture and Organization
Lecture #14, February 20th, 2015
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron, Antonia Zhai

Today
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

IA32 Linux Memory Layout

- Stack
  - Runtime stack (Ubuntu sets 8MB soft 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

Upper 2 hex digits
= 8 bits of address

IA32 Example Addresses

address range ~2^32

FF

Stack

BMB

80

00

Heap

Data

Text

FF

Stack

BMB

80

00

Heap

Data

Text

Reserved for
OS kernel

Reserved
for
OS kernel

Stack

B7

C0

BF

Shared library

Other things
placed between
stack and heap start

Reserved
for
OS kernel

Stack

B7

C0

BF

Other heap obj.

Heap

Data

Text

malloc() is dynamically linked
address determined at runtime

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 ... */
}

Where does everything go?

IA32 Layout Variations

32-bit: 3GB/1GB split common with

Reserved
for
OS kernel

Stack

B7

C0

BF

Shared library

Other things
placed between
stack and heap start

Reserved
for
OS kernel

Stack

B7

C0

BF

Other heap obj.

Heap

Data

Text
x86-64 Example Addresses

address range “2^8”, rest reserved

$rsp 0x00007ffffff8d1f8
p3 0xe0002dabead0d010
p1 0xe0002dadaed0d010
p4 0xe0000000011501120
p2 0xe0000000011501010
sp2 0xe0000000010500a60
beyond 0xe0000000000500a44
big_array 0xe0000000010500a80
huge_array 0xe0000000000500a50
main() 0xe0000000000400510
useless() 0xe0000000000400500

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?

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.
- What was the final round in the war?
- 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.

Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

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 %a conversion specification
Vulnerable Buffer Code

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

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

Buffer Overflow Example #1

```
Before call to gets
```
**Buffer Overflow Example #3**

Before call to `gets`

<table>
<thead>
<tr>
<th>Stack Frame for <code>main</code></th>
<th>Stack Frame for <code>echo</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 00</td>
<td>8048678</td>
</tr>
<tr>
<td>FF FF 06 00</td>
<td>FF FF FF FF FF FF</td>
</tr>
<tr>
<td>Stack off <code>buf</code></td>
<td>Stack off <code>buf</code></td>
</tr>
<tr>
<td>FF FF FF FF FF FF FF</td>
<td>FF FF FF FF FF FF</td>
</tr>
</tbody>
</table>

Input `123456789ABC`

<table>
<thead>
<tr>
<th>Stack Frame for <code>echo</code></th>
<th>Stack Frame for <code>main</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 00</td>
<td>8048678</td>
</tr>
<tr>
<td>43 42 41 39</td>
<td>38 37 36 35</td>
</tr>
<tr>
<td>34 33 32 31</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Return address corrupted

**Malicious Use of Buffer Overflow**

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

Stack after call to `gets ()`

```
foo stack frame
```

```
bar stack frame
```

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

---

**Discussion Break: Unknown Addresses?**

- Basic attack requires attacker to know address B of buffer
- Is an attack still possible if B is variable?
  - E.g. what if attacker only knows B +/- 30?

- Some possible attack strategies:
  - Try attack repeatedly
  - "NOP sled": (0x90 is one-byte no-operation in IA32)

**Exploits Based on Buffer Overflow**

- **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 finger 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.
Avoiding Overflow Vulnerability

Use library routines that limit string lengths
- `fgets` instead of `gets`
- `strncpy` instead of `strcpy`
- Don’t use `scanf` with plain `%s` conversion specification
  - Use `fgets` to read the string
  - Or use `%n%s` 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
  - Modern version: address space layout randomization "ASLR"

- Nonexecutable data segments
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
  - Can execute anything readable
  - More recent processors added explicit way to disable "execute" permission, e.g. for stack

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`

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

```
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
```

Protected Buffer Disassembly

```
echo:
804864d:  55      push   %ebp
804864e:  89 e5    mov    %esp,%ebp
8048650:  53      push   %ebx
8048651:  83 ec      sub    $0x14,%esp
8048654:  65 a1 14 00 00 00 mov    %gs:0x14,%eax
804865a:  89 45 f8    mov 0xfffffff8(%ebp),%eax
804865d:  31 c0    xor    %eax,%eax
804865f:  8d 5d f4    lea    0xfffffff4(%ebp),%ebx
8048662:  89 1c 24    mov    %ebx,(%esp)
8048665:  e8 a9 ff ff ff call    80485e1 <gets>
804866a:  89 1c 24    mov    %ebx,(%esp)
804866d:  e8 ca ff ff ff call    804842c <FAIL>
8048672:  8b 45 f8    mov    0xfffffff8(%ebp),%eax
8048675:  83 c4 14    add    $0x14,%esp
8048678:  5b      pop    %ebx
8048679:  5d      pop    %ebp
804867a:  c3      ret
```

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

Setting Up Canary

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

```
unix> gdb bufdemo-protected
type a string: 1234
1234
unix> gdb bufdemo-protected
type a string: 12345
*** stack smashing detected ***
```

Checking Canary

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

```
unix> gdb bufdemo-protected
(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
```

```
echo:
804864d:  55      push   %ebp
804864e:  89 e5    mov    %esp,%ebp
8048650:  53      push   %ebx
8048651:  83 ec      sub    $0x14,%esp
8048654:  65 a1 14 00 00 00 mov    %gs:0x14,%eax
804865a:  89 45 f8    mov 0xfffffff8(%ebp),%eax
804865d:  31 c0    xor    %eax,%eax
804865f:  8d 5d f4    lea    0xfffffff4(%ebp),%ebx
8048662:  89 1c 24    mov    %ebx,(%esp)
8048665:  e8 a9 ff ff ff call    80485e1 <gets>
804866a:  89 1c 24    mov    %ebx,(%esp)
804866d:  e8 ca ff ff ff call    804842c <FAIL>
8048672:  8b 45 f8    mov    0xfffffff8(%ebp),%eax
8048675:  83 c4 14    add    $0x14,%esp
8048678:  5b      pop    %ebx
8048679:  5d      pop    %ebp
804867a:  c3      ret
```
Canary Example

Before call to gets

Input 12345

(The canary always ends with a zero byte. Why is this a good idea?)

Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection