Machine-Level Programming V: Advanced Topics

Introduction to Computer Systems
11th Lecture, Oct. 13, 2015

Instructor:
Younghoon Kim
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E.g., local variables

- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`

- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants

- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only

Hex Address

00000000  00007FFFFFFFFFFFFF
Memory Allocation Example

```c
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```
x86-64 Example Addresses

address range ~$2^{47}$

local
p1
p3
p4
p2
big_array
huge_array
main()
useless()

0x00007ffe4d3be87c
0x00007f7262a1e010
0x00007f7162a1d010
0x000000008359d120
0x000000008359d010
0x0000000080601060
0x0000000000601060
0x000000000040060c
0x0000000000400590

not drawn to scale
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
Recall: Memory Referencing Bug Example

typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}

fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault

- Result is system specific
Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;
```

<table>
<thead>
<tr>
<th>fun(i)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(0)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(1)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(2)</td>
<td>3.1399998664856</td>
</tr>
<tr>
<td>fun(3)</td>
<td>2.00000061035156</td>
</tr>
<tr>
<td>fun(4)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(6)</td>
<td>Segmentation fault</td>
</tr>
</tbody>
</table>

**Explanation:**

The critical state is shown below:

```
struct_t
```

<table>
<thead>
<tr>
<th>Location accessed by fun(i)</th>
<th>Critical State</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>?</td>
<td>4</td>
</tr>
<tr>
<td>?</td>
<td>5</td>
</tr>
<tr>
<td>?</td>
<td>6</td>
</tr>
</tbody>
</table>
Such problems are a BIG deal

- Generally called a “buffer overflow”
  - when exceeding the memory size allocated for an array

- Why a big deal?
  - It’s the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance

- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing
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 `%s` conversion specification
Vulnerable Buffer Code

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

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

```bash
unix> ./bufdemo
Type a string: 012345678901234567890123 012345678901234567890123
unix> ./bufdemo
Type a string: 0123456789012345678901234
Segmentation Fault
```

 btw, how big is big enough?
Buffer Overflow Disassembly

echo:

```
0000000000004006cf <echo>:
  4006cf: 48 83 ec 18            sub $0x18,%rsp
  4006d3: 48 89 e7              mov %rsp,%rdi
  4006d6: e8 a5 ff ff ff         callq 400680 <gets>
  4006db: 48 89 e7              mov %rsp,%rdi
  4006de: e8 3d fe ff ff         callq 400520 <puts@plt>
  4006e3: 48 83 c4 18            add $0x18,%rsp
  4006e7: c3                      retq
```

call_echo:

```
4006e8: 48 83 ec 08            sub $0x8,%rsp
  4006ec: b8 00 00 00 00 00     mov $0x0,%eax
  4006f1: e8 d9 ff ff ff         callq 4006cf <echo>
  4006f6: 48 83 c4 08            add $0x8,%rsp
  4006fa: c3                      retq
```
Buffer Overflow Stack

Before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

20 bytes unused

buf ← %rsp

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

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
Buffer Overflow Stack Example

Before call to gets

Stack Frame for call_echo

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

call_echo:

buf ← %rsp

echo:

subq $24, %rsp
movq %rsp, %rdi
call gets

. . .

. . .

20 bytes unused

[3] [2] [1] [0]
Buffer Overflow Stack Example #1

After call to gets

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...
    4006f1: callq 4006cf <echo>
    4006f6: add $0x8,%rsp
    ...

buf ← %rsp

unix>./bufdemo
Type a string:01234567890123456789012
01234567890123456789012

Overflowed buffer, but did not corrupt state
Buffer Overflow Stack Example #2

After call to gets

Stack Frame for call_echo

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00 40 00 34</td>
<td></td>
</tr>
<tr>
<td>33 32 31 30</td>
<td></td>
</tr>
<tr>
<td>39 38 37 36</td>
<td></td>
</tr>
<tr>
<td>35 34 33 32</td>
<td></td>
</tr>
<tr>
<td>31 30 39 38</td>
<td></td>
</tr>
<tr>
<td>37 36 35 34</td>
<td></td>
</tr>
<tr>
<td>33 32 31 30</td>
<td></td>
</tr>
</tbody>
</table>

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...
    4006f1: callq 4006cf <echo>
    4006f6: add $0x8,%rsp
    ...

buf ← %rsp

unix>./bufdemo
Type a string:0123456789012345678901234
Segmentation Fault

Overflowed buffer and corrupted return pointer
Buffer Overflow Stack Example #3

After call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 00</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
<tr>
<td>39 38 37 36</td>
</tr>
<tr>
<td>35 34 33 32</td>
</tr>
<tr>
<td>31 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

void echo()
{
    char buf[4];
    gets(buf);
    . . .
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    . . .

call_echo:
    . . .
    4006f1: callq 4006cf <echo>
    4006f6: add $0x8, %rsp
    . . .

buf ← %rsp

unix>./bufdemo
Type a string: 012345678901234567890123
012345678901234567890123

Overflowed buffer, corrupted return pointer, but program seems to work!
**Buffer Overflow Stack Example #3 Explained**

*After call to gets*

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 00</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
<tr>
<td>39 38 37 36</td>
</tr>
<tr>
<td>35 34 33 32</td>
</tr>
<tr>
<td>31 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

**register_tm_clones:**

```
400600:    mov   %rsp,%rbp
400603:    mov   %rax,%rdx
400606:    shr   $0x3f,%rdx
40060a:    add   %rdx,%rax
40060d:    sar   %rax
400610:    jne   400614  
400612:    pop   %rbp
400613:    retq
```

`buf ← %rsp`

“Returns” to unrelated code
Lots of things happen, without modifying critical state
Eventually executes retq back to main
Exploits Based on Buffer Overflows

- **Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines**

- Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult

- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more
Example: the original Internet worm (1988)

- Exploited a few vulnerabilities to spread
  - 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.

- Once on a machine, scanned for other machines to attack
  - invaded ~6000 computers in hours (10% of the Internet 😊)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted...
  - and CERT was formed... still homed at CMU (http://www.cert.org/)
Example 2: IM War

- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers

https://nplusonemag.com/issue-19/essays/chat-wars/
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 really happening?
  - AOL had discovered a buffer overflow bug in their own AIM clients
  - They exploited it to detect and block Microsoft: the exploit code returned a 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
Aside: Worms and Viruses

- **Worm: A program that**
  - Can run by itself
  - Can propagate a fully working version of itself to other computers

- **Virus: Code that**
  - Adds itself to other programs
  - Does not run independently

- Both are (usually) designed to spread among computers and to wreak havoc
OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
1. Avoid Overflow Vulnerabilities in Code (!)

For example, 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

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
2. System-Level Protections can help

- **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
  - E.g.: 5 executions of memory allocation code
    ```
    local 0x7ffe4d3be87c 0x7ff75a4f9fc 0x7feadb7c80c 0x7feaea2fda 0x7ffcd452017c
    ```
  - Stack repositioned each time program executes

- **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 marked as non-executable
3. Stack Canaries can help

**Idea**
- Place special value (“canary”) on stack just beyond buffer
- Check for corruption before exiting function

**GCC Implementation**
- `-fstack-protector`
- Now the default (disabled earlier)

```
unix>./bufdemo-protected
Type a string: 0123456
0123456

unix>./bufdemo-protected
Type a string: 01234567
*** stack smashing detected ***
```
Protected Buffer Disassembly

echo:

```
40072f:   sub   $0x18,%rsp
400733:   mov   %fs:0x28,%rax
40073c:   mov   %rax,0x8(%rsp)
400741:   xor   %eax,%eax
400743:   mov   %rsp,%rdi
400746:   callq 4006e0 <gets>
40074b:   mov   %rsp,%rdi
40074e:   callq 400570 <puts@plt>
400753:   mov   0x8(%rsp),%rax
400758:   xor   %fs:0x28,%rax
400761:   je    400768 <echo+0x39>
400768:   add   $0x18,%rsp
40076c:   retq
```
Setting Up Canary

*Before call to gets*

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

Stack Frame for `call_echo`

- Return Address (8 bytes)
- Canary (8 bytes)

Before call to `gets`

```
buf ← %rsp
```

```
echo:
    . . .
    movq %fs:40, %rax    # Get canary
    movq %rax, 8(%rsp)   # Place on stack
    xorl %eax, %eax      # Erase canary
    . . .
```
Checking Canary

After call to gets

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

Input: 0123456

buf ← %rsp

echo:
    . . .
    movq 8(%rsp), %rax   # Retrieve from stack
    xorq %fs:40, %rax    # Compare to canary
    je .L6               # If same, OK
    call __stack_chk_fail # FAIL
Summary of Compound Types in C

- **Arrays**
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - Pointer to first element
  - No bounds checking

- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment