Machine-Level Programming V: Advanced Topics

CENG331 - Computer Organization
Fall 2019

Instructor:
Murat Manguoglu (Sections 1-2)

Adapted from slides of the textbook: http://csapp.cs.cmu.edu/
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

```
0000000000000000
```

```
00007FFFFFFF000000
```

```
00007FFFFFFF000000
```

```
00007FFFFFFFFFFF
```

not drawn to scale

8MB
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 ... */
}
```

Where does everything go?
x86-64 Example Addresses

address range $\sim 2^{47}$

- local
- p1
- p3
- p4
- p2
- big_array
- huge_array
- main()
- useless()
Runaway Stack Example

Functions store local data on in stack frame

Recursive functions cause deep nesting of frames

```c
int recurse(int x) {
    int a[1<<15]; // 4*2^15 = 128 KiB
    printf("x = %d.  a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0)
        return -1;
    return recurse(a[a[0]]) - 1;
}
```

./runaway 67
x = 67.  a at 0x7fffd18aba930
x = 66.  a at 0x7fffd18a9a920
x = 65.  a at 0x7fffd18a7a910
x = 64.  a at 0x7fffd18a5a900
... 
x = 4.  a at 0x7fffd182da540
x = 3.  a at 0x7fffd182ba530
x = 2.  a at 0x7fffd1829a520
Segmentation fault (core dumped)
Today

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

```c
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)   | 3.1400000000 |
| fun(1)   | 3.1400000000 |
| fun(2)   | 3.1399998665 |
| fun(3)   | 2.0000006104 |
| fun(6)   | Stack smashing detected |
| fun(8)   | Segmentation fault |

- Result is system specific
Memory Referencing Bug Example

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

fun(0) -> 3.1400000000
fun(1) -> 3.1400000000
fun(2) -> 3.1399998665
fun(3) -> 2.0000006104
fun(4) -> Segmentation fault
fun(8) -> 3.1400000000

Explanation:

<table>
<thead>
<tr>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 . . . d4</td>
</tr>
<tr>
<td>d3 . . . d0</td>
</tr>
<tr>
<td>a[1]</td>
</tr>
<tr>
<td>a[0]</td>
</tr>
</tbody>
</table>

struct_t

List of Critical States:

8. ???
7. Critical State
6. Critical State
5. Critical State
4. Critical State
3. d7 . . . d4
2. d3 . . . d0
1. a[1]
0. a[0]
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();
}
```

Unix>
./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012

Unix>
./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Segmentation Fault

btw, how big is big enough?
Buffer Overflow Disassembly

echo:

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>4006cf</td>
<td>48 83 ec 18</td>
<td>sub $0x18,%rsp</td>
</tr>
<tr>
<td>4006d3</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006d6</td>
<td>e8 a5 ff ff ff</td>
<td>callq 400680 &lt;gets&gt;</td>
</tr>
<tr>
<td>4006db</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006de</td>
<td>e8 3d fe ff ff</td>
<td>callq 400520 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>4006e3</td>
<td>48 83 c4 18</td>
<td>add $0x18,%rsp</td>
</tr>
<tr>
<td>4006e7</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>

call_echo:

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>4006e8</td>
<td>48 83 ec 08</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>4006ec</td>
<td>b8 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>4006f1</td>
<td>e8 d9 ff ff ff</td>
<td>callq 4006cf &lt;echo&gt;</td>
</tr>
<tr>
<td>4006f6</td>
<td>48 83 c4 08</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>4006fa</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

Before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

20 bytes unused

buf ← %rsp

/* Echo Line */
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);
    ... 
}
```

echo:

```
    subq $x18, %rsp
    movq %rsp, %rdi
    call gets
    ... 
```

call_echo:

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

buf ← %rsp

20 bytes unused

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

After call to `gets`

### Stack Frame for `call_echo`

- `buf`: 00 00 00 00
- `buf`: 00 40 06 f6
- `buf`: 00 32 31 30
- `buf`: 39 38 37 36
- `buf`: 35 34 33 32
- `buf`: 31 30 39 38
- `buf`: 37 36 35 34
- `buf`: 33 32 31 30

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

### `echo`

```c
subq $0x18, %rsp
movq %rsp, %rdi
call gets
...
```

### `call_echo`

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

- `buf ← %rsp`

- `unix> ./bufdemo-nsp`
- `Type a string: 01234567890123456789012 01234567890123456789012`
- `"01234567890123456789012\0"`

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

After call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
<th>void echo()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
<td>{</td>
</tr>
<tr>
<td>00 40 06 00</td>
<td>char buf[4];</td>
</tr>
<tr>
<td>33 32 31 30</td>
<td>gets(buf);</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>

After call to gets:

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

Address Stack Frame for call_echo

buf ← %rsp

Program “returned” to 0x0400600, and then crashed.
Stack Smashing Attacks

- Overwrite normal return address A with address of some other code S
- When Q executes `ret`, will jump to other code

```
void P() {
    Q();
    ...
}
```

```
int Q() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```

```
void S() {
    /* Something unexpected */
    ...
}
```
Crafting Smashing String

Stack Frame for call echo

| 00 | 00 | 00 | 00 |
| 00 | 48 | 83 | 80 |
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | fb |

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

Target Code

void smash() {
    printf("I've been smashed!\n");
    exit(0);
}

Attack String (Hex)

00000000004006fb <smash>:
4006fb: 48 83 ec 08
Smashing String Effect

Stack Frame for call echo

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>48</td>
<td>83</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>fb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Target Code

```c
void smash() {
    printf("I've been smashed!\n");
    exit(0);
}
```

Attack String (Hex)

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fb</td>
<td>06</td>
<td>40</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

000000000004006fb <smash>:
4006fb: 48 83 ec 08
Code Injection Attacks

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code
How Does The Attack Code Execute?

```c
void P() {
    Q();
    ...
}

int Q() {
    char buf[64];
    gets(buf); // A->B
    ...
    return ...;
}
```
What To Do About Buffer Overflow Attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

- Lets talk about each...
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

```c
/* 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
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code
    - Stack repositioned each time program executes

![Diagram showing stack base, randomized allocation, main application code, pad, and exploit code]
2. System-Level Protections can help

- 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

Any attempt to execute this code will fail
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-sp
Type a string: 0123456
0123456

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

```
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>
400746:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

echo:
**Setting Up Canary**

*Before call to gets*

<table>
<thead>
<tr>
<th>Stack Frame for <code>call_echo</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>(8 bytes)</td>
</tr>
<tr>
<td>Canary</td>
</tr>
<tr>
<td>(8 bytes)</td>
</tr>
<tr>
<td>[3] [2] [1] [0]</td>
</tr>
</tbody>
</table>

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

**buf ← %rsp**

```assembly
    echo:
        ...  %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

```
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
```
Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code

- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - *Does not overcome stack canaries*

- **Construct program from gadgets**
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable
Gadget Example #1

long ab_plus_c
    (long a, long b, long c)
{
    return a*b + c;
}

000000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq

rax ← rdi + rdx
Gadget address = 0x4004d4

- Use tail end of existing functions
void setval(unsigned *p) {
    *p = 3347663060u;
}

<setval>:
4004d9: c7 07 d4 48 89 c7
4004df: c3

Encodes movq %rax, %rdi

rdi ← rax
Gadget address = 0x4004dc

- Repurpose byte codes
ROP Execution

- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**
Crafting an ROB Attack String

<table>
<thead>
<tr>
<th>Stack Frame for <code>call echo</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00 48 83 80 00 00 00 00</td>
</tr>
<tr>
<td>00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 f6 00 00 00 00</td>
</tr>
<tr>
<td>33 32 31 30 00 00 00 00</td>
</tr>
<tr>
<td>39 38 37 36 00 00 00 00</td>
</tr>
<tr>
<td>35 34 33 32 00 00 00 00</td>
</tr>
<tr>
<td>31 30 39 38 00 00 00 00</td>
</tr>
<tr>
<td>37 36 35 34 00 00 00 00</td>
</tr>
<tr>
<td>33 32 31 30 00 00 00 00</td>
</tr>
</tbody>
</table>

**Gadget**

```
00000000004004d0 <ab_plus_c>:
  4004d0: 48 0f af fe imul %rsi,%rdi
  4004d4: 48 8d 04 17 lea (%rdi,%rdx,1),%rax
  4004d8: c3 retq
```

**Attack:** `int echo()` returns `rdi + rdx`

```
int echo() {
    char buf[4];
    gets(buf);
    ...
    return ...;
}
```

**Attack String (Hex)**

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
```

d4 04 40 00 00 00 00 00 00

Multiple gadgets will corrupt stack upwards
Summary

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
  - Code Injection Attack
  - Return Oriented Programming
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

- **You will learn some of the tricks in attacklab**
  - Hopefully to convince you to never leave such holes in your programs!!
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
Example 2: IM War

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