ADVANCED TOPICS IN ASSEMBLY

Spring, 2016
Euiseong Seo
(euiseong@skku.edu)
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- **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: 400000 000000
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 $\sim2^{47}$

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

```
local          0x00007ffe4d3be87c
p1            0x00007f7262a1e010
p3           0x00007f7162a1d010
p4           0x0000000008359d120
p2         0x0000000008359d010
big_array    0x0000000080601060
huge_array   0x00000000601060
main()       0x00000000040060c0
useless()    0x000000000400590
```
Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
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(4) ➞ 3.14
fun(6) ➞ Segmentation fault

- Result is system specific
typedef struct {
    int a[2];
    double d;
} struct_t;

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

Explanation:

<table>
<thead>
<tr>
<th>Critical State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>a[0]</td>
</tr>
<tr>
<td>5</td>
<td>a[1]</td>
</tr>
<tr>
<td>4</td>
<td>d3 ... d0</td>
</tr>
<tr>
<td>3</td>
<td>d7 ... d4</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>?</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
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
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo()
{
    echo();
}

unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault

btw, how big is big enough?
Buffer Overflow Disassembly

echo:

```
00000000004006cf <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
```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}

Before call to gets

Stack Frame
for call_echo

Return Address
(8 bytes)

20 bytes unused

buf ← %rsp

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

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

Before call to gets

Stack Frame for call_echo

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
</tr>
</tbody>
</table>

20 bytes unused

[3] [2] [1] [0]

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
**Buffer Overflow Stack Example #1**

### After call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 f6</td>
</tr>
<tr>
<td>00 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>

```c
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-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

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>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 00 34</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>

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

```assembly
subq $24, %rsp
movq %rsp, %rdi
call gets
...
```

#### call_echo:

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

```assembly
buf ← %rsp
```

#### Call stack:

```assembly
unix> ./bufdemo-nsp
```

Type a string: 0123456789012345678901234

Segmentation Fault

Overflowed buffer and corrupted return pointer
After call to gets

Stack Frame for call_echo

<table>
<thead>
<tr>
<th>00 00 00 00</th>
</tr>
</thead>
<tbody>
<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-nsp
Type a string: 012345678901234567890123
012345678901234567890123

Overflowed buffer, corrupted return pointer, but program seems to work!
After call to gets

Stack Frame for call_echo

| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | 00 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

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
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
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
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
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!
**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”
- 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);
}
```
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

```
local  0x7ffe4d3be87c  0x7fff75a4f9fc  0x7ffeadb7c80c  0x7ffeaea2fdac  0x7ffcd452017c
```
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 ***
```
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>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add   $0x18,%rsp
40076c:  retq
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

Before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

Canary (8 bytes)

buf ← %rsp

echo:

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

After call to gets

Stack Frame for call_echo

Return Address (8 bytes)

Canary (8 bytes)

00 36 35 34
33 32 31 30

Input: 0123456

buf ← %rsp

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

Input:

0123456

Before call to gets

Saved %ebp

Stack Frame for main

[3][2][1][0]

Canary Return Address (8 bytes)

%rsp

After call to gets

20 bytes unused

Canary (8 bytes)

00 36 35 34

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
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
Use tail end of existing functions

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

Gadget address = 0x4004d4
void setval(unsigned *p) {
    *p = 3347663060u;
}

Gadget address = 0x4004dc

Repurpose byte codes
- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one