Machine-Level Programming V: Advanced Topics

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant & Dave O’Hallaron
Today

- Structures and Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
**Structure Allocation**

```c
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

- **Struct members laid out contiguously in memory**
  - Offset of each struct member determined at compile time
  - Members may be of different types

<table>
<thead>
<tr>
<th>Memory Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Structure Access

```c
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

- **Accessing Structure Member**
  - Pointer indicates first byte of structure
  - Access elements with offsets

```c
void set_i(struct rec *r, int val) {
    r->i = val;
}
```

**IA32 Assembly**

```assembly
# %edx = val, %eax = r
movl %edx, 12(%eax) # Mem[r+12] = val
```
void set_all_values (struct rec *r, int val) {
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->n;
    }
}

struct rec {
    int a[3];
    int i;
    struct rec *n;
};

Element i

# %edx = r
.L17:
    movl 12(%edx), %eax  # r->i
    movl %ecx, (%edx,%eax,4)  # r->a[i] = val
    movl 16(%edx), %edx  # r = r->n
    testl %edx, %edx  # Test r
    jne .L17  # If != 0 goto loop
Structures & Alignment

- **Unaligned Data**

  ```
  struct S1 {
      char c;
      int i[2];
      double v;
  } *p;
  ```

- **Aligned Data**
  - Important for memory management (paging/etc.)
  - Done by the compiler
  - For a primitive data type of K bytes, address is multiple of K
    - Can be inefficient usage of space
Satisfying Alignment with Structures

Alignment requirement:
1. Must align each element of a struct
2. Initial address & structure length must be multiples of the biggest alignment of a struct’s elements

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

Biggest alignment of elements: 8

- c: 3 bytes, Multiple of 8
- i[0]: 4 bytes, Multiple of 4
- i[1]: 4 bytes, Multiple of 8
- v: 8 bytes, Multiple of 8
Saving Space

- Define a struct to put large data types first

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```c
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

- Note: May not be worth optimizing
Union Allocation

- Unions can store different kinds of data in one memory allocation (but only one type at a time)
- Allocated according to largest element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```
Byte Ordering Example

union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;

32-bit

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Today

- Structures and Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
IA32 Linux Memory Layout

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

- **Heap**
  - Dynamically allocated storage
  - Used when calling `malloc()`, `calloc()`, `new()`

- **Data**
  - Statically allocated data
    - E.g., Global variables

- **Text**
  - Executable machine instructions
  - Read-only
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 ... */
}
IA32 Example Addresses

address range ~$2^{32}$

$p3$ 0x65586008
$p1$ 0x55585008
$p4$ 0x1904a110
$p2$ 0x1904a008
&amp2 0x18049760
&ampbeyond 0x08049744
big_array 0x18049780
huge_array 0x08049760
main() 0x080483c6
useless() 0x08049744
final malloc() 0x006be166

malloc() is dynamically linked
address determined at runtime
x86-64 Example Addresses

`address range ~2^47`

```
$rs p 0x00007fffffff8d1f8
p3 0x00002aaabaadd010
p1 0x00002aaaaaad010
p4 0x0000000011501120
p2 0x00000000011501010
&p2 0x00000000010500a60
&beyond 0x0000000000500a44
big_array 0x0000000010500a80
huge_array 0x0000000000500a50
main() 0x0000000000400510
useless() 0x0000000000400500
final malloc() 0x00000000386ae6a170
```

malloc() is dynamically linked
address determined at runtime
Today

- Structures and Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
Internet Worm

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
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

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

void call_echo()
{
    echo();
}

unix>./bufdemo
Type a string:1234567
1234567
unix>./bufdemo
Type a string:12345678
Segmentation Fault
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
Buffer Overflow Disassembly

echo:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80485c5</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>80485c6</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>80485c8</td>
<td>53</td>
<td>push %ebx</td>
</tr>
<tr>
<td>80485c9</td>
<td>83 ec 14</td>
<td>sub $20,%esp</td>
</tr>
<tr>
<td>80485cc</td>
<td>8d 5d f8</td>
<td>lea -8(%ebp),%ebx</td>
</tr>
<tr>
<td>80485cf</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>80485d2</td>
<td>e8 9e ff ff ff ff</td>
<td>call 8048575 &lt;gets&gt;</td>
</tr>
<tr>
<td>80485d7</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>80485da</td>
<td>e8 05 fe ff ff ff</td>
<td>call 80483e4 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>80485df</td>
<td>83 c4 14</td>
<td>add $20,%esp</td>
</tr>
<tr>
<td>80485e2</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>80485e3</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>80485e4</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
```

call_echo:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80485eb</td>
<td>e8 d5 ff ff ff ff</td>
<td>call 80485c5 &lt;echo&gt;</td>
</tr>
<tr>
<td>80485f0</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>80485f1</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
```
Buffer Overflow Stack

Before call to gets

Stack Frame for main

- Return Address
- Saved %ebp
- Saved %ebx
- [3][2][1][0]

Stack Frame for echo

-- %ebp
-- buf

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

echo:
    pushl %ebp      # Save %ebp on stack
    movl %esp, %ebp
    pushl %ebx      # Save %ebx
    subl $20, %esp  # Allocate stack space
    leal -8(%ebp),%ebx  # Compute buf as %ebp-8
    movl %ebx, (%esp)  # Push buf on stack
    call gets        # Call gets
    ...

Return Address Saved %ebp %ebp
Saved %ebx %ebx
[3][2][1][0]
Buffer Overflow
Stack Example

Before call to gets

Stack Frame for **main**

<table>
<thead>
<tr>
<th>Return Address</th>
<th>Saved %ebp</th>
<th>Saved %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>buf</td>
<td>[3] [2] [1] [0]</td>
</tr>
</tbody>
</table>

Stack Frame for **echo**

<table>
<thead>
<tr>
<th>Saved %ebp</th>
<th>Saved %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>[3] [2] [1] [0]</td>
</tr>
</tbody>
</table>

Before call to gets

Stack Frame for **main**

| 08 04 85 f0 | ff ff d6 88 |

Saved %ebx

| xx xx xx xx |

Stack Frame for **echo**

| 80485eb: call 80485c5 <echo> |
| 80485f0: leave |

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xfffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xfffffd88
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
Buffer Overflow Example #1

Before call to gets

Stack Frame for main

08 04 85 f0
ff ff d6 88
Saved %ebx

buf

Stack Frame for echo

Input 1234567\0

Stack Frame for main

08 04 85 f0
ff ff d6 88

0x00 37 36 35
34 33 32 31
buf

Stack Frame for echo

Overflow buf, and corrupt %ebx
Buffer Overflow Example #2

Before call to gets

Input 12345678\0

Stack Frame for main

Before call to gets

Input 12345678\0

Stack Frame for main

echo:
...
call 8048575 <gets>
leave # Reset %ebp to corrupted value
ret
Buffer Overflow Example #3

Before call to gets

Stack Frame for main

0xffffd688

08 04 85 f0
ff ff d6 88
Saved %ebx
xx xx xx xx

buf

0xffffd678

Stack Frame for echo

Input 123456789ABC\0

Stack Frame for main

0xffffd688

08 04 85 00
43 42 41 39
38 37 36 35
34 33 32 31
buf

echo:
... call 8048575 <gets>
leave          # Reset %ebp to corrupted value
ret
**Malicious Use of Buffer Overflow**

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When \( \text{bar}() \) executes `ret`, will jump to exploit code
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 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.
Code Red Exploit Code

- Exploited bug in Microsoft IIS web server
- Spread self
  - Generate random IP addresses & send attack string
  - Infected >300,000 hosts
- Attack www.whitehouse.gov
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
- Deface server’s home page
  - After waiting 2 hours
Avoiding Overflow Vulnerability

/* 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
    - Or use `%ns` 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

- **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

```text
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffffffc6a8
```
Compile-Level Protection: 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`

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

```assembly
push   %ebp
mov    %esp,%ebp
push   %ebx
sub    $20,%esp
mov    %gs:0x14,%eax
mov    %eax,0xfffffffff8(%ebp)
xor    %eax,%eax
lea    0xffffffff4(%ebp),%ebx
mov    %ebx,(%esp)
call   80485e1 <gets>
call   804843c <puts@plt>
add    $20,%esp
pop    %ebx
pop    %ebp
ret
```
Setting Up Canary

Before call to gets

Stack Frame for main

- Return Address
- Saved %ebp
- Saved %ebx
- Canary
  [3][2][1][0]
- Stack Frame for echo

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

Echo:  
...
movl %gs:20, %eax  # Get canary
movl %eax, -8(%ebp) # Put on stack
xorl %eax, %eax  # Erase canary
...
Checking Canary

Before call to `gets`

Stack Frame for `main`

Return Address

Saved `%ebp`

Saved `%ebx`

Canary

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

Stack Frame for `echo`

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

Echo:

```
    movl -8(%ebp), %eax       # Retrieve from stack
    xorl %gs:20, %eax         # Compare with Canary
    je .L24                   # Same: skip ahead
    call __stack_chk_fail     # ERROR
    .L24:                      # ...
```
Canary Example

**Before call to gets**

<table>
<thead>
<tr>
<th>Stack Frame for <strong>main</strong></th>
<th>Stack Frame for <strong>echo</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>Saved <code>%ebp</code></td>
<td></td>
</tr>
<tr>
<td>Saved <code>%ebx</code></td>
<td></td>
</tr>
<tr>
<td>03 e3 7d 00</td>
<td>[3] [2] [1] [0]</td>
</tr>
</tbody>
</table>

- buf

**Input 1234**

<table>
<thead>
<tr>
<th>Stack Frame for <strong>main</strong></th>
<th>Stack Frame for <strong>echo</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>Saved <code>%ebp</code></td>
<td></td>
</tr>
<tr>
<td>Saved <code>%ebx</code></td>
<td></td>
</tr>
<tr>
<td>03 e3 7d 00</td>
<td>34 33 32 31</td>
</tr>
</tbody>
</table>

- buf

(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00

Benign corruption!
(allow programmers to make silent off-by-one errors)