Machine-Level Programming V: Advanced Topics

Kai Shen

Outline

- Structures and Alignment
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

Structure Layout

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

Alignment Principles

- Aligned data
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K_a$
  - $K_a$ is usually a power of 2, $K_a = K$ in strongest alignment
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!
- Motivation for aligning data
  - Memory accessed physically by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans “word” boundaries
  - Virtual memory very tricky when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields
**Structures & Alignment**

- **Unaligned Data**
  
  - Primitive data type requires \( K \) bytes
  - Address must be multiple of \( K \)

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

- **Aligned Data**
  
  - Primitive data type requires \( K \) bytes
  - Address must be multiple of \( K \)

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

**Satisfying Alignment with Structures**

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement:
  - Each structure has alignment requirement \( K_A \)
  - \( K_A \) = largest alignment of any element
  - Initial address & structure length must be multiples of \( K_A \)
- Example (under Windows or x86-64):
  - \( K_A = 8 \), due to `double`

**Arrays of Structures**

- End padding to satisfy alignment requirement for every element

**Accessing Array Elements**

- `short get_j(int idx)`

  ```
  short get_j(int idx) {
    return a[idx].j;
  }
  ```

  ```
  # %eax = idx
  leal (%eax,%eax,2),%eax # 3*idx
  movswl a+8(%eax,%eax,4),%eax
  ```

- **Assembler gives offset `a+8`**
- **Resolved during linking**
Saving Space

- Put large data types first

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

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

Outline

- Structures and Alignment
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

Union Allocation

- Allocate according to largest element
- Can only use one field at a time

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

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

Example of A Multi-Type Union

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

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- **Big endian**
  - Most significant byte has lowest address
  - Sparc
- **Little endian**
  - Least significant byte has lowest address
  - Intel x86

---

**Summary**

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Multi-type interpretation of the same data

---

**Outline**

- **Structures and Alignment**
- **Unions**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

---

**IA32 Linux Memory Layout**

- **Stack**
  - Runtime stack (8MB 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

---
Internet Worm

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

Instant Messenger War

- July 1999
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging (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
  - How did it happen?
    - Note that AOL does not require reinstallation of its own AOL clients

Stack Buffer Overflow Exploits

- The Internet Worm and AOL/Microsoft War were both based on stack buffer overflow exploits!
- Buffer on stack overflows, then overwrites stack data, then causes trouble or hijacks the machine

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;
  }
  ```
- Does not allow specification of 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
Type a string:
1234567
1234567
unix>
./bufdemo
Type a string:
12345678
Segmentation Fault
unix>
./bufdemo
Type a string:
123456789ABC
Segmentation Fault

Buffer Overflow Stack Example

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

Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 ff 08 ff dd 88</td>
</tr>
<tr>
<td>Saved %ebp</td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>Stack Frame for echo</td>
</tr>
</tbody>
</table>

Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 70 04 0f 8f ff</td>
</tr>
<tr>
<td>Saved %ebp</td>
</tr>
<tr>
<td>Saved %ebx</td>
</tr>
<tr>
<td>03 13 11 01</td>
</tr>
<tr>
<td>Stack Frame for echo</td>
</tr>
</tbody>
</table>

Base pointer corrupted

Buffer Overflow Example #1

Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 ff 08 ff dd 88</td>
</tr>
<tr>
<td>Saved %ebp</td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>Stack Frame for echo</td>
</tr>
</tbody>
</table>

Input 1234567

<table>
<thead>
<tr>
<th>Stack Frame for caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 70 04 0f 8f ff</td>
</tr>
<tr>
<td>Saved %ebp</td>
</tr>
<tr>
<td>Saved %ebx</td>
</tr>
<tr>
<td>03 13 11 01</td>
</tr>
<tr>
<td>Stack Frame for echo</td>
</tr>
</tbody>
</table>

Overflow buf, and corrupt %ebx, but no problem
* because %ebx isn’t used by caller

Buffer Overflow Example #2

Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 ff 08 ff dd 88</td>
</tr>
<tr>
<td>Saved %ebp</td>
</tr>
<tr>
<td>buf</td>
</tr>
<tr>
<td>Stack Frame for echo</td>
</tr>
</tbody>
</table>

Input 1234567

<table>
<thead>
<tr>
<th>Stack Frame for caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 70 04 0f 8f ff</td>
</tr>
<tr>
<td>Saved %ebp</td>
</tr>
<tr>
<td>Saved %ebx</td>
</tr>
<tr>
<td>03 13 11 01</td>
</tr>
<tr>
<td>Stack Frame for echo</td>
</tr>
</tbody>
</table>

Base pointer corrupted
Buffer Overflow Example #3

Before call to `gets`

```
Stack Frame for caller
08 04 85 00
ff ff de 88
Saved %ebx
AX AX AX AX
Stack Frame for echo
```

Input `123456789ABC`

```
Stack Frame for caller
08 04 85 00
43 42 41 39
Stack Frame for echo
```

Return address corrupted

Malicious Use of Buffer Overflow

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

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

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

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 kshen@cycle1.cs.rochester.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.

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 (WITHOUT requiring reinstallation of its own AOL clients).
### Avoiding Overflow Vulnerability

```c
/* 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

### Stack Canaries

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC implementation
  - `-fstack-protector`

```c
unix> ./bufdemo-protected
Type a string: 123
unix> ./bufdemo-protected
Type a string: 123
```

### Setting Up Canary for Suspicious Data Structures

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

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC implementation
  - `-fstack-protector`

```c
unix> gcc bufdemo
unix> gdb bufdemo
(gdb) break echo
(gdb) run
```

```mlist
Canary
Stack Frame for echo
Saved %ebp
Saved %ebx
```

```c
unix> gdb bufdemo
(gdb) break echo
(gdb) run
```

```c
movl %gs:20, %eax  # Get canary
movl %eax, -8(%ebp)  # Put on stack
xorl %eax, %eax  # Erase canary
```
Checking Canary before Restoring Registers and Returning

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

Stack Frame for caller:
- Return Address
- Saved ebp
- Saved ebx
- Canary
- buf
- Saved %ebp

Disco farmer:
- 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:
  ...

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.