Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Alignment
Structures & Alignment

• Unaligned Data

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
<td>p+9</td>
</tr>
</tbody>
</table>

• Aligned Data

- For a primitive data type of $K$ bytes, address is a multiple of $K$
  - Inefficient to load or store data that spans word boundaries

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

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

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i[0]</th>
<th>i[1]</th>
<th>4 bytes</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
<td>p+24</td>
<td></td>
</tr>
</tbody>
</table>

- Multiple of 4
- Multiple of 8
- Multiple of 8
- Multiple of 8

Biggest alignment of elements: 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;
```

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
</table>

| i | c | d | 2 bytes |
Union Allocation

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

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

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
Memory Layout
IA32 Linux Memory Layout

• Stack
  – Local variables

• Heap
  – Dynamically allocated memory
  – When calling malloc, new

• Data
  – Statically allocated variables declared in code
    • E.g. Global variables

• Text
  – Executable machine instructions
  – Read-only
Memory Allocation Example

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 ... */
}

Where does everything go?
About Security!
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`
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

Before call to gets

Stack Frame for call_echo

Return Address
Saved %ebp
Saved %ebx
buf

Stack Frame for echo

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

pushl %ebp # Save %ebp on stack
movl %esp, %ebp # Save %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

...
Buffer Overflow Stack Example

Before call to gets

Stack Frame for call_echo

Return Address
Saved %ebp
Saved %ebx

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

Stack Frame for echo

Before call to gets

Stack Frame for call_echo

Stack Frame for echo

80485eb: call 80485c5 <echo>
80485f0: leave
Buffer Overflow Example #1

Before call to gets

Stack Frame for call_echo

Stack Frame for echo

Input 1234567

Stack Frame for call_echo

Stack Frame for echo

Overflow buf, and corrupt %ebx

Note: ASCII of 1→31, 2→32, ..., 9→39
Buffer Overflow Example #2

Before call to gets

```
Input 12345678

Stack Frame for call_echo

Saved %ebx

buf

Stack Frame for echo
```

echo:
.
.
.
call 8048575 <gets>
leave                 # Set %ebp to corrupted value
ret

Note: leave is equivalent to movl %ebp, %esp and pop %ebp
Buffer Overflow Example #3

Before call to `gets`

```
Stack Frame for call_echo

<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>
Saved %ebx

Stack Frame for echo

| xx | xx | xx | xx |
```

```
Input 123456789

<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>42</td>
<td>41</td>
<td>39</td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>
```

```
echo:
    ...
call 8048575 <gets>
leave          # Set %ebp to corrupted value
ret           # pop and return to corrupted return address
```
Malicious Use of Buffer Overflow

- 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**

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

text

```

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

Stack after call to `gets()`

- **foo** stack frame
- **bar** stack frame

- **data written by** `gets()`
- **pad**
- **exploit code**
Avoiding Overflow Vulnerability

- Use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don’t use `scanf` with `%s`
    - 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);
}
```
System-Level Protections

• Randomized stack offsets
  – At start of program, allocate random amount of space on stack
  – Makes it difficult for hacker to predict address of inserted code

• Non-executable code segments
  – In old x86, memory is marked as either “read-only” or “writeable”
    • Can execute anything readable
  – X86-64 added explicit “execute” permission
    • Mark stack as non-executable
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

unix>./bufdemo-protected
Type a string: 1234
1234

unix>./bufdemo-protected
Type a string: 12345
*** stack smashing detected ***
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:
   . . .
movl %gs:20, %eax       # Get canary
movl %eax, -8(%ebp)     # Put on stack
xorl %eax, %eax         # Erase canary
   . . .

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

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary
Stack Frame for echo

buf

.*
... 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:
        ...
Conclusions

• We have looked at the main characteristics of x86 assembly (i.e. IA 32)
• We took a glimpse at x86_64
• It is now very useful that you write some simple C code, compile it with `gcc -S -m32` and compare it to assembly version
• Buffer-overflow is a security breach to a code that you would have never figured it out if you did not know assembly!