Computer Systems
CSE 410 Spring 2012
9 – Memory Allocation and Buffer Overflow
Buffer Overflow

- Basics of memory allocation
- Buffers on stacks
- Overwriting buffers
- Injecting code
IA32 Linux Memory Layout

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

- **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

Upper 2 hex digits = 8 bits of address
Memory Allocation Example

```c
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?
IA32 Example Addresses

address range ~$2^{32}$

为首的 0xffffbcd0
p3  0x65586008
p1  0x55585008
p4  0x1904a110
p2  0x1904a008
&p2  0x18049760
beyond  0x08049744
big_array  0x18049780
huge_array  0x08049760
main()  0x080483c6
useless()  0x08049744
final malloc()  0x006be166

malloc() is dynamically linked
address determined at runtime

not drawn to scale
Internet Worm

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

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

- The Internet Worm was based on stack buffer overflow exploits!
  - many Unix functions do not check argument sizes
  - allows target buffers to overflow
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;
}
```

- Anything interesting in the above?
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 Unix functions
  - `strcpy`: Copies string 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);
}

int main()
{
    printf("Type a string: ");
    echo();
    return 0;
}
```

```
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

080484f0 <echo>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80484f0:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>80484f1:</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>80484f3:</td>
<td>53</td>
<td>push %ebx</td>
</tr>
<tr>
<td>80484f4:</td>
<td>8d 5d f8</td>
<td>lea 0xfffffffffffffff8(%ebp),%ebx</td>
</tr>
<tr>
<td>80484f7:</td>
<td>83 ec 14</td>
<td>sub $0x14,%esp</td>
</tr>
<tr>
<td>80484fa:</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>80484fd:</td>
<td>e8 ae ff ff ff</td>
<td>call 80484b0 &lt;gets&gt;</td>
</tr>
<tr>
<td>8048502:</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>8048505:</td>
<td>e8 8a fe ff ff</td>
<td>call 8048394 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>804850a:</td>
<td>83 c4 14</td>
<td>add $0x14,%esp</td>
</tr>
<tr>
<td>804850d:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>804850e:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>804850f:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80485f2:</td>
<td>e8 f9 fe ff ff</td>
<td>call 80484f0 &lt;echo&gt;</td>
</tr>
<tr>
<td>80485f7:</td>
<td>8b 5d fc</td>
<td>mov 0xfffffffffffffff8c(%ebp),%ebx</td>
</tr>
<tr>
<td>80485fa:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>80485fb:</td>
<td>31 c0</td>
<td>xor %eax,%eax</td>
</tr>
<tr>
<td>80485fd:</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

Stack Frame for echo

/* 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
    leal -8(%ebp),%ebx  # Compute buf as %ebp-8
    subl $20, %esp    # Allocate stack space
    movl %ebx, (%esp) # Push buf addr on stack
    call gets      # Call gets

...
Buffer Overflow Stack Example

Before call to gets

Stack Frame for main

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

Before call to gets

Stack Frame for main

0xfffffc658
f7 85 04 08
58 c6 ff ff
Stack Frame for echo

buf

80485f2: call 80484f0 <echo>
80485f7: mov 0xfffffffff(%ebp),%ebx # Return Point
Buffer Overflow Example #1

Before call to gets

Stack Frame for main

0xfffffc658

Stack Frame for main

f7 85 04 08
58 c6 ff ff

Stack Frame for echo

xff xff xff xff

Input 1234567

Stack Frame for main

0xfffffc658

Stack Frame for main

f7 85 04 08
58 c6 ff ff

Stack Frame for echo

00 37 36 35
34 33 32 31

Overflow buf, but no problem
Buffer Overflow Example #2

Before call to gets

Stack Frame for main
0xffffffffc658

Stack Frame for echo
0xffffffffc638

Input 12345678

Stack Frame for main
0xffffffffc658

Stack Frame for echo
0xffffffffc638

Base pointer corrupted

804850a: 83 c4 14  add $0x14,%esp  # deallocate space
804850d: 5b        pop    %ebx  # restore %ebx
804850e: c9        leave  # movl %ebp, %esp; popl %ebp
804850f: c3        ret  # Return
Buffer Overflow Example #3

Before call to gets

Stack Frame for main

<table>
<thead>
<tr>
<th>f7</th>
<th>85</th>
<th>04</th>
<th>08</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>c6</td>
<td>ff</td>
<td>ff</td>
</tr>
</tbody>
</table>

Stack Frame for echo

| xx | xx | xx | xx |

buf

0xfffffc658

Input 123456789ABC

Stack Frame for main

<table>
<thead>
<tr>
<th>f7</th>
<th>85</th>
<th>04</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>

buf

0xfffffc638

Return address corrupted

80485f2: call 80484f0 <echo>
80485f7: mov 0xffffffffc(%ebp),%ebx # Return Point
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Stack frame must be big enough to hold exploit code
- Overwrite return address with address of buffer (need to know B)
- When `bar()` executes `ret`, will jump to exploit code (instead of A)
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
Avoiding Overflow Vulnerability

Use library routines that limit string lengths
- `fgets` instead of `gets` (second argument to `fgets` sets limit)
- `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
  - Only allow code to execute from “text” sections of memory
  - Do NOT execute code in stack, data, or heap regions
  - Hardware support