Today

- Memory layout
- Buffer overflow, worms, and viruses
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

- Not drawn to scale
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 ... */
}
IA32 Example Addresses

_address range \sim 2^{32}\_

\begin{align*}
  \$\text{esp} & : 0xfffffbc0 \\
  p3 & : 0x65586008 \\
  p1 & : 0x55585008 \\
  p4 & : 0x1904a110 \\
  p2 & : 0x1904a008 \\
  \&p2 & : 0x18049760 \\
  \text{beyond} & : 0x08049744 \\
  \text{big\_array} & : 0x18049780 \\
  \text{huge\_array} & : 0x08049760 \\
  \text{main()} & : 0x080483c6 \\
  \text{useless()} & : 0x08049744 \\
  \text{final malloc()} & : 0x006be166
\end{align*}

\text{malloc()} \text{ is dynamically linked}

address determined at runtime

\textit{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?
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;
    }
    *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

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>80484f0:</td>
<td>push %ebp</td>
</tr>
<tr>
<td>80484f1:</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>80484f3:</td>
<td>push %ebx</td>
</tr>
<tr>
<td>80484f4:</td>
<td>lea 0xfffffffff8(%ebp),%ebx</td>
</tr>
<tr>
<td>80484f7:</td>
<td>sub $0x14,%esp</td>
</tr>
<tr>
<td>8048fa:</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>8048fd:</td>
<td>call 80484b0 &lt;gets&gt;</td>
</tr>
<tr>
<td>8048502:</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>8048505:</td>
<td>call 8048394 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>804850a:</td>
<td>add $0x14,%esp</td>
</tr>
<tr>
<td>80485d:</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>80485e:</td>
<td>leave</td>
</tr>
<tr>
<td>80485f:</td>
<td>ret</td>
</tr>
<tr>
<td>80485f2:</td>
<td>call 80484f0 &lt;echo&gt;</td>
</tr>
<tr>
<td>80485f7:</td>
<td>mov 0xfffffffff8(%ebp),%ebx</td>
</tr>
<tr>
<td>8048fa:</td>
<td>leave</td>
</tr>
<tr>
<td>8048fb:</td>
<td>xor %eax,%eax</td>
</tr>
<tr>
<td>8048fd:</td>
<td>ret</td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

**Before call to gets**

```
Stack Frame for main

Return Address
Saved %ebp

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

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  # Save %esp
    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

80485f2: call 80484f0 <echo>
80485f7: mov 0xfffffffffc(%ebp),%ebx # Return Point

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

0xffffffffc658

Stack Frame for echo

0xffffffffc638

buf

buf

0xffffffffc658

0xffffffffc638

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

**Before call to gets**

Stack Frame for main

0xffffffffc658

<table>
<thead>
<tr>
<th>ff</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

0xffffffffc638

| xx | xx | xx | xx |

Input 1234567

Stack Frame for main

0xffffffffc658

<table>
<thead>
<tr>
<th>ff</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>

<table>
<thead>
<tr>
<th>00</th>
<th>37</th>
<th>36</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

buf

Overflow buf, but no problem
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

<table>
<thead>
<tr>
<th>0xfffffc658</th>
</tr>
</thead>
<tbody>
<tr>
<td>f7 85 04 08</td>
</tr>
<tr>
<td>58 c6 ff ff</td>
</tr>
<tr>
<td>xx xx xx xx</td>
</tr>
</tbody>
</table>

Stack Frame for echo

Input 12345678

Stack Frame for main

<table>
<thead>
<tr>
<th>0xfffffc658</th>
</tr>
</thead>
<tbody>
<tr>
<td>f7 85 04 08</td>
</tr>
<tr>
<td>58 c6 ff 00</td>
</tr>
<tr>
<td>38 37 36 35</td>
</tr>
<tr>
<td>34 33 32 31</td>
</tr>
</tbody>
</table>

buf

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`
- `0xffffffffc658`
- `0xffffffffc638`
- `buf`
- Stack Frame for `echo`

Input 123456789ABC

- Stack Frame for `main`
- `0xffffffffc658`
- `0xffffffffc638`
- `buf`
- Stack Frame for `echo`

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.
Code Red Worm

History

- June 18, 2001. Microsoft announces buffer overflow vulnerability in IIS Internet server
- July 19, 2001. over 250,000 machines infected by new virus in 9 hours
- White house must change its IP address. Pentagon shut down public WWW servers for day
Code Red Exploit Code

- Starts 100 threads running
- Spread self
  - Generate random IP addresses & send attack string
  - Between 1st & 19th of month
- Attack www.whitehouse.gov
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
    - Between 21st & 27th of month
- Deface server’s home page
  - After waiting 2 hours
- Later versions even more aggressive
- And it goes on still...
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` (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
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
  - Cannot run independently

- **Both are (usually) designed to spread among computers and to wreak havoc (and, these days, profit$$$)**