Today

- Buffer overflow example
- Memory hierarchy organization

- But first, Midterm info! 🎉
  - Midterm to be held next Wednesday, May 4th, in class.
  - Tomorrow’s section will be a review for the midterm.
  - And the structure is...

Buffer Overflow Disassembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Register/Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0048400</td>
<td>push</td>
<td>%ebp</td>
</tr>
<tr>
<td>0048401</td>
<td>89</td>
<td>5</td>
</tr>
<tr>
<td>0048402</td>
<td>push</td>
<td>%ebp</td>
</tr>
<tr>
<td>0048403</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>0048404</td>
<td>8d</td>
<td>5f</td>
</tr>
<tr>
<td>0048405</td>
<td>lea</td>
<td>0xfffff0(%ebp),%edx</td>
</tr>
<tr>
<td>0048407</td>
<td>83</td>
<td>ec 14</td>
</tr>
<tr>
<td>0048409</td>
<td>add</td>
<td>$0x14,%esp</td>
</tr>
<tr>
<td>004840a</td>
<td>89</td>
<td>1c 24</td>
</tr>
<tr>
<td>004840c</td>
<td>mov</td>
<td>%edi,%ebp</td>
</tr>
<tr>
<td>004840c</td>
<td>80</td>
<td>48</td>
</tr>
<tr>
<td>004840d</td>
<td>lea</td>
<td>0x0048400(%esp)</td>
</tr>
<tr>
<td>004840f</td>
<td>80</td>
<td>5d f6</td>
</tr>
<tr>
<td>0048411</td>
<td>mov</td>
<td>0xfffff0(%ebp),%ebx</td>
</tr>
<tr>
<td>0048413</td>
<td>leave</td>
<td></td>
</tr>
<tr>
<td>0048414</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

Buffer Overflow Stack

Before call to gets:
- Stack Frame for main
- Return Address
- %ebp
- %esp
- %eax
- %ebx
- %ecx
- %edx

Before call to gets:
- Stack Frame for gets
- %ebp
- %esp
- %eax
- %ebx
- %ecx
- %edx

Vulnerable Buffer Code

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

Buffer Overflow Example

Before call to gets:
- Stack Frame for main
- Return Address
- %ebp
- %esp
- %eax
- %ebx
- %ecx
- %edx

Before call to gets:
- Stack Frame for gets
- %ebp
- %esp
- %eax
- %ebx
- %ecx
- %edx

Buffer Overflow Example #1

Before call to gets:
- Stack Frame for main
- Return Address
- %ebp
- %esp
- %eax
- %ebx
- %ecx
- %edx

Overflow but, no problem

Buffer Overflow Example
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

0x2000000
0x2000000
buf

Stack Frame for echo

0x2000060
0x2000060
buf

Base pointer corrupted

Before call to gets

Stack Frame for main

0x2000000
0x2000000
buf

Stack Frame for echo

0x2000060
0x2000060
buf

Return address corrupted

Malicious Use of Buffer Overflow

void foo() {
    bar();
    ... // return address A
}

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 rechts.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 & 39th 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...
How could we avoid buffer overflows vulnerabilities?

```
#include <stdio.h>

int main()
{
    int size = 10;
    char buf[size] = “Hi, you can’t limit to buf size”;
    gets(buf); /* Way too small! */
}
```

**Avoiding Overflow Vulnerability**

- Use library routines that limit string lengths
  - `fgets` instead of `gets` (second argument to `fgets` sets limit)
  - `strlen` instead of `strncpy`
  - Don’t use `scanf` with its conversion specification
    - Use `fgets` to read the string
    - Or use `vsnprintf` 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$)