Machine Programming 5: Buffer Overruns and Stack Exploits

CS61, Lecture 6
Prof. Stephen Chong
September 22, 2011
Thinking about grad school in Computer Science?

Panel discussion
Tuesday September 27th, 6:00pm
Maxwell Dworkin 119

CS faculty and grad students will answer your questions about grad school in Computer Science: Why to apply, how to apply, how to get in, research, reference letters, personal statement, common pitfalls, what to do during your sophomore and junior years, and more…

Undergraduates at all levels are encouraged to attend.
Questions? Email chong@seas.harvard.edu

Pizza will be served!
Announcements

- HW 2 (Binary bomb) due tonight
- HW 3 (Buffer bomb) will be released today
  - Due Thurs Oct 6 (2 weeks)

- Final will be in class on Thurs 1 Dec
  - Extension school final will also be on or around 1 Dec
Memory vulnerabilities

• Many C programs contain subtle bugs that can lead to remote exploits

• Most common case: **Buffer overflow attacks**
  • Program reads data into a fixed-size buffer
  • Remote attacker feeds program data that overflows the buffer
  • How can this lead to a security hole?

• Buffer overflow overwrites other memory used by the program
  • For example, the return address on the stack

• Attacker sends machine instructions that end up being executed by the remote host!
  • Allows the attacker to cause the remote machine to run (almost) any code.
Real vulnerabilities

- Internet Worm: 1988
  - First widespread worm on the Internet
  - Estimated infected 10% of machines on the Internet
- Code Red, Code Red II, NIMDA, SQL Slammer
  - Various worms that attacked Windows machines
  - Led to denial of service attacks, backdoors, web pages being defaced, etc.
- AOL vs. Microsoft in the Internet Messaging Wars
  - AOL exploiting a buffer overrun in its own AIM client
- iPhone jail breaking, Xbox modding, Wii modding...
- Homework 3!!
The Internet Worm

• November 2, 1988: One of first large-scale worm attacks on the Internet launched
  • At the time, just 60,000 machines on Internet
  • Most were VAX or Sun machines running BSD UNIX
• Worm repeatedly infected machines, causing huge load, slow down, lots of weird activity
  • At first it was not clear what was going on
  • Lots of universities and companies notice the attack
• Very rapid response by the community
  • Nov 3, teams at MIT and Berkeley “capture” worm and disassemble it
  • Within few days they have a basic understanding of how it works, and patches to prevent its spread
• See “The Internet Worm Program: An Analysis” by Eugene H. Spafford (Purdue Technical Report CSD-TR-823)
Details of the Worm

- Three basic attack mechanisms:
  - 1) Exploited debugging “feature” of sendmail
    - Allowed remote user to send an email with a program as the recipient
    - Caused remote machine to interpret email message as a shell script!
    - Shell script extracted a C program from the message, compiled it, and ran it
  - 2) Exploited rsh “.rhosts” feature
    - rsh allows users to create file of machines trusted to log in with no password!
    - Worm cracks user's password locally, sh to that user, then rsh to remotely
  - 3) **Buffer overflow** in fingerd
    - Finger daemon (fingerd) provides info on users on machine
    - fingerd reads its input insecurely, allows arbitrary code to run within fingerd
    - Since fingerd generally runs as root, gives remote user root access!
Example: gets() library routine

```c
/* Get string from stdin */
char *gets(char *dest){
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- `char *gets(char*)` reads a string from stdin and stores it in buffer provided by caller
- What’s wrong with this code?
Example: gets() library routine

- Does not check the size of buffer `dest`!
  - No way to check it: not passed in as an argument

- Similar problems with other Unix functions
  - `strcpy`: copy string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` specification
Example of badly written 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;
}
```
What happens when we run?

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

[chong@cs61 ~]$ ./bufdemo
Type a string: 123
123
[chong@cs61 ~]$ ./bufdemo
Type a string: 123456
123456
[chong@cs61 ~]$ ./bufdemo
Type a string: 1234567890
1234567890
Segmentation fault
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

echo:
pushl %ebp    # save %ebp on stack
movl %esp, %ebp
subl $12, %esp  # allocate space on stack
leal -4(%ebp), %eax  # %eax = buf = %ebp-4
movl %eax, (%esp)  # push buf on stack
call gets        # call gets
...
Stack layout for echo()

Stack frame for main

<table>
<thead>
<tr>
<th>%ebp</th>
<th>08</th>
<th>04</th>
<th>86</th>
<th>4d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>bf</td>
<td>ff</td>
<td>f8</td>
<td>f8</td>
</tr>
<tr>
<td></td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

%ebp → %esp

Stack frame for echo

```c
#include <stdio.h>

void echo(char *buf, int len)
{
    char *temp = (char *) malloc(len + 1);
    strlcat(temp, buf, len + 1);
    printf("%s\n", temp);
    free(temp);
}
```

```c
int main()
{
    char *buf = "Hello, World!";
    echo(buf, strlen(buf));
    return 0;
}
```
Entering a string that fits in buf[]

Before Call to `gets`

```
Stack frame for main

%ebp →

08 04 86 4d
bf ff f8 f8
?? ?? ?? ??

... rest of stack frame for echo

%esp →
```

After Call to `gets` with input “123”

```
Stack frame for main

%ebp →

08 04 86 4d
bf ff f8 f8
00 33 32 31

... rest of stack frame for echo

%esp →
```
Entering a string TOO BIG for buf[]

• What if we enter the string “12345”?  
  • Will overflow the buffer  
  • Where do the extra bytes end up?  
• Overwrite the saved %ebp on the stack!  
  • What will this do to the program?

After Call to `gets` with input “12345”

Stack frame for `main`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>04</td>
<td>86</td>
<td>4d</td>
</tr>
<tr>
<td>%ebp</td>
<td>bf</td>
<td>ff</td>
<td>00</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

... rest of stack frame for `echo`

Stephen Chong, Harvard University
Entering a string TOO BIG for buf[]

- Restores incorrect value for %ebp!!!
- Restores 0xbfff0035 instead of 0xbfffff8f8

After Call to `gets` with input “12345”

```
  ... call gets    # call gets
  ... movl %ebp, %esp  # %esp = %ebp
  popl %ebp      # restore old %ebp
  ret
```

Stack frame for `main`

- %ebp → bf ff 00 35
- %esp → 34 33 32 31
- return address
- saved %ebp
- buf
- rest of stack frame for `echo`
Entering EVEN BIGGER string

• Restores incorrect value for %ebp
• Jumps to wrong return address!!!
Malicious use of buffer overflow

• If we can overwrite portions of the stack, we can cause the program to jump to an address of our choosing!
• This can be used to do all kinds of nasty things.
• Say we knew the memory address of a routine that, say, deleted all of the files in the user's home directory.
  • Most programs would not contain such a routine, but it could happen ...
  • If we can coerce the program to jump to that routine, we can do major damage.
• This attack is fundamentally limited, however...
  • Can only cause the program to run code that's already part of the program.
• How can we inject our own code into the running program?
Injecting code

- Suppose routine puts data into buffer on stack (like previous example)
- Provide **x86 machine code** as input to the routine!
  - Fill buffer with instructions we want to run
  - Overwrite return address to point to buffer

```c
void some_routine() {
    char buf[64];
    gets(buf);
}
```
Injecting code

- Suppose routine puts data into buffer on stack (like previous example)
- Provide x86 machine code as input to the routine!
  - Fill buffer with instructions we want to run
  - Overwrite return address to point to buffer
- When routine tries to return...
  - `ret` pops return address off the stack
  - But return address now points to buffer!
  - Processor starts running code in buffer!

```c
void some_routine() {
    char buf[64];
    gets(buf);
}
```
Some limitations of this attack

• Executing this attack on arbitrary programs is tricky.
  • 1) Need to know where on the stack the buffer is (and how big it is)
  • 2) Need to know where return address is on the stack (relative to the buffer).
    • Remember, you can only control what goes into the buffer (and any addresses beyond the end of the buffer).

• If you have access to the binary, this is not too difficult...
  • Can just use gdb, set breakpoints, inspect the stack, and figure it out.

• If you're attacking a service on the Internet and don't have the binary, this becomes much harder.
  • But it can be done, usually with a lot of trial and error.
Mitigating buffer overflow attacks

• Three common mechanisms
  • Stack randomization
  • Stack corruption detection
  • Non-executable memory
Stack randomization

- Exploiting stack-based buffer overflows requires knowing where buffer is in memory
  - Need to overwrite return address on stack with pointer to buffer

- One way to thwart this: **Address space randomization**
  - When kernel runs a program, it puts the stack at a (slightly) random location in memory each time.
  - Thus attacker unlikely to correctly guess buffer's address.
  - Implemented by recent Linux kernels by default.
  - *(We have disabled this on your VMs to let you do Assignment 3)*

- To thwart address space randomization...
The NOP Sled attack

• Idea: Start out buffer with long string of \texttt{nop} instructions
  • “No-op” instruction doesn’t do anything; just moves to next instruction.

• Put best guess of exploit code location in return address.
  • OK if we “undershoot” a bit.

• When program resumes execution within the NOP sled region, code will execute until it hits your exploit code.
  • Note: won't work if we “overshoot” guess of exploit code location.
Detecting stack corruption

- Try to detect when array on stack has overflowed
- Store special **canary value** (aka **guard value**) on stack
  - Generated randomly every time program is run
    - Attacker can’t predict value
- Before returning from function check canary value unchanged
  - If changed, stop execution
- Recent versions of gcc do this for functions that may be vulnerable
Non-executable memory

- Idea: limit which memory regions can hold executable code
  - Modern operating systems and hardware support different forms of memory protection
    - Readable memory, writeable memory, executable memory
  - We’ll learn more about the mechanisms that enable this
- Make stack readable, writeable, but not executable
- Note: some languages/programs dynamically generated code
  - E.g., Just-in-time (JIT) compilation of Java bytecode
  - Non-executable memory may not be a feasible in these settings
Avoiding Overflow Vulnerability

- Mitigation techniques (stack randomization, detecting stack corruption, non-executable memory) make it harder to perform buffer overflow attacks
  - But not impossible!

- How do we prevent all overflow vulnerabilities?
Avoiding Overflow Vulnerability

• Rule #1: Don't program in C!

• Java (and many other languages) make this kind of attack more or less impossible. How?

• In Java, all array accesses are bounds-checked at runtime.
  • No way to stuff data into an array beyond its size limit.

• Also, Java doesn't let you directly manipulate pointers.
  • No way to cause the program to jump to an arbitrary memory address.

• Of course, this relies on the Java Virtual Machine being free of any bugs itself...
  • No guarantees that this is the case!
Avoiding Overflow Vulnerability

• **Rule #2: Always check buffer lengths!**
  • Especially when reading data from the outside world – a user or a network socket.

• Use standard library routines that check buffer bounds
  • `fgets` instead of `gets`
  • `strncpy` instead of `strcpy` – checks length of string.
  • Don’t use `scanf` with `%s` conversion specification
    • Use `fgets` to read the string

```c
/* Echo Line */
void echo() {
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```
Buffer exploits over the network

- Attacks so far use the `gets()` routine
  - Reads a string from standard input, typically user input, or from a file
- Problems also exists in programs that read data from the network.
  - Web browsers, IM clients, MP3 players, games, ...
  - Program reads data from network into buffer on the stack, and fails to check the data fits into the buffer ⇒ vulnerable to buffer overflow exploits
- Happens a lot in the real world.

- More serious issue: Programs running as the “root” user
  - Many services on UNIX systems run as “root”: Admin user that can do anything on the machine.
    - Example: Web servers, file servers, ssh daemon, etc.
  - If you can attack these services, exploit code will run as root, and can do arbitrary damage to machine.
Code Red Worm

- June 18, 2001 – Buffer overflow vulnerability in Microsoft IIS Web server announced
- June 26, 2001 – Microsoft releases patch for vulnerability
- July 13, 2001 – Code Red v1 worm released
  - Infects machines and causes them to perform denial-of-service attacks
  - Bug in random number generator slows infection rate. New version released a few days later
- August 4, 2001 – Code Red II worm released
  - Same basic attack vector, but somewhat different behavior
How does Code Red work?

- Overflows stack of the IIS web server
  - Causes it to overwrite return address on the stack
  - IIS then jumps into the machine code in HTTP request
- Defaces server’s home page
How does Code Red work?

- Start 100 threads running
- Spread self
  - Open connections to random IP addresses and send attack string
    - May or may not be IIS
    - Between 1st and 19th of month
- Attack several static IP addresses, including www.whitehouse.gov
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
  - This is called a **denial-of-service** attack
  - Between 21st and 27th of month
  - White House had to change IP address
# The Code Red II Attack

The Code Red II Attack is a type of 

### Padding (overflows buffer)

```plaintext
<table>
<thead>
<tr>
<th>Offset</th>
<th>Hex</th>
<th>Machine code for exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>47 45 54 20 2f 64 65 66 61 75 6c 74 2e 69 64 61</td>
<td>GET /default.ida</td>
</tr>
<tr>
<td>0010</td>
<td>3f 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>?XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0020</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0030</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0040</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0050</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0060</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0070</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0080</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>0090</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>00a0</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>00b0</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>00c0</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>00d0</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>00e0</td>
<td>58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58</td>
<td>XXXXXXXXXXXXXXXX</td>
</tr>
<tr>
<td>00f0</td>
<td>58 25 75 39 30 39 30 25 75 36 38 35 38 25 75</td>
<td>X%u9090%u6858%uc</td>
</tr>
<tr>
<td>0100</td>
<td>62 64 33 25 75 37 38 30 31 25 75 39 30 39 30 25</td>
<td>bd3%u7801%u9090%</td>
</tr>
<tr>
<td>0110</td>
<td>75 36 38 35 38 25 75 63 62 64 33 25 75 37 38 30</td>
<td>u6858%ucbd3%u780</td>
</tr>
<tr>
<td>0120</td>
<td>31 25 75 39 30 39 30 25 75 36 38 35 38 25 75 63</td>
<td>1%u9090%u6858%uc</td>
</tr>
<tr>
<td>0130</td>
<td>62 64 33 25 75 37 38 30 31 25 75 39 30 39 30 25</td>
<td>bd3%u7801%u9090%</td>
</tr>
<tr>
<td>0140</td>
<td>75 39 30 39 30 25 75 38 31 39 30 25 75 30 30 63</td>
<td>u9090%u8190%u00c</td>
</tr>
<tr>
<td>0150</td>
<td>33 25 75 30 30 30 33 25 75 38 62 30 30 25 75 35</td>
<td>3%u003%u8b00%u5</td>
</tr>
<tr>
<td>0160</td>
<td>33 31 62 25 75 35 35 33 66 66 25 75 30 30 37 38 25</td>
<td>31b%53ff%u0078%</td>
</tr>
<tr>
<td>0170</td>
<td>75 30 30 30 30 25 75 30 30 3d 61 20 20 48 54 54</td>
<td>u0000%u00-a HTT</td>
</tr>
<tr>
<td>0180</td>
<td>50 2f 31 2e 30 0d 0a 43 6f 6e 74 65 6e 74 2d 74</td>
<td>P/1.0..Content-t</td>
</tr>
<tr>
<td>0190</td>
<td>79 70 65 3a 20 74 74 79 6f 66 66 69 6e 67 2e 68</td>
<td>ype: text/xml.Co</td>
</tr>
<tr>
<td>01a0</td>
<td>6e 74 65 6e 74 2d 6d 6c 0a 43 6f 6e 74 65 6e 74</td>
<td>ntent-length: 33</td>
</tr>
<tr>
<td>01b0</td>
<td>37 39 20 0d 0a 0d 0a 0c 8c 8b 01 00 00 0b e8 03</td>
<td>79 ............</td>
</tr>
<tr>
<td>01c0</td>
<td>00 cc eb fe 64 67 ff 36 00 00 64 67 89 26 00 00</td>
<td>....dg.6..dg.&amp;..</td>
</tr>
<tr>
<td>01d0</td>
<td>e8 df 02 00 00 68 04 01 00 00 8d 85 5c fe ff ff</td>
<td>.....h........</td>
</tr>
<tr>
<td>01e0</td>
<td>50 ff 55 9c 8d 85 5c fe ff ff ff ff ff 55 59 88 80</td>
<td>P.U...\P.U...@</td>
</tr>
<tr>
<td>01f0</td>
<td>10 8b 08 89 8d 58 fe ff ff ff ff 55 e4 3d 04 04</td>
<td>.....X....U...=..</td>
</tr>
</tbody>
</table>
```
The Instant Messaging Wars of 1999

- Microsoft launches MSN Messenger (instant messaging system).
- MSN clients can also access popular AOL Instant Messaging Service (AIM) servers.
The Instant Messaging Wars of 1999

- AOL wanted to prevent MSN clients from accessing its servers.  
  But, the MSN clients mimicked the AIM protocol exactly.  
  And, AOL didn't want to change their protocol – that would require that all of their users download a new client.

- Instead, AOL exploited a buffer overrun bug in their own client!  
  One case of the protocol reads a string into a buffer of size 0x100  
  AIM code was not checking that the string would fit into this size buffer

- AOL crafted an attack on their own client that would:  
  Overflow buffer with about 40 bytes of x86 code  
  Exploit code causes client to read data from a portion of the AIM binary  
  Send that data back to the server, as a kind of “signature”  
  AOL server would only accept the client if it sent back the right signature

- This attack would not work on the MSN client, of course.  
  So MSN clients could not send back the correct signature, and would be rejected.
The Instant Messaging Wars of 1999

- Microsoft caught onto this pretty quick.
  - Changed the MSN client so it would send back the right signature.
- AOL just changed the attack code slightly so a different signature would be sent back to the server.
- Microsoft changed their clients again...
- This skirmish went back and forth 13 times!
Worm vs. Virus

• **Worm**
  - Spreads from one computer to another
  - Can propagate fully working version of itself to another machine
  - Can spread without human interaction
  - Derived from *tapeworm*: a parasite that lives inside a host and uses its resources to maintain itself.

• **Virus**
  - Spreads from one computer to another
  - Attaches itself to program or file
  - Cannot exist independently
  - Requires a human action to spread (e.g., executing or opening a file)
Next Lecture

• Processor architecture
  • How does a computer implement machine-level instructions?