Buffer Overflows
CSE 351 Autumn 2020

Alt text: I looked at some of the data dumps from vulnerable sites, and it was ... bad. I saw emails, passwords, password hints. SSL keys and session cookies. Important servers brimming with visitor IPs. Attack ships on fire off the shoulder of Orion, c-beams glittering in the dark near the Tannhäuser Gate. I should probably patch OpenSSL.

http://xkcd.com/1353/
Administrivia

- hw13 due Wednesday (11/4)
- hw15 due Monday (11/9)

- Lab 3 released Wednesday, due next Friday (11/13)
  - You will have everything you need by the end of this lecture

- Midterm Group stage due tonight
  - Individual stage Thu-Fri (expect adjustments)
  - Rubric and grades will be found on Gradescope
  - We will grade as quickly as we can
Buffer Overflows

- Address space layout review
- Input buffers on the stack
- Overflowing buffers and injecting code
- Defenses against buffer overflows
Review: General Memory Layout

- **Stack**
  - Local variables (procedure context)

- **Heap**
  - Dynamically allocated as needed
  - `new`, `malloc()`, `calloc()`, ...

- **Statically-allocated Data**
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)

- **Code/Instructions**
  - Executable machine instructions
  - Read-only
Memory Allocation Example

```c
char big_array[1L<<24];  /* 16 MB */

int global = 0;

int useless() { return 0; }

int main() {
    void *p1, *p2;
    int local = 0;
    p1 = malloc(1L << 28);  /* 256 MB */
    p2 = malloc(1L << 8);  /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?
What Is a Buffer?

- A buffer is an array used to temporarily store data

- You’ve probably seen “video buffering...”
  - The video is being written into a buffer before being played

- Buffers can also store user input
Reminder: x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Arguments (if > 6 args) for this call

- **Current/ Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Caller-saved pushed before setting up arguments for a function call
  - Callee-saved pushed before using long-term registers
  - Local variables
    (if can’t be kept in registers)
  - “Argument build” area
    (Need to call a function with >6 arguments? Put them here)
Buffer Overflow in a Nutshell

- C does not check array bounds
  - Many Unix/Linux/C functions don’t check argument sizes
  - Allows overflowing (writing past the end) of buffers (arrays)

- “Buffer Overflow” = Writing past the end of an array

- Characteristics of the traditional Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory
Buffer Overflow in a Nutshell

- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

Enter input: hello

No overflow 😊
Buffer Overflow in a Nutshell

- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

Enter input: helloabcdef
Buffer Overflow in a Nutshell

- Stack grows down towards lower addresses
- Buffer grows up towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

Enter input: helloabcdef

Buffer overflow! 😞
Buffer Overflow in a Nutshell

- Buffer overflows on the stack can overwrite “interesting” data
  - Attackers just choose the right inputs

- Simplest form (sometimes called “stack smashing”)
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Try to change the return address of the current procedure

- Why is this a big deal?
  - It was the #1 *technical* cause of security vulnerabilities
    - #1 *overall* cause is social engineering / user ignorance
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;
}
```

- What could go wrong in this code?
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 to a dst
  - `scanf`, `fscanf`, `sscanf`, when given `%s` specifier
Vulnerable Buffer Code

```c
/* Echo Line */
void echo() {
    char buf[8]; /* Way too small! */
    gets(buf);
    puts(buf);
}

void call_echo() {
    echo();
}
```

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
unix> ./buf-nsp
Enter string: 1234567890123456
Segmentation fault (core dumped)
```
Buffer Overflow Disassembly (buf-nsp)

echo:

```
0000000000401146 <echo>:
  401146: 48 83 ec 18           sub $0x18,%rsp
         ...                          ... calls printf ...
  401159: 48 8d 7c 24 08        lea 0x8(%rsp),%rdi
  40115e: b8 00 00 00 00        mov $0x0,%eax
  401163: e8 e8 fe ff ff        callq 401050 <gets@plt>
  401168: 48 8d 7c 24 08        lea 0x8(%rsp),%rdi
  40116d: e8 be fe ff ff        callq 401030 <puts@plt>
  401172: 48 83 c4 18           add $0x18,%rsp
  401176: c3                      retq
```

call_echo:

```
0000000000401177 <call_echo>:
  401177: 48 83 ec 08           sub $0x8,%rsp
  40117b: b8 00 00 00 00        mov $0x0,%eax
  401180: e8 c1 ff ff ff        callq 401146 <echo>
  401185: 48 83 c4 08           add $0x8,%rsp
  401189: c3                      retq
```
Buffer Overflow Stack

Before call to gets

/* Echo Line */
void echo()
{
    char buf[8]; /* Way too small! */
    gets(buf);
    puts(buf);
}

echo:
    subq $24, %rsp
    ...
    leaq 8(%rsp), %rdi
    mov  $0x0,%eax
    call  gets
    ...

Note: addresses increasing right-to-left, bottom-to-top
Buffer Overflow Example

**Before call to gets**

Stack frame for `call_echo`

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>40</td>
<td>11</td>
<td>85</td>
</tr>
</tbody>
</table>

8 bytes unused

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>[2]</td>
<td>[1]</td>
<td>[0]</td>
</tr>
</tbody>
</table>

8 bytes unused

void echo()
{
    char buf[8];
    gets(buf);
    . . .
}

call_echo:

. . .

401180: callq 401146 <echo>
401185: add $0x8,%rsp
. . .
Buffer Overflow Example #1

**After call to gets**

Stack frame for call_echo

```
00 00 00 00
00 40 11 85
00 35 34 33
32 31 30 39
38 37 36 35
34 33 32 31
```

8 bytes unused

Note: Digit “$N$” is just $0x3N$ in ASCII!

```c
void echo()
{
    char buf[8];
    gets(buf);
    ...
}
```

call_echo:

```
00 00 00 00
00 40 11 85
00 35 34 33
32 31 30 39
38 37 36 35
34 33 32 31
```

```
void echo()
{
    char buf[8];
    gets(buf);
    ...
}
```

echo:

```
subq $24, %rsp
...
leaq 8(%rsp), %rdi
mov $0x0,%eax
call gets
...
```

```
subq $24, %rsp
...
leaq 8(%rsp), %rdi
mov $0x0,%eax
call gets
...
```

```
00 00 00 00
00 40 11 85
00 35 34 33
32 31 30 39
38 37 36 35
34 33 32 31
```

```
401180: callq 401146 <echo>
401185: add $0x8,%rsp
```

```
401180: callq 401146 <echo>
401185: add $0x8,%rsp
```

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

Overflowed buffer, but did not corrupt state
Buffer Overflow Example #2

After call to gets

void echo()
{
    char buf[8];
    gets(buf);
    ...  
}

call_echo:
...
401180: callq 401146 <echo>
401185: add $0x8,%rsp
... 

Segmentation fault (core dumped)
Overflowed buffer and corrupted return pointer

unix> ./buf-nsp
Enter string: 1234567890123456

Segmentation fault (core dumped)
Buffer Overflow Example #2 Explained

After return from echo

Stack frame for call_echo

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>11</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 bytes unused

00000000004010d0 <register_tm_clones>:
- 4010d0: lea 0x2f61(%rip),%rdi
- 4010d7: lea 0x2f5a(%rip),%rsi
- 4010de: sub %rdi,%rsi
- 4010e1: mov %rsi,%rax
- 4010e4: shr $0x3f,%rsi
- 4010e8: sar $0x3,%rax
- 4010ec: add %rax,%rsi
- 4010ef: sar %rsi
- 4010f2: je 401108
- 4010f4: mov 0x2efd(%rip),%rax
- 4010fb: test %rax,%rax
- 4010fe: je 401108
- 401100: jmpq *%rax
- 401102: nopw 0x0(%rax,%rax,1)
- 401108: retq

"Returns" to a valid instruction, but bad indirect jump so program signals SIGSEGV, Segmentation fault
Malicious Use of Buffer Overflow: Code Injection Attacks

- 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
Practice Question

- smash_me is vulnerable to stack smashing!
- What is the minimum number of characters that gets must read in order for us to change the return address to a stack address?
  - For example: (0x00 00 7f ff ca fe f0 0d)

<table>
<thead>
<tr>
<th>Previous stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 05 d1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>[0]</td>
</tr>
</tbody>
</table>

smash_me:
subq $0x40, %rsp
...
leaq 16(%rsp), %rdi
call gets
...

A. 27
B. 30
C. 51
D. 54
E. We’re lost...
Exploits Based on Buffer Overflows

Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines

- Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult

- Examples across the decades
  - Original “Internet worm” (1988)
  - Heartbleed (2014, affected 17% of servers)
    - Similar issue in Cloudbleed (2017)
  - Hacking embedded devices
    - Cars, Smart homes, Planes
Example: the original Internet worm (1988)

- Exploited a few vulnerabilities to spread
  - 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 with phony argument:
    - `finger "exploit-code padding new-return-addr"`
    - Exploit code: executed a root shell on the victim machine with a direct connection to the attacker

- Scanned for other machines to attack
  - Invaded ~6000 computers in hours (10% of the Internet)
    - see [June 1989 article](https://www.acm.org/publications/crosslistings/crosslistings-acl) in *Comm. of the ACM*
  - The author of the worm (Robert Morris*) was prosecuted...
Example: Heartbleed (2014)

**HOW THE HEARTBLEED BUG WORKS:**

**Server:** Are you still there? If so, reply “POTATO” (6 letters).

**User Meg** wants these 6 letters: POTATO.

User Meg wants pages about “irl games.” Unlocking secure records with master key 5130985733435.

User wants pages about “boats.” Unlocking secure connection using key "4538538374224.

User Meg wants these 6 letters: POTATO.
Example: Heartbleed (2014)

User Olivia from London wants pages about "he's in car why". Note: Files for IP 375.381.383.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 34 connections open. User Brendan uploaded the file elf.png (contents: 234ba962a2c9f9ff89b33ff8).

User Olivia from London wants pages about "he's in car why". Note: Files for IP 375.381.383.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 34 connections open. User Brendan uploaded the file elf.png (contents: 234ba962a2c9f9ff89b33ff8).

SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
Example: Heartbleed (2014)

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server’s master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "Colibri9!".
Heartbleed Details

- Buffer over-read in OpenSSL
  - Open source security library
  - Bug in a small range of versions
- “Heartbeat” packet
  - Specifies length of message
  - Server echoes it back
  - Library just “trusted” this length
  - Allowed attackers to read contents of memory anywhere they wanted
- Est. 17% of Internet affected
  - “Catastrophic”
  - Github, Yahoo, Stack Overflow, Amazon AWS, ...
Hacking Cars (2010)

- UW CSE research demonstrated wirelessly hacking a car using buffer overflow
- Overwrote the onboard control system’s code
  - Disable brakes, unlock doors, turn engine on/off
Hacking DNA Sequencing Tech (2017)

Computer Security and Privacy in DNA Sequencing
Paul G. Allen School of Computer Science & Engineering, University of Washington

- Potential for malicious code to be encoded in DNA!
- Attacker can gain control of DNA sequencing machine when malicious DNA is read
Dealing with buffer overflow attacks

1) Employ system-level protections
2) Avoid overflow vulnerabilities
3) Have compiler use “stack canaries”
1) System-Level Protections

- **Non-executable code segments**
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - x86-64 added explicit “execute” permission
  - **Stack marked as non-executable**
    - Do *NOT* execute code in Stack, Static Data, or Heap regions
    - Hardware support needed

Any attempt to execute this code will fail
1) System-Level Protections

- **Non-executable code segments**
  - Wait, doesn’t this fix everything?
  - Works well, but can’t always use it
  - Many embedded devices *do not* have this protection
    - *e.g.*, cars, smart homes, pacemakers

- **Some exploits still work!**
  - Return-oriented programming
  - Return to libc attack
  - JIT-spray attack

Stack after call to `gets()`

- `foo` stack frame
- `bar` stack frame
- `exploit code`
- `pad`
- `data written by `gets()``

Any attempt to execute this code will fail
1) System-Level Protections

- Randomized stack offsets
  - At start of program, allocate \textit{random} amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
  - Makes it difficult for hacker to predict beginning of inserted code

- \textbf{Example}: Address of variable \texttt{local} for when Slide 5 code executed 3 times:
  - 0x7ffd19d3f8ac
  - 0x7ffe8a462c2c
  - 0x7ffe927c905c

- Stack repositioned each time program executes
2) Avoid Overflow Vulnerabilities in Code

/* Echo Line */
void echo()
{
    char buf[8]; /* Way too small! */
fgets(buf, 8, stdin);
puts(buf);
}

- Use library routines that limit string lengths
  - fgets instead of gets (2nd 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
2) Avoid Overflow Vulnerabilities in Code

- Alternatively, don’t use C - use a language that does array index bounds check
  - Buffer overflow is impossible in Java
    - ArrayIndexOutOfBoundsException
  - Rust language was designed with security in mind
    - Panics on index out of bounds, plus more protections
3) Stack Canaries

- Basic Idea: place special value ("canary") on stack just beyond buffer
  - Secret value that is randomized before main()
  - Placed between buffer and return address
  - Check for corruption before exiting function

- GCC implementation
  - -fstack-protector
Protected Buffer Disassembly (buf)

echo:

```
401156: push %rbx
401157: sub $0x10,%rsp
40115b: mov $0x28,%ebx
401160: mov %fs:(%rbx),%rax
401164: mov %rax,0x8(%rsp)
401169: xor %eax,%eax

... call printf ...
```

```
40117d: callq 401060 <gets@plt>
401182: mov %rsp,%rdi
401185: callq 401030 <puts@plt>
40118a: mov 0x8(%rsp),%rax
40118f: xor %fs:(%rbx),%rax
401193: jne 40119b <echo+0x45>
401195: add $0x10,%rsp
401199: pop %rbx
40119a: retq
40119b: callq 401040 <__stack_chk_fail@plt>
```
Setting Up Canary

Before call to gets

Stack frame for call_echo

Return address (8 bytes)

Canary (8 bytes)

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

/* Echo Line */
void echo()
{
    char buf[8]; /* Way too small! */
    gets(buf);
    puts(buf);
}

echo:
    . . .
    movq %fs:40, %rax  # Get canary
    movq %rax, 8(%rsp)  # Place on stack
    xorl %eax, %eax    # Erase canary
    . . .

buf ← %rsp

This is extra (non-testable) material
Checking Canary

After call to gets

```c
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```assembly
echo:
    .
    movq 8(%rsp), %rax # retrieve from Stack
    xorq %fs:40, %rax # compare to canary
    jne .L4 # if not same, FAIL
    .
    .L4: call __stack_chk_fail

buf ← %rsp
```

Input: 1234567
Summary of Prevention Measures

1) Employ system-level protections
   - Code on the Stack is not executable
   - Randomized Stack offsets

2) Avoid overflow vulnerabilities
   - Use library routines that limit string lengths
   - Use a language that makes them impossible

3) Have compiler use “stack canaries”
Think this is cool?

- You’ll love Lab 3 😊
  - Released Wednesday, due next Friday (11/13)
  - Some parts *must* be run through GDB to disable certain security features

- Take CSE 484 (Security)
  - Several different kinds of buffer overflow exploits
  - Many ways to counter them

- Nintendo fun!
  - Using glitches to rewrite code: [https://www.youtube.com/watch?v=TqK-2jUQBUY](https://www.youtube.com/watch?v=TqK-2jUQBUY)
  - Flappy Bird in Mario: [https://www.youtube.com/watch?v=hB6eY73sLV0](https://www.youtube.com/watch?v=hB6eY73sLV0)