Buffer overflows (a security interlude)

Address space layout
the stack discipline
+ C's lack of bounds-checking
HUGE PROBLEM
x86-64 Linux Memory Layout

0x00007fffffffffffffff

not drawn to scale
String Library Code

C standard library 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?

Same problem in many 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;
}
```

```
$ ./bufdemo
Type a string: 1234567
1234567

$ ./bufdemo
Type a string: 12345678
Segmentation Fault

$ ./bufdemo
Type a string: 123456789ABC
Segmentation Fault
```
## Buffer Overflow Disassembly

### echo code

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>4006cf:</td>
<td>48 83 ec 18</td>
<td>sub $0x24,%rsp</td>
</tr>
<tr>
<td>4006d3:</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006d6:</td>
<td>e8 a5 ff ff ff</td>
<td>callq 400680 &lt;gets&gt;</td>
</tr>
<tr>
<td>4006db:</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006de:</td>
<td>e8 3d fe ff ff</td>
<td>callq 400520 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>4006e3:</td>
<td>48 83 c4 18</td>
<td>add $0x24,%rsp</td>
</tr>
<tr>
<td>4006e7:</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>

### caller code

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>4006e8:</td>
<td>48 83 ec 08</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>4006ec:</td>
<td>b8 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>4006f1:</td>
<td>e8 d9 ff ff ff</td>
<td>callq 4006cf &lt;echo&gt;</td>
</tr>
<tr>
<td><strong>4006f6:</strong></td>
<td>48 83 c4 08</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>4006fa:</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

Before call to gets

Stack frame for call_echo

Return address (8 bytes)

20 bytes unused

buf ← %rsp

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

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
Buffer Overflow Stack Example

Before call to gets

Stack frame for call_echo

00 00 00 00
00 40 06 f6

20 bytes unused

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

void echo() {
    char buf[4];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:

    ...
    4006f1: callq 4006cf <echo>
    4006f6: add $0x8,%rsp
    ...

buf ← %rsp
Buffer Overflow Stack Example #1

After call to `gets`

```c
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
subq $24, %rsp
movq %rsp, %rdi
call gets
...
```

```
buf ← %rsp
```

```
$ ./bufdemo
Type a string: 012345678901234567890123456789012
012345678901234567890123456789012
```

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

After call to gets

```c
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

echo:

```
subq $24, %rsp
movq %rsp, %rdi
call gets
...
```

call_echo:

```
.
4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp
.
```

Overflowed buffer and corrupted return pointer
Buffer Overflow Stack Example #3

After call to `gets`

```c
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

echo:
```
subq $24, %rsp
movq %rsp, %rdi
call gets
...
```

call_echo:
```
...
4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp
...
```

```
unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
```

Overflowed buffer, corrupted return pointer, but program seems to work!
Buffer Overflow Stack Example #3 Explained

After call to `gets`

<table>
<thead>
<tr>
<th>Stack frame for call_echo</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>00 40 06 00</td>
<td></td>
</tr>
<tr>
<td>33 32 31 30</td>
<td></td>
</tr>
<tr>
<td>39 38 37 36</td>
<td></td>
</tr>
<tr>
<td>35 34 33 32</td>
<td></td>
</tr>
<tr>
<td>31 30 39 38</td>
<td></td>
</tr>
<tr>
<td>37 36 35 34</td>
<td></td>
</tr>
<tr>
<td>33 32 31 30</td>
<td></td>
</tr>
</tbody>
</table>

```
buf ← %rsp
```

“Returns” to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

Some other place in `.text`

```
400600:  mov %rsp,%rbp
400603:  mov %rax,%rdx
400606:  shr $0x3f,%rdx
40060a:  add %rdx,%rax
40060d:  sar %rax
400610:  jne 400614
400612:  pop %rbp
400613:  retq
```
Malicious Use of Buffer Overflow

Input string contains byte representation of executable code
Overwrite return address A with address of buffer (need to know B)
When `bar()` executes `ret`, will jump to exploit code (instead of A)
Exploiting Buffer Overflows

*Buffer overflow bugs allow remote attackers to execute arbitrary code on machines running vulnerable software.*

**1988: Internet worm**

Early versions of the finger server daemon (fingerd) used `gets()` to read the argument sent by the client:

```bash
finger somebody@cs.wellesley.edu
```

Command line facebook of the 80s!

Attack by sending phony argument:

```bash
finger "exploit-code padding new-return-address"
```

... Still happening

"Ghost:" 2015

getaddrinfo()  
Feb. 2016
Heartbleed (2014)

Buffer over-read in OpenSSL
Widely used encryption library (https)

“Heartbeat” packet
Specifies length of message
Server echoes that much back
Library just “trusted” this length
Allowed attackers to read contents of memory anywhere they wanted

~17% of Internet affected
“Catastrophic”
Github, Yahoo,
Stack Overflow, Amazon AWS, ...

By FenixFeather - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32276981
Avoiding Overrun Vulnerabilities

1. Use a memory-safe language (not C)!

2. If you have to use C, use library functions that limit string lengths.
   \textbf{fgets} instead of \textbf{gets}

   ```c
   /* Echo Line */
   void echo() {
       char buf[4]; /* Way too small! */
       \textbf{fgets} (buf, 4, stdin);
       \textbf{puts} (buf);
   }
   ```

   \textbf{strncpy} instead of \textbf{strcpy}

   Don’t use \textbf{scanf} with \textbf{%s} conversion specification
   Use \textbf{fgets} to read the string
   Or use \textbf{%ns} where \textbf{n} is a suitable integer

\textit{Other ideas?}
Modern System-Level Protections

Available in modern OSs/compilers/hardware
(We disabled these for buffer assignment.)

1. Randomize stack base, maybe frame padding

2. Detect stack corruption
   save and check stack "canary" values

3. Non-executable memory segments
   stack, heap, data, ... everything except text
   hardware support

Helpful, not foolproof!
   Return-oriented programming, over-reads, etc.