Buffer Overrun Exploits

INTERVIEW QUESTION

HOW WOULD YOU DIAGNOSE A BUFFER OVERFLOW PROBLEM?

I'D PUT THE CIRCUIT BOARD IN A BUCKET OF WATER AND LOOK FOR AIR BUBBLES.

THAT SOUNDS RIGHT.

I JUST DIAGNOSED A PROBLEM WITH YOUR INTERVIEW QUESTION.
x86-64 Linux Memory Layout

not drawn to scale
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;
}
```

```bash
$ ./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>Instruction</th>
<th>Notes</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</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</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>Instruction</th>
<th>Notes</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</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>4006f1:</td>
<td>e8 d9 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**

<table>
<thead>
<tr>
<th>Stack frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address (8 bytes)</td>
</tr>
<tr>
<td>20 bytes unused</td>
</tr>
</tbody>
</table>

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

![Assembly code](echo:)

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

buf ← %rsp
Buffer Overflow Stack Example

Before call to \texttt{gets}

\begin{itemize}
\item \texttt{void echo() \{ \}
\item \texttt{char buf[4];}
\item \texttt{gets(buf);}
\item \texttt{\ldots}
\item \texttt{\}}
\end{itemize}

\begin{itemize}
\item \texttt{echo:}
\item \texttt{subq $24, \%rsp}
\item \texttt{movq \%rsp, \%rdi}
\item \texttt{call \ gets}
\item \texttt{\ldots}
\end{itemize}

\texttt{call\_echo:}

\begin{itemize}
\item \texttt{\ldots}
\item \texttt{4006f1: callq 4006cf <echo>}
\item \texttt{4006f6: add $0x8, \%rsp}
\item \texttt{\ldots}
\end{itemize}

\begin{itemize}
\item \texttt{buf \leftarrow \%rsp}
\end{itemize}

Stack frame for \texttt{call\_echo}

\begin{itemize}
\item \texttt{00 00 00 00}
\item \texttt{00 40 06 f6}
\item \texttt{20 bytes unused}
\item \texttt{[3] [2] [1] [0]}
\end{itemize}
After call to `gets`

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

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

Stack frame for `call_echo`

|   00 |   00 |   00 |   00 |
| 00 40 06 | f6 |

Return Address

```assembly
callq 4006cf <echo>
```

`buf` ← %rsp

```assembly
callq 4006cf <echo>
```

```assembly
$ ./bufdemo
Type a string: 01234567890123456789012
01234567890123456789012
```

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

After call to gets

Stack frame for call_echo

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

Return Address

Call site for call_echo:

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

buf ← %rsp

unix> ./bufdemo
Type a string: 0123456789012345678901234

Segmentation Fault

Overflowed buffer and corrupted return pointer
After call to `gets`

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>06</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

Return Address

`buf ← %rsp`

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

After call to `gets`

Stack frame for `call_echo`

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

Return Address

buf ← %rsp

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
```

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

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

`commandline facebook of the 80s (and 90s)!

Attack by sending phony argument:

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

...

Still happening

"Ghost:" 2015

`gethostbyname()`

`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, ...
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.
   - `fgets` instead of `gets`
   - `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

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

Other ideas?
Modern System-Level Protections

Available in modern OSs/compilers/hardware
(We have disabled these for the 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.