Buffer Overflows

Address space layout
the stack discipline
+ C’s lack of bounds-checking
HUGE PROBLEM

String library code

C standard library function `gets()`

```
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '
') {
        *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:123
123

$ ./bufdemo
Type a string:0123456789012345678901234
Segmentation Fault

$ ./bufdemo
Type a string:012345678901234567890123
012345678901234567890123
Vulnerable buffer code: disassembled x86

00000000004006cf <echo>:
4006cf:
   48 83 ec 18 sub $24,%rsp
4006d3:
   48 89 e7 mov %rsp,%rdi
4006d6:
   e8 a5 ff ff ff callq 400680 <gets>
4006db:
   48 89 e7 mov %rsp,%rdi
4006de:
   e8 a5 ff ff ff callq 400520 <puts@plt>
4006e3:
   48 83 c4 18 add $24,%rsp
4006e7:
   c3 retq
4006e8:
   48 83 ec 08 sub $0x8,%rsp
4006ec:
   b8 00 00 00 00 mov $0x0,%eax
4006f1:
   e8 d9 ff ff ff callq 4006cf <echo>
4006f6:
   48 83 c4 08 add $0x8,%rsp
4006fa:
   c3 retq

Buffer overflow example: before input

Before call to gets

Stack frame for call_echo

00 00 00 00
00 40 06 f6
00 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
37 36 35 34
33 32 31 30

Return Address: call_echo:

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

$ ./bufdemo
Type a string: 01234567890123456789012
01234567890123456789012

 Overflowed buffer, but did not corrupt state

Buffer overflow example: input #1

After call to gets

Stack frame for call_echo

00 00 00 00
00 40 06 f6
00 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
37 36 35 34
33 32 31 30

Return Address: call_echo:

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

$ ./bufdemo
Type a string: 01234567890123456789012
01234567890123456789012

 Overflowed buffer, but did not corrupt state

Buffer overflow example: input #2

After call to gets

Stack frame for call_echo

00 00 00 00
00 40 00 00
00 40 06 f6
00 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
37 36 35 34
33 32 31 30

Return Address: call_echo:

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

$ ./bufdemo
Type a string: 01234567890123456789012
01234567890123456789012

Overflowed buffer and corrupted return pointer
Buffer overflow example: input #3

After call to gets

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

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

void call_echo()
```
subq $24, %rsp
movq %rsp, %rdi
```

call Echo:
```
add $0x8, %rsp
```

Stack frame for call_echo
```
00 00 00 00
00 40 06 00
33 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
37 36 35 34
33 32 31 30
```

Return Address
```
buf ← %rsp
```

unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Overflowed buffer, corrupted return pointer, but program seems to work!

Exploiting buffer overflows

```c
int bar() {
    char buf[64];
    gets(buf);
    . . .
    return ...;
}
```

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)

Exploits in the wild

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
```

command line facebook of the 80s!

Attack by sending phony argument:
```
finger “exploit-code padding new-return-address”
```

Still happening

“Ghost” 2015

getaddrinfo() Feb. 2016

gethostname()
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, ...

Optional

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.
   \texttt{fgets} instead of \texttt{gets}

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

\texttt{strncpy} instead of \texttt{strcpy}

Don’t use \texttt{scanf} with \%s conversion specification

Use \texttt{fgets} to read the string
Or use \%n\texttt{s} where \texttt{n} is a suitable integer

Other ideas?

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.