Buffer overflows

- Buffer overflows are possible because C does not check array boundaries
- Buffer overflows are dangerous because buffers for user input are often stored on the stack

Specific topics:
- Address space layout
- Input buffers on the stack
- Overflowing buffers and injecting code
- Defenses against buffer overflows
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E.g., local variables

- **Heap**
  - Dynamically allocated as needed
  - When call malloc, calloc, new, ...

- **Data**
  - Statically allocated data
    - Read-only: string literals
    - Read/write: global arrays and variables

- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only

Hex Address: 00007FFFFFFF

Not drawn to scale
Reminder: x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Arguments (if > 6 args) for this call
  - Return address
    - Pushed by `call` instruction

- **Current/ Callee Stack Frame**
  - Old frame pointer (optional)
  - Saved register context
    (when reusing registers)
  - Local variables
    (If can’t be kept in registers)
  - “Argument build” area
    (If callee needs to call another function - parameters for function about to call, if needed)
Memory Allocation Example

```c
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8);  /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?
x86-64 Example Addresses

address range \( \sim 2^{47} \)

- &local
- p1
- p3
- p4
- p2
- &big_array[0]
- huge_array
- main()
- useless()

What is approximate &p1?
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
Internet Worm

■ These 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

■ November, 1988
  ▪ Internet Worm attacks thousands of Internet hosts.
  ▪ How did it happen?

■ Stack buffer overflow exploits!
Buffer Overflow in a nutshell

- Many classic Unix/Linux/C functions do not check argument sizes
- C does not check array bounds
- Allows overflowing (writing past the end of) buffers (arrays)
- Overflows of buffers on the stack overwrite interesting data
- Attackers just choose the right inputs
- Why a big deal?
  - It is (was?) the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- Simplest form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing
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 \texttt{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
  - \texttt{strcpy}: Copies string of arbitrary length to a dest
  - \texttt{scanf, fscanf, sscanf}, when given \%s conversion specification
Vulnerable Buffer Code

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

void call_echo()
{
    echo();
}

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

unix> ./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault
Buffer Overflow Disassembly

echo:

```
00000000004006cf <echo>:
4006cf: 48 83 ec 18 sub $0x18,%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 3d fe ff ff callq 400520 <puts@plt>
4006e3: 48 83 c4 18 add $0x18,%rsp
4006e7: c3 retq
```

call_echo:

```
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 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

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

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 |

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

unix> ./bufdemo-nsp
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

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

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

unix> ./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault

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

After call to gets

Stack Frame for call_echo

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

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

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>

register_tm_clones:

```
. . .
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
```

buf ← %rsp

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

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

```c
void foo() {
    bar();
    A: ...
}
```
Exploits Based on Buffer Overflows

- **Buffer overflow bugs can allow remote machines 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)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more

- **You will learn some of the tricks in lab 3**
  - Hopefully to convince you to never leave such holes in your programs!!
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 by sending phony argument:
    - `finger "exploit-code padding new-return-address"
      - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

- Once on a machine, scanned for other machines to attack
  - invaded ~6000 computers in hours (10% of the Internet 😊)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted...
What to do about buffer overflow attacks...

1. Avoid overflow vulnerabilities

2. Employ system-level protections

3. Have compiler use “stack canaries”

- Lets talk about each...
1. Avoid Overflow Vulnerabilities in Code (!)

Use library routines that limit string lengths

- **fgets** instead of **gets** (second 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

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```
2. System-Level Protections can help

Randomized stack offsets

- At start of program, allocate *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
- E.g.: 5 executions of memory allocation code from slide 4, address of variable `local` changes each time:
  - 0x7ffe4d3be87c
  - 0x7fff75a4f9fc
  - 0x7ffeadb7c80c
  - 0x7ffeaea2fdac
  - 0x7ffcd452017c
  - Stack repositioned each time program executes

Diagrams show main's stack frame and other functions' stack frames with randomized allocation.
2. System-Level Protections can help

Nonexecutable 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, data, or heap regions
  - Hardware support needed

Any attempt to execute this code will fail
3. Stack Canaries can help

■ Idea
  - Place special value (“canary”) on stack just beyond buffer
    - “After” buffer but before return address
  - Check for corruption before exiting function

■ GCC Implementation
  - `-fstack-protector`
  - Now the default for gcc
  - Code back on slide 12 (`./bufdemo-nsp`) compiled without this option

```
unix>./bufdemo-sp
Type a string: 0123456
0123456

unix>./bufdemo-sp
Type a string: 01234567
*** stack smashing detected ***
```
Protected Buffer Disassembly

echo:

40072f:    sub     $0x18,%rsp
400733:    mov     %fs:0x28,%rax
40073c:    mov     %rax,0x8(%rsp)
400741:    xor     %eax,%eax
400743:    mov     %rsp,%rdi
400746:    callq   4006e0 <gets>
40074b:    mov     %rsp,%rdi
40074e:    callq   400570 <puts@plt>
400753:    mov     0x8(%rsp),%rax
400758:    xor     %fs:0x28,%rax
400761:    je      400768 <echo+0x39>
400763:    callq   400580 __stack_chk_fail@plt>
400768:    add     $0x18,%rsp
40076c:    retq
Setting Up Canary

Before call to gets

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

Before call to gets:

```
28 Buffer Overflow
```

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

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

Input: 0123456

Before call to gets

Return Address
Saved %ebp

Stack Frame for call_echo

Canary (8 bytes)
00 36 35 34
33 32 31 30

buf ← %rsp

Canary (8 bytes)

After call to gets

movq 8(%rsp), %rax # Retrieve from stack
xorq %fs:40, %rax # Compare to canary
je .L6 # If same, OK
call __stack_chk_fail # FAIL
.L6: ...
Summary: Avoiding buffer overflow attacks

1. Avoid overflow vulnerabilities
   - Use library routines that limit string lengths

2. Employ system-level protections
   - Randomized Stack offsets
   - Code on the stack is not executable

3. Have compiler use “stack canaries”