Buffer Overflows
CSE 351 Summer 2019

Instructor: Sam Wolfson
Teaching Assistants: Rehaan Bhimani, Daniel Hsu, Corbin Modica

http://xkcd.com/804/
Administrivia

- Homework 3, due next Monday (7/29)
  - On midterm material, but due after the midterm

- Mid-Quarter Survey, due tonight!
  - Responses are anonymous, but you will receive credit for filling it out

- Midterm (Fri 7/26, 10:50-11:50am in CSE2 G01)
  - Review session: today 7/24, 4-6pm in MOR 220
  - Extra office hours on Thursday!
Buffer Overflows

- Address space layout (more details!)
- 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
  - `malloc()`, `calloc()`, `new`, ...

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

- **Code/Instructions**
  - Executable machine instructions
  - Read-only
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack has 8 MiB limit

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

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

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

Hex Address: 0x400000

Stack

Heap

Shared Libraries

Heap

Data

Instructions
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?
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?
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)
  - 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)
Buffer Overflow in a Nutshell

- 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

- 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 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 is (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 \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 \texttt{limit} on number of characters to read

- Similar problems with other Unix functions:
  - \texttt{strcpy}: Copies string of arbitrary length to a dst
  - \texttt{scanf, fscanf, sscanf, when given} \texttt{\%s} \texttt{specifier}
Vulnerable Buffer Code

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

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

```
unix> ./buf-nsp
Enter string: 123456789012345678901234567890123
Illegal instruction
```

```
unix> ./buf-nsp
Enter string: 123456789012345678901234567890123
```
Buffer Overflow Disassembly (buf-nsp)

**echo:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000400597 &lt;echo&gt;:</td>
<td>sub $0x18,%rsp</td>
</tr>
<tr>
<td>400597:</td>
<td>48 83 ec 18</td>
</tr>
<tr>
<td>...</td>
<td>... calls printf ...</td>
</tr>
<tr>
<td>4005aa:</td>
<td>48 89 e7</td>
</tr>
<tr>
<td>4005af:</td>
<td>e8 dd fe ff ff</td>
</tr>
<tr>
<td>4005b4:</td>
<td>48 89 e7</td>
</tr>
<tr>
<td>4005b9:</td>
<td>e8 95 fe ff ff</td>
</tr>
<tr>
<td>4005be:</td>
<td>48 83 c4 18</td>
</tr>
<tr>
<td>4005c2:</td>
<td>c3</td>
</tr>
</tbody>
</table>

**call_echo:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000004005c3 &lt;call_echo&gt;:</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>4005c3:</td>
<td>48 83 ec 08</td>
</tr>
<tr>
<td>4005c7:</td>
<td>b8 00 00 00 00 00</td>
</tr>
<tr>
<td>4005cc:</td>
<td>e8 ca ff ff ff</td>
</tr>
<tr>
<td>4005d1:</td>
<td>48 83 c4 08</td>
</tr>
<tr>
<td>4006d5:</td>
<td>c3</td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

Before call to gets

Stack frame for call_echo

Return address (8 bytes)

8 bytes unused

[3] [2] [1] [0]
8 bytes unused

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

echo:
    subq $24, %rsp
    ...
    movq 8(%rsp), %rdi
    call gets
    ...

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

Before call to gets

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

call_echo:

... 
4005cc: callq 400597 <echo>
4005d1: add $0x8,%rsp 
...
Buffer Overflow Example #1

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>05</td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

call_echo:
  ...
  4005cc: callq 400597 <echo>
  4005d1: add $0x8,%rsp
  ...

```
unix> ./buf-nsp
Enter string: 1234567890123456789012345
1234567890123456789012345
```

Overflowed buffer, but did not corrupt state
Buffer Overflow Example #1

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>05</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>30</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

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

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

call `_echo`:

```asm
. . .
4005cc: callq 4005c6 <echo>
4005d1: add $0x8,%rsp
. . .
```

`unix>` `./buf-nsp`

Enter string: 1234567890123456

Illegal instruction

Overflowed buffer and corrupted return pointer
Buffer Overflow Example #2 Explained

After return from echo

Stack frame for call_echo

| 00 | 00 | 00 | 00 |
| 00 | 40 | 05 | 00 |
| 36 | 35 | 34 | 33 |
| 32 | 31 | 30 | 39 |
| 38 | 37 | 36 | 35 |
| 34 | 33 | 32 | 31 |

%rsp

000000000004004f0 <deregister_tm_clones>:
4004f0: push %rbp
4004f1: mov $0x601040,%eax
4004f6: cmp $0x601040,%rax
4004fc: mov %rsp,%rbp
4004ff: je 400518
400501: mov $0x0,%eax
400506: test %rax,%rax
400509: je 400518
40050b: pop %rbp
40050c: mov $0x601040,%edi
400511: jmpq *%rax
400513: nopl 0x0(%rax,%rax,1)
400518: pop %rbp
400519: retq
40051a: nopw 0x0(%rax,%rax,1)

Tries to “return” to middle of unrelated code!
“Illegal instruction” error occurs because 0x400500 is in the middle of an instruction.
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 \texttt{bar()} executes \texttt{ret}, will jump to exploit code

```c
void foo() {
    bar();
    A: ...
}

int bar() {
    char buf[64];
    gets(buf);
    ...
    return ...;
}
```
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)
  - *Still happens!!*
    - Heartbleed (2014, affected 17% of servers)
    - Cloudbleed (2017)
  - **Fun**: Nintendo hacks
    - Using glitches to rewrite code: [https://www.youtube.com/watch?v=TqK-2jUQBUY](https://www.youtube.com/watch?v=TqK-2jUQBUY)
    - FlappyBird in Mario: [https://www.youtube.com/watch?v=hB6eY73sLV0](https://www.youtube.com/watch?v=hB6eY73sLV0)
    - Arbitrary code execution in Zelda: [https://www.youtube.com/watch?v=fj9u00PMkYU](https://www.youtube.com/watch?v=fj9u00PMkYU)
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 TCP connection to the attacker
- 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...
Heartbleed (2014)

- 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, ...

Heartbeat – Normal usage

- Client
  - Server, send me this 4 letter word if you are there: "bird"

Heartbeat – Malicious usage

- Client
  - Server, send me this 500 letter word if you are there: "bird"
  - bird. Server master key is 31431498531054. User Carol wants to change password to "password123"...

By FenixFeather - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32276981
Dealing with buffer overflow attacks

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

- **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

- **Example:** Code from Slide 6 executed 5 times; address of variable `local`
  - `0x7fffd19d3f8ac`
  - `0x7ffe8a462c2c`
  - `0x7ffe927c905c`
  - `0x7ffefd5c27dc`
  - `0x7fffa0175afc`

- Stack repositioned each time program executes
2) 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.
3) Stack Canaries

- Basic Idea: place special value ("canary") on stack just beyond buffer
  - Secret value known only to compiler
  - "After" buffer but before return address
  - Check for corruption before exiting function

- GCC implementation (now default)
  - -fstack-protector
  - Code back on Slide 14 (buf-nsp) compiled with
    -fno-stack-protector flag

```
unix> ./buf
Enter string: 12345678
12345678
unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```
Protected Buffer Disassembly (buf)

```assembly
echo:

400638:  sub    $0x18,%rsp
40063c:  mov    %fs:0x28,%rax
400645:  mov    %rax,0x8(%rsp)
40064a:  xor    %eax,%eax
        ...    ... call printf ...
400656:  mov    %rsp,%rdi
400659:  callq  400510 <gets@plt>
40065e:  mov    %rsp,%rdi
400661:  callq  4004d0 <puts@plt>
400666:  mov    0x8(%rsp),%rax
40066b:  xor    %fs:0x28,%rax
400674:  je     40067b <echo+0x43>
400676:  callq  4004f0 <__stack_chk_fail@plt>
40067b:  add    $0x18,%rsp
40067f:  retq
```

This is extra (non-testable) material
Setting Up Canary

*Before call to gets*

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

![Stack frame for call_echo](image)

- Return address (8 bytes)
- Stack frame for call_echo
- Canary (8 bytes)

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

buf ←%rsp
Checking Canary

After call to gets

Stack frame for call_echo

Return address (8 bytes)

Canary (8 bytes)

Input: 1234567
Summary

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”
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structs**
  - Alignment

- **Unions**
Structure Representation

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering would be more compact
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```
Review: Memory Alignment in x86-64

*Aligned* means that any primitive object of $K$ bytes must have an address that is a multiple of $K$.

Aligned addresses for data types:

<table>
<thead>
<tr>
<th>$K$</th>
<th>Type</th>
<th>Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
<td>No restrictions</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
<td>Lowest bit must be zero: $...0_2$</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
<td>Lowest 2 bits zero: $...00_2$</td>
</tr>
<tr>
<td>8</td>
<td>long, double, *</td>
<td>Lowest 3 bits zero: $...000_2$</td>
</tr>
<tr>
<td>16</td>
<td>long double</td>
<td>Lowest 4 bits zero: $...0000_2$</td>
</tr>
</tbody>
</table>
Alignment Principles

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on x86-64

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of bytes (width is system dependent)
    - Inefficient to load or store value that spans quad word boundaries
    - Virtual memory trickier when value spans 2 pages (more on this later)
  - Though x86-64 hardware will work regardless of alignment of data
Structures & Alignment

- **Unaligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Satisfying Alignment with Structures (1)

- **Within structure:**
  - Must satisfy each element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement $K_{\text{max}}$
    - $K_{\text{max}} = \text{Largest alignment of any element}$
    - Counts array elements individually as elements

- **Example:**
  - $K_{\text{max}} = 8$, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Satisfying Alignment with Structures (2)

- Can find offset of individual fields using `offsetof()`
  - Need to `#include <stddef.h>`
  - Example: `offsetof(struct S2, c)` returns 16

- For largest alignment requirement $K_{max}$, overall structure size must be multiple of $K_{max}$
  - Compiler will add padding at end of structure to meet overall structure alignment requirement

```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
Arrays of Structures

- Overall structure length multiple of $K_{max}$
- Satisfy alignment requirement for every element in array

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Alignment of Structs

- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each *field* must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be *aligned* according to largest field
  - Total struct *size* must be multiple of its alignment *(may insert padding)*
    - `sizeof` should be used to get true size of structs
How the Programmer Can Save Space

- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>i</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 bytes</td>
<td></td>
<td></td>
<td>3 bytes</td>
</tr>
</tbody>
</table>

12 bytes

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 bytes
Peer Instruction Question

Which of the following statements is FALSE?

Vote at http://pollev.com/wolfson

A. `sea[4][-2]` is a valid array reference
B. `sea[1][1]` makes two memory accesses
C. `sea[2][1]` will always be a higher address than `sea[1][2]`
D. `sea[2]` is calculated using only 1 `lea`
E. We’re lost...