Security: Buffer Overflow

CS 105: Computer Systems

Guest Lecturer: Beth Trushkowsky

February 22, 2022

Learning Goals

- Understand what a buffer overflow is and how it can happen
- See how the runtime stack can be exploited to run malicious code
- Practice writing an exploit
- Discuss techniques to address buffer overflow attacks

Exercise: memory layout of a double

Recall the data type double uses 8 bytes, as shown above. Suppose we have: double pi = 3.14;
In hex, the value of the variable pi is 0x40091eb851eb851f

1. Underline which hex digits encode the fractional part of the mantissa.
   0x40 09 1e b8 51 eb 85 1f

2. If &pi is 0x100, what should be the one-byte content (in hex) at memory address 0x102 on a little endian machine?
   0xeb
x86-64 Memory Layout

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

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

- **Data**
  - Statically allocated data
  - E.g., global vars, static vars, string constants

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

---

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?

Hex Address: 400000

---

Memory Referencing Bug Example

```c
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    struct_t s;
    s.d = 3.14;
    s.a[i] = 0x40000000; /* Possibly out of bounds */
    return s.d;
}
```

```c
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14
fun(5) → 3.14
fun(6) → Segmentation fault
```

Assume each row in the stack diagram is 8 bytes
- Addresses increase from bottom to top
- Addresses increase from right to left within a row

Note that `s` requires 16 bytes, as shown. Indicate where in the diagram `s.a[0]`, `s.a[1]`, and `s.d` are located.
- Recall an int is 4 bytes and a double is 8 bytes

---

Exercise: Memory Referencing Bug Example

```c
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    struct_t s;
    s.d = 3.14;
    s.a[i] = 0x40000000;
    return s.d;
}
```

```c
stack

s
s.d
s.a[1] s.a[0]
```
Memory Referencing Bug: Explanation

typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    struct_t s;
    s.d = 3.14; /* 0x40091eb851eb851f */
    s.a[i] = 0x40000000;
    return s.d;
}

fun(0) ➞ 3.14
fun(1) ➞ 3.14
fun(2) ➞ 3.1399998664856
fun(3) ➞ 2.00000061035156
fun(4) ➞ 3.14
fun(5) ➞ 3.14
fun(6) ➞ Segmentation fault

What sort of critical state could be here?

<table>
<thead>
<tr>
<th>Critical state</th>
<th>Overwritten by fun(6) and fun(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.d</td>
<td>Overwritten by fun(4) and fun(5)</td>
</tr>
<tr>
<td>s.a[1]</td>
<td>Overwritten by fun(2) and fun(3)</td>
</tr>
</tbody>
</table>

Buffer Overflow

- Exceeding memory size allocated for an array
  - Generally called a “buffer overflow” aka “stack smashing”

- Why is it a big deal? Causes a lot of security vulnerabilities!

Morris Worm

- Nov. 2, 1988 -- Cornell grad student Robert Morris (somewhat unintentionally) creates first internet worm
  - Affected about a tenth of computers on the Internet at the time
  - Morris fined $10,050, 400 hours community service, and 3 years probation

- Robert Morris now a professor at MIT...

- Part of his approach was a buffer overflow attack!
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's the limit on characters that are read?

Similar problems with other library functions
- `strcpy`, `strcat`: Copy strings of arbitrary length
- `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Running example using `gets`

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

Example: calling `echo`

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

Example: instruction `sub` in `echo`

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

Allocate space on stack for `buf`

Why 24 bytes? Often more space is allocated than is actually needed because of data alignment requirements.
Example: preparing to call gets (in echo)

```c
void call_echo() {
  void echo();
  /* Echo Line */
}
```

Example: in gets, reading first character

```c
void call_echo() {
  /* Echo Line */
}
```

Example: Calling gets (in echo)

```c
void call_echo() {
  void echo();
  /* Echo Line */
}
```

Example: in gets, read string length 23

```c
void call_echo() {
  /* Echo Line */
}
```
**Example: in `gets`, read string length 25**

```c
void echo() {
    puts(buf); // Echo Line
}
```

---

**Example: In echo after `gets` read 25 and `puts` returns**

```c
void call_echo() {
    callq 4004cf <echo> retq
}
```

---

**Example: Returning (from `echo`, `gets` read 25)**

```c
void echo() {

    /* Echo Line */
    void echo() {
        char buf[4];
        puts(buf);
    }

    gets(buf);
}
```

---

**Example: What instruction gets executed?**

- Valid machine instruction
- Not a valid machine instruction
- Continue execution
- Segmentation fault

**AND THEN ...**

---

Adapted from Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition
### Code Injection Attacks

**Stack before call to `gets()`**

```c
void P() {
    Q();
    ...
}
```

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

- Return address A
- Return address B
- Q stack frame
- P stack frame
- buf

**Stack after call to `gets()`**

- Return address A
- Return address B
- Q stack frame
- P stack frame
- buf
- padding
- exploit code

- Input string contains byte representation of executable code
- Overwrite return address A with address of `buf` array

What happens when Q returns?

---

### 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
- You will learn some of the tricks in Attack Lab
  - Hopefully to convince you to never leave such holes in your programs!!
- **Prevention techniques**
  1. Avoid overflow vulnerabilities
  2. Employ system-level protections
  3. Have compiler use “stack canaries”

---

### 1. Avoid Overflow Vulnerabilities in Code (!)

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

- For example, use library routines 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
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 so address of buffer is not known
  - Makes it difficult for hacker to determine address of inserted code

- Non-executable code segments
  - In previous x86, could mark region of memory as either "read-only" or "writeable" and could execute anything readable
  - X86-64 added explicit "execute" permission
  - Stack marked as non-executable

Any attempt to execute this code will fail

3. Stack Canaries can help

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function

- GCC Implementation
  - -fstack-protector
  - Now the default (disabled earlier)

Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Exercise

Assume your computer uses ASCII encoding for strings and that the ASCII for the string “BANG” is also a machine instruction that makes your computer explode. Come up with an input to echo that makes your computer explode. You can assume the system knows how many bytes the “BANG” instruction is after it reads the first byte corresponding to “B”.

1. Show the stack (use hex values) after the call to gets. An ASCII table is below.

2. Write the text input string here: BANG012345678901234567890@

## ASCII Table

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>42</td>
<td>0010101</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
<td>0010100</td>
</tr>
<tr>
<td>N</td>
<td>4E</td>
<td>00101110</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
<td>00100111</td>
</tr>
</tbody>
</table>

## Stack after gets

Return address

Before call to gets

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0010101</td>
</tr>
<tr>
<td>01</td>
<td>0010100</td>
</tr>
<tr>
<td>02</td>
<td>0010100</td>
</tr>
<tr>
<td>03</td>
<td>0010100</td>
</tr>
<tr>
<td>04</td>
<td>0010100</td>
</tr>
<tr>
<td>05</td>
<td>0010100</td>
</tr>
<tr>
<td>06</td>
<td>0010100</td>
</tr>
<tr>
<td>07</td>
<td>0010100</td>
</tr>
<tr>
<td>08</td>
<td>0010100</td>
</tr>
<tr>
<td>09</td>
<td>0010100</td>
</tr>
<tr>
<td>10</td>
<td>0010100</td>
</tr>
<tr>
<td>11</td>
<td>0010100</td>
</tr>
<tr>
<td>12</td>
<td>0010100</td>
</tr>
<tr>
<td>13</td>
<td>0010100</td>
</tr>
<tr>
<td>14</td>
<td>0010100</td>
</tr>
<tr>
<td>15</td>
<td>0010100</td>
</tr>
<tr>
<td>16</td>
<td>0010100</td>
</tr>
<tr>
<td>17</td>
<td>0010100</td>
</tr>
<tr>
<td>18</td>
<td>0010100</td>
</tr>
<tr>
<td>19</td>
<td>0010100</td>
</tr>
<tr>
<td>20</td>
<td>0010100</td>
</tr>
<tr>
<td>21</td>
<td>0010100</td>
</tr>
<tr>
<td>22</td>
<td>0010100</td>
</tr>
<tr>
<td>23</td>
<td>0010100</td>
</tr>
<tr>
<td>24</td>
<td>0010100</td>
</tr>
<tr>
<td>25</td>
<td>0010100</td>
</tr>
<tr>
<td>26</td>
<td>0010100</td>
</tr>
<tr>
<td>27</td>
<td>0010100</td>
</tr>
<tr>
<td>28</td>
<td>0010100</td>
</tr>
<tr>
<td>29</td>
<td>0010100</td>
</tr>
<tr>
<td>30</td>
<td>0010100</td>
</tr>
</tbody>
</table>

## Stack after call to echo

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>0010100</td>
</tr>
<tr>
<td>34</td>
<td>0010100</td>
</tr>
<tr>
<td>35</td>
<td>0010100</td>
</tr>
<tr>
<td>36</td>
<td>0010100</td>
</tr>
<tr>
<td>37</td>
<td>0010100</td>
</tr>
<tr>
<td>38</td>
<td>0010100</td>
</tr>
<tr>
<td>39</td>
<td>0010100</td>
</tr>
<tr>
<td>40</td>
<td>0010100</td>
</tr>
<tr>
<td>41</td>
<td>0010100</td>
</tr>
<tr>
<td>42</td>
<td>0010100</td>
</tr>
<tr>
<td>43</td>
<td>0010100</td>
</tr>
<tr>
<td>44</td>
<td>0010100</td>
</tr>
<tr>
<td>45</td>
<td>0010100</td>
</tr>
<tr>
<td>46</td>
<td>0010100</td>
</tr>
<tr>
<td>47</td>
<td>0010100</td>
</tr>
<tr>
<td>48</td>
<td>0010100</td>
</tr>
<tr>
<td>49</td>
<td>0010100</td>
</tr>
<tr>
<td>50</td>
<td>0010100</td>
</tr>
<tr>
<td>51</td>
<td>0010100</td>
</tr>
<tr>
<td>52</td>
<td>0010100</td>
</tr>
<tr>
<td>53</td>
<td>0010100</td>
</tr>
<tr>
<td>54</td>
<td>0010100</td>
</tr>
<tr>
<td>55</td>
<td>0010100</td>
</tr>
<tr>
<td>56</td>
<td>0010100</td>
</tr>
<tr>
<td>57</td>
<td>0010100</td>
</tr>
<tr>
<td>58</td>
<td>0010100</td>
</tr>
<tr>
<td>59</td>
<td>0010100</td>
</tr>
<tr>
<td>60</td>
<td>0010100</td>
</tr>
<tr>
<td>61</td>
<td>0010100</td>
</tr>
<tr>
<td>62</td>
<td>0010100</td>
</tr>
<tr>
<td>63</td>
<td>0010100</td>
</tr>
</tbody>
</table>

Source: www.LookupTables.com