Lecture 11: Security

January 25, 2020

Chris Stone

Lab 3 (Bomb) Due 1:15pm Friday
Lab 4 (Attack) Starts 1:15pm Friday
Take-Home Midterm available by 5pm Friday Afternoon
(75-minute exam due 5pm the following Friday)

Recommended textbook reading (ASAP): 3.9, 3.10
Good review problems: 3.46-3.48
Textbook preview for Thursday: Review through 3.10
But first: Unions!
Unions

Like structs, but all the fields all overlap.

```c
struct {
    char c;
    int i[2];
    double v;
} *sp;

union {
    char c;
    int i[2];
    double v;
} *up;
```
Example: Lexing/Tokenizing Strings

```c
int count = f ( 3, ++i ) ;
```

- the type "int"
- the name "count"
- the name "f"
- the number 3
- etc.

...
int lex(inputString) {
    if (...inputString starts with an integer...) {
        yylval.n = ...the value of that integer...;
        return 256;
    } else if (...inputString starts with the = operator...) {
        return '=';
    } else if (...inputString starts with the ++ operator...) {
        return 257;
    } else if (...inputString starts with a variable name...) {
        yylval.sp = ...pointer to that variable name...;
        return 258;
    } else if (...inputString starts with a type name...) {
        yylval.sp = ...pointer to the type name...;
        return 259;
    } else ...
}

union {
    int n;
    double d;
    char* sp;
} yylval;

We'll see a more convincing application when we reach networking!
Using Unions To Peek at Byte Ordering

```c
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td></td>
<td>i[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

larger addresses →
Endianness Revealed!

union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;

Sun/Internet (big-endian):

i[0] = 0xf0f1f2f3
l[0] = 0xf0f1f2f3f4f4f6f7

Intel/ARM (little-endian):

i[0] = 0xf3f2f1f0
l[0] = 0xf7f6f5f4f3f2f1f0
Memory and Memory Safety
Learning Goals

• Understand what a buffer overflow is and how it can happen
• See how the stack can be exploited to run malicious code
• Practice writing an exploit
• Discuss techniques to address buffer overflow attacks
x86-64 Linux 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 bigger_array[1L<<30]; /* 1 GB */

int global = 0;

int useless() { return 0; }

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

Where does everything go?
**x86-64 Example Addresses**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;local</td>
<td>0x00007fffffffffe504</td>
</tr>
<tr>
<td>p1</td>
<td>0x00007ffe7e26010</td>
</tr>
<tr>
<td>p3</td>
<td>0x00007ffee7e25010</td>
</tr>
<tr>
<td>p4</td>
<td>0x000000000414053b0</td>
</tr>
<tr>
<td>p2</td>
<td>0x000000000414052a0</td>
</tr>
<tr>
<td>&amp;big_array</td>
<td>0x0000000004040460</td>
</tr>
<tr>
<td>&amp;bigger_array</td>
<td>0x0000000000040460</td>
</tr>
<tr>
<td>&amp;global</td>
<td>0x0000000000004044</td>
</tr>
<tr>
<td>&amp;main()</td>
<td>0x000000000000401125</td>
</tr>
<tr>
<td>&amp;useless()</td>
<td>0x00000000000040111a</td>
</tr>
</tbody>
</table>

Note: very much not to scale!
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;
}

fun(0) → 3.14 0x40091eb51eb51f
fun(1) → 3.14 0x40091eb51eb51f
fun(2) → 3.1399998664856 0x40091eb840000000
fun(3) → 2.00000061035156 0x40000000051eb51f
fun(4) → 3.14 0x40091eb51eb51f
fun(5) → 3.14 0x40091eb51eb51f
fun(6) → Segmentation fault
Thinking About the Crash

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 (little endian)

If s is located as shown, where are s.a[0], s.a[1], and s.d?

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;
}
Buffer Overflow

Exceeding memory size allocated for an array
  • Generally called a "buffer overflow"
  • If the array is on the stack, "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, given 400 hours community service, and 3 years probation
• Robert Morris now a professor at MIT…

• Part of his approach was a buffer overflow attack!
Morris Worm

The Morris Internet Worm source code

This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.

The worm was the first of many intrusive programs that use the Internet to spread.
The implementation of Unix gets() function

What's the problem here?

```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;
}
```

• Similar problems with other library functions
  • strcpy, strcat: Copy strings of arbitrary length
  • scanf, fscanf, sscanf, when given %s conversion specification
Vulnerable Code (Running Example)

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

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

☞ BTW, how big is big enough?

```bash
unix> ./bufdemo
Type a string:012345678901234567890123
012345678901234567890123

unix> ./bufdemo
Type a string:0123456789012345678901234
Segmentation Fault
```
Normal Run: 3 characters

void call_echo() {
    echo();
}

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

unix> ./bufdemo-nsp
Type a string:012
012

ascii of 0 is 0x30
void call_echo() {
    echo();
}

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

void call_echo() {
    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

unix> ./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
ascii of 0 is 0x30

Risky Run: 23 characters
/** Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}

void call_echo()
{
    echo();
}

#include <stdio.h>

int main()
{
    void *buf = malloc(4);
   gets(buf);
    puts(buf);
    free(buf);
    return 0;
}
The program crashed because the code "returned" (jumped) to address 0x400034, which didn't contain valid machine code.

And by typing in a carefully-chosen 32-character string, we can make echo() "return" (jump) to any address we want!
**Code Injection Attacks**

Input string includes bytes encoding machine code

Overwrite return address A with address of that code!

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

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

What happens when Q returns?
Exercise

Assume 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.

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

2. Write the text input string here:

```
BANG-**********-...@?@
```

Before call to `gets`

<table>
<thead>
<tr>
<th>Return address</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>40</th>
<th>06</th>
<th>fa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After call to `gets`

<table>
<thead>
<tr>
<th>Return address</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>40</th>
<th>38</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2A</td>
<td>2A</td>
<td>2A</td>
<td>2A</td>
<td>2A</td>
<td>47</td>
<td>4E</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>2A</td>
<td>2A</td>
<td>2A</td>
<td>2A</td>
<td>2A</td>
<td>47</td>
<td>4E</td>
<td>41</td>
</tr>
</tbody>
</table>

\&buf = 0x403f30
<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
<th>Description</th>
<th>Code</th>
<th>Description</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>000</td>
<td>(null)</td>
<td>040</td>
<td>Space</td>
<td>100</td>
<td>Space</td>
</tr>
<tr>
<td>SOH</td>
<td>001</td>
<td>(start of heading)</td>
<td>041</td>
<td>!</td>
<td>101</td>
<td>!</td>
</tr>
<tr>
<td>STX</td>
<td>002</td>
<td>(start of text)</td>
<td>042</td>
<td>&quot;</td>
<td>102</td>
<td>&quot;</td>
</tr>
<tr>
<td>ETX</td>
<td>003</td>
<td>(end of text)</td>
<td>043</td>
<td>#</td>
<td>103</td>
<td>#</td>
</tr>
<tr>
<td>EOT</td>
<td>004</td>
<td>(end of transmission)</td>
<td>044</td>
<td>$</td>
<td>104</td>
<td>$</td>
</tr>
<tr>
<td>ENQ</td>
<td>005</td>
<td>(enquiry)</td>
<td>045</td>
<td>%</td>
<td>105</td>
<td>%</td>
</tr>
<tr>
<td>ACK</td>
<td>006</td>
<td>(acknowledge)</td>
<td>046</td>
<td>&amp;</td>
<td>106</td>
<td>&amp;</td>
</tr>
<tr>
<td>BS</td>
<td>010</td>
<td>(backspace)</td>
<td>050</td>
<td>(</td>
<td>110</td>
<td>(</td>
</tr>
<tr>
<td>TAB</td>
<td>011</td>
<td>(horizontal tab)</td>
<td>051</td>
<td>)</td>
<td>111</td>
<td>)</td>
</tr>
<tr>
<td>LF</td>
<td>012</td>
<td>(NL line feed, new line)</td>
<td>052</td>
<td>*</td>
<td>112</td>
<td>*</td>
</tr>
<tr>
<td>VT</td>
<td>013</td>
<td>(vertical tab)</td>
<td>053</td>
<td>+</td>
<td>113</td>
<td>+</td>
</tr>
<tr>
<td>FF</td>
<td>014</td>
<td>(NP form feed, new page)</td>
<td>054</td>
<td>,</td>
<td>114</td>
<td>,</td>
</tr>
<tr>
<td>CR</td>
<td>015</td>
<td>(carriage return)</td>
<td>055</td>
<td>-</td>
<td>115</td>
<td>-</td>
</tr>
<tr>
<td>SO</td>
<td>016</td>
<td>(shift out)</td>
<td>056</td>
<td>.</td>
<td>116</td>
<td>.</td>
</tr>
<tr>
<td>SI</td>
<td>017</td>
<td>(shift in)</td>
<td>057</td>
<td>/</td>
<td>117</td>
<td>/</td>
</tr>
<tr>
<td>DLE</td>
<td>020</td>
<td>(data link escape)</td>
<td>060</td>
<td>0</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>DC1</td>
<td>021</td>
<td>(device control 1)</td>
<td>061</td>
<td>1</td>
<td>121</td>
<td>1</td>
</tr>
<tr>
<td>DC2</td>
<td>022</td>
<td>(device control 2)</td>
<td>062</td>
<td>2</td>
<td>122</td>
<td>2</td>
</tr>
<tr>
<td>DC3</td>
<td>023</td>
<td>(device control 3)</td>
<td>063</td>
<td>3</td>
<td>123</td>
<td>3</td>
</tr>
<tr>
<td>DC4</td>
<td>024</td>
<td>(device control 4)</td>
<td>064</td>
<td>4</td>
<td>124</td>
<td>4</td>
</tr>
<tr>
<td>MAK</td>
<td>025</td>
<td>(negative acknowledge)</td>
<td>065</td>
<td>5</td>
<td>125</td>
<td>5</td>
</tr>
<tr>
<td>SYN</td>
<td>026</td>
<td>(synchronous idle)</td>
<td>066</td>
<td>6</td>
<td>126</td>
<td>6</td>
</tr>
<tr>
<td>ETB</td>
<td>027</td>
<td>(end of trans. block)</td>
<td>067</td>
<td>7</td>
<td>127</td>
<td>7</td>
</tr>
<tr>
<td>CAN</td>
<td>030</td>
<td>(cancel)</td>
<td>068</td>
<td>8</td>
<td>130</td>
<td>8</td>
</tr>
<tr>
<td>EM</td>
<td>031</td>
<td>(end of medium)</td>
<td>071</td>
<td>9</td>
<td>131</td>
<td>9</td>
</tr>
<tr>
<td>SUB</td>
<td>032</td>
<td>(substitute)</td>
<td>072</td>
<td>10</td>
<td>132</td>
<td>10</td>
</tr>
<tr>
<td>ESC</td>
<td>033</td>
<td>(escape)</td>
<td>073</td>
<td>11</td>
<td>133</td>
<td>11</td>
</tr>
<tr>
<td>FS</td>
<td>034</td>
<td>(file separator)</td>
<td>074</td>
<td>12</td>
<td>134</td>
<td>12</td>
</tr>
<tr>
<td>GS</td>
<td>035</td>
<td>(group separator)</td>
<td>075</td>
<td>13</td>
<td>135</td>
<td>13</td>
</tr>
<tr>
<td>RS</td>
<td>036</td>
<td>(record separator)</td>
<td>076</td>
<td>14</td>
<td>136</td>
<td>14</td>
</tr>
<tr>
<td>US</td>
<td>037</td>
<td>(unit separator)</td>
<td>077</td>
<td>15</td>
<td>137</td>
<td>15</td>
</tr>
</tbody>
</table>

Source: www.LookupTables.com
Exercise

Assume 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.

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?@

\[
\begin{array}{cccccccc}
\text{Before call to gets} & & & & & & & \\
\text{Rest of stack frame} & & & & & & & \\
\text{for call\_echo} & & & & & & & \\
\hline
& 00 & 00 & 00 & 00 & 00 & 40 & 06 & \text{fa} & \\
\hline
\hline
\text{Return} & & & & & & & & & \\
\hline
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{After call to gets} & & & & & & & \\
\text{Rest of stack Frame} & & & & & & & \\
\text{for call\_echo} & & & & & & & \\
\hline
& & & & & & & & \\
\hline
& 00 & 40 & 3f & 30 & & & & & \\
\hline
& 39 & 38 & 37 & 36 & 35 & 34 & 33 & 32 & \\
\hline
& 31 & 30 & 39 & 38 & 37 & 36 & 35 & 34 & \\
\hline
& 33 & 32 & 31 & 30 & 47 & 4e & 41 & 42 & \\
\hline
\end{array}
\]

&buf = 0x403f30
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:
  • Avoid overflow vulnerabilities
  • Employ system-level protections
  • Have compiler use “stack canaries”
1. Avoid Overflow Vulnerabilities in Code (!)

- For example, use library routines that limit string lengths
  - fgets instead of gets
  - strncpy instead of strcpy
- E.g., see "Secure Programming in C and C++" (linked on Piazza)

```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 so address of buffer is not known
  - Makes it difficult for hacker to determine address of inserted code

![Diagram showing stack layout with randomized allocation]

- Stack base
- Random allocation
- Application Code
- B?
- pad
- exploit code
- B?
2. System-Level Protections can help

- Non-executable code segments
  - In previous x86, could mark region of memory as either “read-only” or “writeable”… 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, 
    *e.g.*, in your bomb machine code.

```bash
unix>./bufdemo-sp
Type a string:0123456
0123456
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```
Canary-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>
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
```

- **Put canary on stack**
- **Check canary on stack**
- **Detect buffer overflow**