Announcement 2/24/2016

- Bomb Lab due 11:55pm Friday 2/26/2016
- First Mid-term Exam next Monday 2/29/2016 in class
  - Coverage Chapt 1 – Chapt 3
  - Open book, open notes
  - No e-devices
- How to prepare for the exams?
  - Go over all lecture slides, and make sure you understand all of the materials covered there.
  - UNITE has recorded lecture videos for review
    https://www.unite.umn.edu/

Outline

- Memory Layout
- Buffer Overflow
- Vulnerability
- Protection

Review: x86-64/Linux Stack Frame

```plaintext
Current Stack Frame ("Top" to Bottom)
- "Argument build:
  - Parameters for function about to call
- Local variables
  - If can’t keep in registers
- Saved register context
  - Old frame pointer (optional)
- Caller Stack Frame (ABI – Application Binary Interface)
  - Return address
    - Pushed by call instruction
  - Arguments for this call
```

Memory Allocation Example

```plaintext
char big_array[1L<<24]; /* 16 MB global var */
char huge_array[1L<<31]; /* 2 GB global var */
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 */
}
```
x86-64 Example Addresses

```
local
p1 0x00007f7524a6e100
p2 0x00007f7524a6d010
big_array 0x0000000080601060
huge_array 0x0000000000601060
main() 0x000000000040060c
useless() 0x0000000000400590
```

Outline

• Memory Layout
  • Buffer Overflow
    • Vulnerability
    • Protection

Such problems are a BIG deal

• Generally called a “buffer overflow”
  • When exceeding the memory size allocated for an array

• Why a big deal?
  • It’s the #1 technical cause of security vulnerabilities
    • #1 overall cause is social engineering / user ignorance

• Most common form
  • Unchecked lengths on string inputs
    • Particularly for bounded character arrays on the stack
      • Sometimes referred to as stack smashing

Memory Referencing Bug Example

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

double fun(int i) {
  volatile struct_t *s;
  s.d = 3.14;
  s.a[i] = 1073741824;
  return s.d;
}
```

Result is system specific

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Critical State</td>
<td>Location accessed by <code>fun(1)</code></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>d7 ... d4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>d3 ... d0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>s[1]</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>s[0]</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

String Library Code

```
#define gets(str) +++
char *gets(char *dest)
{
  int c = getchar();
  char *p = dest;
  while (c != EOF && c != '\n') {
    if (c == '\n')
      c = getchar();
    *p++ = c;
  }
  *p = '\0';
  return dest;
}
```

• No way to specify limit on number of characters to read
• Similar problems with other library functions
  • strcpy, strcat: Copy strings of arbitrary length
  • scanf, fscanf, sscanf, when given % conversion specification
### Announcement 2/26/2016

- Bomb Lab due today 2/26/2016 at 11:55pm
- First Mid-term Exam next Monday 2/29/2016 in class
  - Coverage Chaps 1 – 3
  - Open book, open notes
  - No e-devices
- How to prepare for the exams?
  - Go over all lecture slides, and make sure you understand all of the materials covered there.
  - Go over Practice Problems as many as you can.
- UNITE has recorded lecture videos for review.
  
  [https://www.unite.umn.edu/](https://www.unite.umn.edu/)

### Review: Code Injection Attacks

- Add input string contains byte representation of executable code.
- Overwrite return address A with address of buffer B.
- When Q executes ret, will jump to exploit code

### Review: Buffer Overflow Stack

- Overflowed buffer, corrupted return pointer, but program seems to work.

### Aside: Worms and Viruses

- **Worm:** A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- **Virus:** Code that
  - Adds itself to other programs
  - Does not run independently
- Both are (usually) designed to spread among computers and to wreak havoc
OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each...

Review: char *gets(*dest)

- 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;
}
```

- No way to specify limit on number of characters to read
- Similar problems with other library functions
  - strcpy, strcat: Copy strings of arbitrary length
  - scanf, fscanf, sscanf, when given %s conversion specification

x86-64 Example Addresses

```
address range "2G"    0000000000000000

local    0x00007FFeba7c
p1 0x00007FF262a1e010
p3 0x00007FF262a1d010
p4 0x000000008359d010
big_array 0x0000000000601060
huge_array 0x0000000000601060
makes() 0x000000000060050c
useless() 0x000000000060050c
```

1. Avoid Overflow Vulnerabilities in Code (!)

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    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
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

```c
local /fed6b8c9f775e4f9f/fed6b8c9f775e4f9f
```

- Stack repositioned each time program executes

```
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

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

- 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__
  - New the default (disabled earlier)

---

Protected Buffer Disassembly

- **Before call to gets**
  - Stack Frame for call_echo
  - Return Address (8 bytes)
  - Canary (8 bytes)

---

Setting Up Canary

- **Before call to gets**
  - Stack Frame for call_echo
  - Return Address (8 bytes)
  - Canary (8 bytes)

---

Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code

- **Alternative Strategy**
  - Use existing code
    - e.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - Does not overcome stack canaries

- **Construct program from gadgets**
  - Sequence of instructions ending in ret
    - Encoded by single byte 0xa3
  - Code positions fixed from run to run

---

Gadget Example #1

- **Long ab_plus_c**
  - (long a, long b, long c)
    - return a*b + c;

---
Gadget Example #2

```c
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

Gadget address = 0x4004dc

- Repurpose byte codes

ROP Execution

- The `set` instruction does two things:
  - It pops the entry pointed by `rESP` to `rip` (the next instruction to execute)
  - It increments `rESP` to the next stack top, which stores the address of next Gadget code

- Final `set` in each gadget will start next one. It can create a function to do almost anything


Outline

- Memory Layout
- Buffer Overflow
- Vulnerability
- Protection
- Union

Union Allocation

- Allocate according to largest element
- Can only use one field at a time

Using Union to Access Bit Patterns

```c
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

Same as (float) `u`?  
Same as (unsigned) `f`?

Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines

- BigEndian
  - Most significant byte has lowest address
  - Sparc

- LittleEndian
  - Least significant byte has lowest address
  - Intel x86, ARM Android and iOS

- BiEndian
  - Can be configured either way
  - ARM
**Byte Ordering Example**

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

32-bit:
```
i[0]  i[1]
l[0]
```

64-bit:
```
i[0]  i[1]
l[0]
```

**Byte Ordering on IA32**

Little Endian:
```
0  e0  e1  e2  e3  e4  e5  e6  e7
i[0]  i[1]
l[0]
```

Output:
- Characters 0-7 == `[0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]`
- Shorts 0-3 == `[0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]`
- Ints 0-1 == `[0xf0f1f2f3, 0xf4f5f6f7]`
- Long 0 == `[0xf0f1f2f3]`

**Byte Ordering on Sun**

Big Endian:
```
0  e0  e1  e2  e3  e4  e5  e6  e7
i[0]  i[1]
l[0]
```

Output on Sun:
- Characters 0-7 == `[0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]`
- Shorts 0-3 == `[0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]`
- Ints 0-1 == `[0xf0f1f2f3, 0xf4f5f6f7]`
- Long 0 == `[0xf0f1f2f3]`

**Byte Ordering on x86-64**

Little Endian:
```
0  e0  e1  e2  e3  e4  e5  e6  e7
i[0]  i[1]
l[0]
```

Output on x86-64:
- Characters 0-7 == `[0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]`
- Shorts 0-3 == `[0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]`
- Ints 0-1 == `[0xf0f1f2f3, 0xf4f5f6f7]`
- Long 0 == `[0xf7f6f5f4f3f2f1f0]`

**Summary of Compound Types in C**

- Arrays
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - Pointer to first element
  - No bounds checking
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system