Today
- Memory Layout
- Unions
- Buffer Overflow
  - Vulnerability
  - Protection

Machine-Level Programming V: Advanced Topics

CSCI 2021: Machine Architecture and Organization
March 6th, 2020
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron

x86-64 Linux Memory Layout

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

Heap
- Dynamically allocated as needed
- When you call malloc(), calloc(), C++ new

Data
- Statically (compiler-)allocated data
- E.g., global vars, static vars, string constants

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

Hex Address

000000
000000
000000
000000

not drawn to scale

x86-64 Example Addresses
address range ~2^47

local 0x00007efe4d3be87c
p1 0x00007f7262a1e010
p3 0x00007f7162a1d010
p4 0x00000008355d9120
p2 0x00000008355d9120
big_array 0x0000000080601060
huge_array 0x0000000000601060
main() 0x0000000000040060
useless() 0x0000000000040050

not drawn to scale
Union Allocation

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

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp0</td>
<td>sp+4</td>
<td>sp+8</td>
<td>sp+16</td>
</tr>
</tbody>
</table>

Using Union to Access Bit Patterns

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

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

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

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

- Big Endian
  - Most significant byte has lowest address
  - Sparc

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

- Bi-Endian
  - Can be configured either way
  - ARM

Byte Ordering Example

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

<table>
<thead>
<tr>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[0]</td>
<td>c[0]</td>
</tr>
<tr>
<td>c[1]</td>
<td>c[1]</td>
</tr>
<tr>
<td>s[0]</td>
<td>s[0]</td>
</tr>
<tr>
<td>s[1]</td>
<td>s[1]</td>
</tr>
<tr>
<td>s[3]</td>
<td>s[3]</td>
</tr>
<tr>
<td>i[0]</td>
<td>i[0]</td>
</tr>
<tr>
<td>i[1]</td>
<td>i[1]</td>
</tr>
</tbody>
</table>

Byte Ordering on IA32

Little Endian

<table>
<thead>
<tr>
<th>0f</th>
<th>1e</th>
<th>2d</th>
<th>3c</th>
<th>4b</th>
<th>5a</th>
<th>69</th>
<th>78</th>
</tr>
</thead>
<tbody>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
<td>0f</td>
</tr>
<tr>
<td>1f</td>
<td>1f</td>
<td>1f</td>
<td>1f</td>
<td>1f</td>
<td>1f</td>
<td>1f</td>
<td>1f</td>
</tr>
</tbody>
</table>

Output:

- Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- Shorts 0-3 == [0xf0, 0xf1, 0xf2, 0xf3]
- Ints 0-1 == [0xf0, 0xf1]
- Long 0 == [0xf0]
Byte Ordering on Sun

Big Endian

<table>
<thead>
<tr>
<th>Bytes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>0x1</td>
<td>0x2</td>
<td>0x3</td>
<td>0x4</td>
<td>0x5</td>
<td>0x6</td>
<td>0x7</td>
</tr>
<tr>
<td>1</td>
<td>0x0</td>
<td>0x1</td>
<td>0x2</td>
<td>0x3</td>
<td>0x4</td>
<td>0x5</td>
<td>0x6</td>
<td>0x7</td>
</tr>
</tbody>
</table>

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]

Output on x86-64:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]

Byte Ordering on Sun

Little Endian

<table>
<thead>
<tr>
<th>Bytes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>0x1</td>
<td>0x2</td>
<td>0x3</td>
<td>0x4</td>
<td>0x5</td>
<td>0x6</td>
<td>0x7</td>
</tr>
<tr>
<td>1</td>
<td>0x0</td>
<td>0x1</td>
<td>0x2</td>
<td>0x3</td>
<td>0x4</td>
<td>0x5</td>
<td>0x6</td>
<td>0x7</td>
</tr>
</tbody>
</table>

Output on x86-64:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
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

Today

- **Memory Layout**
- **Unions**
- **Buffer Overflow**
  - Vulnerability
  - Protection

Recall: Memory Referencing Bug Example

```c
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; /* Possibly out of bounds */
  return s.d;
}
```

fun(0) -> 3.14
fun(1) -> 3.14
fun(2) -> 3.139999864856
fun(3) -> 3.00000000000000000000000000000
fun(4) -> 1.00000000000000000000000000000
fun(6) -> Segmentation fault

Result is system specific
Such problems are a BIG deal

- Generally called a “buffer overflow”
- When exceeding the memory size allocated for an array
- Why a big deal?
  - One of the most common technical causes of security vulnerabilities
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - Sometimes referred to as stack smashing

String Library Code

- Implementation of old standard C 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;
}
```

Buffer Overflow Disassembly

```
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 ff ff ff callq 400520 <puts@plt>
4006e3:
48 83 c4 18 add $0x18, %rsp
4006e7:
c3 retq
4006e8:
48 83 ec 08 sub $0x8, %rsp
4006e9:
b8 00 00 00 00 mov %0x0, %eax
4006ec:
e8 d9 ff ff ff callq 400680 <gets>
4006f1:
48 83 c4 08 add $0x8, %rsp
4006f2:
c3 retq
```

Buffer Overflow Stack Example

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

```
void call_echo()
{
    gets(buf);
    puts(buf);
}
```

```c
char buf[4]; /* Way too small! */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```
**Buffer Overflow Stack Example #1**

Stack Frame for `call_echo`

```
00 00 00 00
00 40 06 06
33 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
27 26 25 24
23 22 21 20
```

```
buf = esp
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
echo: 4006f1: callq 4006cf <echo>
400666: add $0x8,krsp
...
```

```
buf = esp
```

```
unix:/bufdemo-nsp
Type a string: 01234567890123456789012
```

Overflown buffer, but did not corrupt state.

---

**Buffer Overflow Stack Example #2**

Stack Frame for `call_echo`

```
00 00 00 00
00 40 06 06
33 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
27 26 25 24
23 22 21 20
```

```
buf = esp
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
echo: 4006f1: callq 4006cf <echo>
400666: add $0x8,krsp
...
```

```
buf = esp
```

```
unix:/bufdemo-nsp
Type a string: 0123456789012345678901234
```

Overflown buffer and corrupted return pointer.

---

**Buffer Overflow Stack Example #3**

Stack Frame for `call_echo`

```
00 00 00 00
00 40 06 06
33 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
27 26 25 24
23 22 21 20
```

```
buf = esp
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
echo: 4006f1: callq 4006cf <echo>
400666: add $0x8,krsp
...
```

```
buf = esp
```

```
unix:/bufdemo-nsp
Type a string: 012345678901234567890123
```

Overflown buffer, corrupted return pointer, but program seems to work.

---

**Buffer Overflow Stack Example #3 Explained**

Stack Frame for `call_echo`

```
00 00 00 00
00 40 06 06
33 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
27 26 25 24
23 22 21 20
```

```
buf = esp
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
void echo()
{
  char buf[4];
  gets(buf);
  ...
}
```

```
echo: 4006f1: callq 4006cf <echo>
400666: add $0x8,krsp
...
```

```
buf = esp
```

```
unix:/bufdemo-nsp
Type a string: 012345678901234567890123
```

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

---

**Code Injection Attacks**

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

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

- 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

---

**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 Wi1 (2000s)
  - ... and many, many more
- You will try out some techniques in lab
  - Hopefully to convince you to never leave such holes in your programs!!

---

Last page of the document.
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 drch@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
  - and CERT was formed

Discussion Break: Unknown Addresses?

- Basic attack requires attacker to know address B of buffer
- Is an attack still possible if B is variable?
- E.g. what if attacker only knows B +/- 30?

Some possible attack strategies:
- Try attack repeatedly
- “NOP sled”: (0x90 is one-byte no-operation in x86)

```
NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP NOP
```

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

Let’s talk about each...

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 `strncpy` where `n` is a suitable integer

2. System-Level Protections can help

- Randomized stack offsets (“ASLR”)
  - 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

Local: 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c
Random allocation:
- 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 era CPUs added explicit "(non-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 commonly enabled by default

```
unix> ./bufdoemo-tp
Type a string: 0123456
0123456
unix> ./bufdoemo-tp
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
```

Checking Canary

```
After call to gets
```

```
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>
```

```
Stack Frame for call_echo
```

```
Set Up Canary
```

```
echo:
33 32 31 30
buf ----> %rsp
```

```
Before call to gets
```

```
Ret-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 on its own overcome stack canaries

- Construct program from `gadgets`
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable
Gadget Example #1

```c
long ab_plus_c(long a, long b, long c) {
    return a*b + c;
}
```

Use tail end of existing functions

```
<table>
<thead>
<tr>
<th>OFFSET</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004d0</td>
<td>48 0f af fe</td>
</tr>
<tr>
<td>4004d4</td>
<td>48 8d 04 17</td>
</tr>
<tr>
<td>4004d8</td>
<td>c3</td>
</tr>
</tbody>
</table>
```

Gadget address = 0x4004d4

Gadget Example #2

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

Repurpose instruction bytes

```
<table>
<thead>
<tr>
<th>OFFSET</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>4004d9</td>
<td>c7 07 d4 48 89</td>
</tr>
<tr>
<td>4004df</td>
<td>c3</td>
</tr>
</tbody>
</table>
```

Gadget address = 0x4004dc

ROP Execution

- Trigger with `ret` instruction
  - Will start executing Gadget 1
  - Final `ret` in each gadget will start next one