Machine-Level Programming V: Advanced Topics

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

CSci 2021: Machine Architecture and Organization
October 17th, 2018
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

400000
000000

x86-64 Example Addresses

address range $\sim 2^{47}$

local
p1
p3
p4
p2
big_array
huge_array
main()
useless()

0x00007ffe4d3be87c
0x00007f7262a1e010
0x00007f7162a0d010
0x000000008359d120
0x000000008359d010
0x0000000008359d010
0x0000000000601060
0x000000000000400000
0x0000000000000400000
0x0000000000000400000

Memory Allocation Example

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?
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; /* Possibly out of bounds */
    return s->d;
}
```

**Explanation:**
- Critical State
- Location accessed by `fun(i)`

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

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
- Most common form
  - 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

```
echo:
  4006cf: 48 83 ec 18     sub    $0x18,%rsp
  4006d3: 48 89 97 04     mov     %rdi,%rdx
  4006d7: 48 83 c4 08     add    $0x8,%rsp
  4006e3: c3              retq
```
```c
void call_echo() {
    echo();
}
```
**Buffer Overflow Stack Example #3**

After call to `gets`

Stack Frame for `bufdemo`

```c
buf[0-3] 4006f6 echo
```

```c
buf demo
```

```c
buf[4-7] 4006f1 echo
```

```c
buf demo
```

Call to `gets`

```c
buf[0-3] 4006f6 echo
```

```c
buf demo
```

```c
buf[4-7] 4006f1 echo
```

```c
buf demo
```

```c
buf[0-3] 4006f6 echo
```

```c
buf demo
```

```c
buf[4-7] 4006f1 echo
```

```c
buf demo
```

```c
buf[0-3] 4006f6 echo
```

```c
buf demo
```

```c
buf[4-7] 4006f1 echo
```

```c
buf demo
```

```c
buf[0-3] 4006f6 echo
```

```c
buf demo
```

```c
buf[4-7] 4006f1 echo
```

```c
buf demo
```

```c
buf[0-3] 4006f6 echo
```

```c
buf demo
```

```c
buf[4-7] 4006f1 echo
```

```c
buf demo
```

```c
buf[0-3] 4006f6 echo
```

```c
buf demo
```

```c
buf[4-7] 4006f1 echo
```

```c
buf demo
```
Code Injection Attacks

- 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 Wii (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!!

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 finger 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)

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...

Announcements

- HA2 dlc fixes due tomorrow
- HA3 (binary bombs) out now
  - Due next Friday 10/26
1. Avoid Overflow Vulnerabilities in Code (!)

- 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

```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
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code
    - Stack repositioned each time program executes

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

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

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

Protected Buffer Disassembly

```
echo: 040072f: sub $0x18,%rsp
0400733: mov %fs:0x28,%rax
040073c: mov %rax,0x8(%rsp)
0400741: xor %eax,%eax
0400743: mov %rax,rax
0400746: callq 400580 <gets>@plt
040074b: mov %rax,rax
0400750: callq 400570 <puts@plt>
0400755: mov 0x8(%rsp),%rax
0400758: xor %fs:0x28,%rax
0400758: je 4000766 <echo+0x39>
0400761: je 4000768 <call_echo+0x10>
0400763: callq 400080 <__stack_chk_fail@plt>
0400768: add $0x18,%rsp
040076c: retq
```

Setting Up Canary

```
echo: 040072f: sub $0x18,%rsp
```

```
echo: 040072f: sub $0x18,%rsp
0400733: mov %fs:0x28,%rax
040073c: mov %rax,0x8(%rsp)
0400741: xor %eax,%eax
0400743: mov %rax,rax
0400746: callq 400580 <gets>@plt
040074b: mov %rax,rax
0400750: callq 400570 <puts@plt>
0400755: mov 0x8(%rsp),%rax
0400758: xor %fs:0x28,%rax
0400758: je 4000766 <echo+0x39>
0400761: je 4000768 <call_echo+0x10>
0400763: callq 400080 <__stack_chk_fail@plt>
0400768: add $0x18,%rsp
040076c: retq
```

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

After call to gets

Stack Frame for call_to_gets

Canary (8 bytes)

Input: 0123456

buf — trap

Gadget Example #1

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

00000000004004d0 <ab_plus_c>:
    48 0f af fe imul %rdi,%rdx
    4004d4:
    48 8d 04 17 lea (%rdi,%rdx,1),%rax
    4004d8:
    c3 retq

rax ← rdi + rdx
Gadget address = 0x4004d4

Use tail end of existing functions

Gadget Example #2

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

<setval>:
    c7 07 d4 48 89 c7 movl $0xc78948d4,(%rdi)
    4004df:
    c3 retq

rdi ← rax
Gadget address = 0x4004dc

Repurpose instruction 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 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

ROP Execution

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

Today

■ Memory Layout
■ Buffer Overflow
  • Vulnerability
  • Protection
■ Unions
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;
```

- c 3 bytes
- i[0] 4 bytes
- v

Using Union to Access Bit Patterns

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

```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
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;
```

```c
printf("Characters 0-7 == \n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);
```

```c
printf("Shorts 0-3 == \n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);
```

```c
printf("Ints 0-1 == \n",
    dw.i[0], dw.i[1]);
```

```c
printf("Long 0 == \n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

<table>
<thead>
<tr>
<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>i[0]</td>
<td>i[1]</td>
<td>i[2]</td>
<td>i[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:

- Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- Shorts 0-3 == [0xf020, 0xf040, 0xf060, 0xf080]
- Ints 0-1 == [0x3f2f1e0, 0x7f6565f4]
- Long 0 == [0x3f2f1e0]
Byte Ordering on Sun

Big Endian

<table>
<thead>
<tr>
<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>
</table>

Output on Sun:
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf001,0xf203,0xf405,0xf607]
Ints 0-1 == [0xf001f203,0xf4f6f5f7]
Long 0 == [0xf0f2f1f0]

Byte Ordering on x86-64

Little Endian

<table>
<thead>
<tr>
<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>
</table>

Output on x86-64:
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf2f3,0xf4f5,0xf7f6]
Ints 0-1 == [0xf3f2f1f0,0xf6f5f4f3f2f1f0]
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