Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O’Hallaron

Stack Overflow

With Slides from Bryant and O’Hallaron
String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```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 Buffer Code

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

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

With Slides from Bryant and O’Hallaron
Buffer Overflow Disassembly

**echo:**

80485c5:  55  push  %ebp
80485c6:  89 e5  mov  %esp, %ebp
80485c8:  53  push  %ebx
80485c9:  83 ec 14  sub  $0x14, %esp
80485cc:  8d 5d f8  lea  0xfffffff8(%ebp), %ebx
80485cf:  89 1c 24  mov  %ebx, (%esp)
80485d2:  e8 9e ff ff ff  call  8048575 <gets>
80485d7:  89 1c 24  mov  %ebx, (%esp)
80485da:  e8 05 fe ff ff  call  80483e4 <puts@plt>
80485df:  83 c4 14  add  $0x14, %esp
80485e2:  5b  pop  %ebx
80485e3:  5d  pop  %ebp
80485e4:  c3  ret

**call_echo:**

80485eb:  e8 d5 ff ff ff  call  80485c5 <echo>
80485f0:  c9  leave
80485f1:  c3  ret

Buffer Overflow Stack

Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for main</th>
<th>Saved %ebp</th>
<th>Saved %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
<td>%ebp</td>
<td>buf</td>
</tr>
</tbody>
</table>

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

echo:
pushl %ebp  # Save %ebp on stack
movl %esp, %ebp
pushl %ebx  # Save %ebx
subl $20, %esp  # Allocate stack space
leal -8(%ebp), %ebx  # Compute buf as %ebp-8
movl %ebx, (%esp)  # Push buf on stack
call gets  # Call gets
.
.
.
With Slides from Bryant and O’Hallaron
Buffer Overflow
Stack Example

Before call to gets

Stack Frame
for main

Return Address

Saved %ebp
Saved %ebx

buf

Stack Frame
for echo

0xffffd678
0xffffd688

80485eb: e8 d5 ff ff ff
80485f0: c9

Before call to gets

Stack Frame
for main

08 04 85 f0
ff ff d6 88
0xfffffd78

Saved %ebx

buf

Stack Frame
for echo

overflow buf, and corrupt %ebx, but no problem

With Slides from Bryant and O’Hallaron
Buffer Overflow Example #2

Before call to gets

Stack Frame for main
08 04 85 f0
ff ff d6 88 0xfffffd678
Saved %ebx
xx xx xx xx buf
Stack Frame for echo

Input 12345678

Stack Frame for main
08 04 85 f0
ff ff d6 80 0xfffffd688
Stack Frame for echo

Base pointer corrupted

... 80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Set %ebp to corrupted value
80485f1: c3 ret

Buffer Overflow Example #3

Before call to gets

Stack Frame for main
08 04 85 f0
ff ff d6 88 0xfffffd678
Saved %ebx
xx xx xx xx buf
Stack Frame for echo

Input 12345679

Stack Frame for main
08 04 85 dd 00
ff ff d6 88 0xfffffd678
Stack Frame for echo

Return address corrupted

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Desired return point

With Slides from Bryant and O’Hallaron
Malicious Use of Buffer Overflow

Input string contains byte representation of executable code
Overwrite return address with address of buffer
When bar() executes ret, will jump to exploit code

Avoiding Overflow Vulnerability

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
Yet Another Example

```c
main() {
    unsigned long long ll = 0xdeadbeefbeefdead;
    unsigned int i = 0x12345678;
    printf("%x %x\n", ll, i);
}
```

Yet Another Example

Main stack frame

With Slides from Bryant and O’Hallaron
System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code

- 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 Canaries

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

- GCC Implementation
  - -fstack-protector
  - -fstack-protector-all
### Protected Buffer Disassembly

```
804864d  55  push  %ebp
804864e  89 e5  mov  %esp,%ebp
8048650  53  push  %ebx
8048651  83 ec 14  sub  $0x14,%esp
8048654  65 a1 14 00 00 00  mov  %gs:0x14,%eax
804865a  89 45 f8  mov  %eax,0xffffffff8(%ebp)
804865d  31 c0  xor  %eax,%eax
804865f  8d 5d f4  lea  0xffffffff4(%ebp),%ebx
8048662  89 1c 24  mov  %ebx,(%esp)
8048665  e8 77 ff ff ff  call  80485e1 <gets>
8048666  e8 77 ff ff ff  call  804843c <puts@plt>
8048672  83 c4 14  add  $0x14,%esp
8048675  5b  pop  %ebx
8048676  5d  pop  %ebp
8048677  c3  ret
```

### Setting Up Canary

**Before call to gets**

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

**Before call to gets**

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

```
8048683  83 c4 14  add  $0x14,%esp
8048685  5b  pop  %ebx
8048687  5d  pop  %ebp
8048688  c3  ret
```

With Slides from Bryant and O’Hallaron
Checking Canary

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

Before call to gets

Stack Frame
for main

Return Address

Saved %ebp
Saved %ebx
Canary
[3][2][1][0]
Stack Frame
for echo

%ebp

buf

movl -8(%ebp), %eax       # Retrieve from stack
xorl %gs:20, %eax         # Compare with Canary
je .L24                   # Same: skip ahead
call __stack_chk_fail     # ERROR
.L24:
    ...  

Canary Example

Before call to gets

Stack Frame
for main

Return Address

Saved %ebp
Saved %ebx
03 e3 7d 00
[3][2][1][0]
Stack Frame
for echo

(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$s1 = 0x3e37d00

Input 1234

Stack Frame
for main

Return Address

Saved %ebp
Saved %ebx
03 e3 7d 00
34 33 32 31
Stack Frame
for echo

(gdb) print /x *((unsigned *) $ebp - 2)
$s1 = 0x3e37d00

Benign corruption!
allows programmers to make silent off-by-one errors

With Slides from Bryant and O'Hallaron
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
  - Add itself to other programs
  - Cannot run independently

- Both are (usually) designed to spread among computers and to wreak havoc

Non-Local Jumps

With Slides from Bryant and O'Hallaron
Nonlocal Jumps: `setjmp/longjmp`

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location
  - Controlled to way to break the procedure call / return discipline
  - Useful for error recovery and signal handling

- `int setjmp(jmp_buf j)`
  - Must be called before `longjmp`
  - Identifies a return site for a subsequent `longjmp`
  - Called once, returns one or more times

- Implementation:
  - Remember where you are by storing the current register context, stack pointer, and PC value in `jmp_buf`
  - Return 0

- `void longjmp(jmp_buf j, int i)`
  - Meaning:
    - return from the `setjmp` remembered by jump buffer `j` again ...
    - ... this time returning `i` instead of 0
  - Called after `setjmp`
  - Called once, but never returns

- `longjmp` Implementation:
  - Restore register context (stack pointer, base pointer, PC value) from jump buffer `j`
  - Set `%eax` (the return value) to `i`
  - Jump to the location indicated by the PC stored in jump buffer `j`
#include <setjmp.h>
jmp_buf buf;

main() {
    if (setjmp(buf) != 0) {
        printf("back in main due to an error\n");
        else
        printf("first time through\n");
        p1(); /* p1 calls p2, which calls p3 */
    }
    ...
    p3() {
        <error checking code>
        if (error)
            longjmp(buf, 1)
    }

Limitations of Nonlocal Jumps

- Works within stack discipline
  - Can only long jump to environment of function that has been called
    but not yet completed

```c
jmp_buf env;
P1()
{
    if (setjmp(env)) {
        /* Long Jump to here */
    } else {
        p2();
    }
}
P2()
{  . . . p2(); . . . p3();  }
P3()
{
    longjmp(env, 1);
}
```
Limitations of Long Jumps (cont.)

- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;
P1()
{
P2(); P3();
}
P2()
{
  if (setjmp(env)) {
    /* Long Jump to here */
  }
}
P3()
{
  longjmp(env, 1);
}
```

Procedures (x86-64)
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Twice the number of registers that are accessible as 8, 16, 32, 64 bits

### x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>%rax</th>
<th>Return value</th>
<th>%r8</th>
<th>Argument #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
x86-64 Registers

- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well

- All references to stack frame via stack pointer
  - Eliminates need to update %ebp/%rbp

- Other Registers
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required (except ret)
- Avoiding stack
  - Can hold all local information in registers
x86-64 Locals in the Red Zone

/* Swap, using local array */
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}

Avoiding Stack Pointer Change
• Can hold all information within small window beyond stack pointer

x86-64 NonLeaf without Stack Frame

/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i) {
    swap(&a[i], &a[i+1]);
}

swap_ele:
    movslq %esi,%rsi          # Sign extend i
    leaq  8(%rdi,%rsi,8), %rax # &a[i+1]
    leaq  (%rdi,%rsi,8), %rdi  # &a[i] (1st arg)
    movq %rax, %rsi           # (2nd arg)
    call  swap                # No-op
    rep
    ret

With Slides from Bryant and O’Hallaron
x86-64 Stack Frame Example

```c
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of &a[i] and &a[i+1] in callee save registers
- Must set up stack frame to save these registers

```assembly
swap_ele_su:
movq %rbx, -16(%rsp)
movq %rbp, -8(%rsp)
subq $16, %rsp
movslq %esi,%rax
leaq 8(%rdi,%rax,8), %rbx
leaq (%rdi,%rax,8), %rbp
movq %rbx, %rsi
movq %rbp, %rdi
call swap
movq (%rbx), %rax
imulq (%rbp), %rax
addq %rax, sum(%rip)
movq (%rsp), %rbx
movq 8(%rsp), %rbp
addq $16, %rsp
ret
```

Understanding x86-64 Stack Frame

```assembly
swap_ele_su:
movq %rbx, -16(%rsp) # Save %rbx
movq %rbp, -8(%rsp) # Save %rbp
subq $16, %rsp # Allocate stack frame
movslq %esi,%rax # Extend i
leaq 8(%rdi,%rax,8), %rbx # &a[i+1] (callee save)
leaq (%rdi,%rax,8), %rbp # &a[i] (callee save)
movq %rbx, %rsi # 2nd argument
movq %rbp, %rdi # 1st argument
call swap
movq (%rbx), %rax # Get a[i+1]
imulq (%rbp), %rax # Multiply by a[i]
addq %rax, sum(%rip) # Add to sum
movq (%rsp), %rbx # Restore %rbx
movq 8(%rsp), %rbp # Restore %rbp
addq $16, %rsp # Deallocate frame
ret
```

With Slides from Bryant and O’Hallaron
Understanding x86-64 Stack Frame

```
movq %rbx, -16(%rsp)  # Save %rbx
movq %rbp, -8(%rsp)   # Save %rbp

subq $16, %rsp       # Allocate stack frame
```

```
...  # Save %rbx
movq %rbp, -8(%rsp)  # Save %rbp
movq (%rsp), %rbx    # Restore %rbx
movq 8(%rsp), %rbp   # Restore %rbp
addq $16, %rsp       # Deallocate frame
```

Interesting Features of Stack Frame

**Allocate entire frame at once**

- All stack accesses can be relative to %rsp
- Do by decrementing stack pointer
- Can delay allocation, since safe to temporarily use red zone

**Simple deallocation**

- Increment stack pointer
- No base/frame pointer needed
x86-64 Procedure Summary

Heavy use of registers
- Parameter passing
- More temporaries since more registers

Minimal use of stack
- Sometimes none
- Allocate/deallocate entire block

Many tricky optimizations
- What kind of stack frame to use
- Various allocation techniques

With Slides from Bryant and O'Hallaron