CSC 252: Computer Organization
Fall 2021: Lecture 9

Assembly Programming:
  • Data Structures
  • Buffer overflow

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

A2 due Thursday night

Today’s lecture is the last one on assembly programming

A3 will be out this weekend
Managing Function Local Variables

• Two ways: registers and memory (stack)
• Registers are faster, but limited. Memory is slower, but large. Smart compilers will optimize the usage.

```c
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```
Register Saving Conventions
Register Saving Conventions

• Any issue with using registers for temporary storage?

Caller

yoo:
  ...  
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  ...
  ret

Callee

who:
  ...  
  subq $18213, %rdx
  ...
  ret
Register Saving Conventions

• Any issue with using registers for temporary storage?
  • Contents of register `%rdx` overwritten by `who()`

```
Caller

yoo:
  ...
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  ...
  ret

Callee

who:
  ...
  subq $18213, %rdx
  ...
  ret
```
Register Saving Conventions

• Any issue with using registers for temporary storage?
  • Contents of register %rdx overwritten by who()
  • This could be trouble ➔ Need some coordination

```assembly
yoo:
  ...
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  ...
  ret

who:
  ...
  subq $18213, %rdx
  ...
  ret
```
Register Saving Conventions

• Common conventions
  • “Caller Saved”
  • Caller saves temporary values in its frame (on the stack) before the call
  • Callee is then free to modify their values
  • “Callee Saved”
  • Callee saves temporary values in its frame before using
  • Callee restores them before returning to caller
  • Caller can safely assume that register values won’t change after the function call
Register Saving Conventions

- Conventions used in x86-64 (*Part of the Calling Conventions*)
  - Some registers are saved by caller, some are by callee.
  - Caller saved: %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11
  - Callee saved: %rbx, %rbp, %r12, %r13, %r14, %r15
  - %rax holds return value, so implicitly caller saved
  - %rsp is the stack pointer, so implicitly callee saved
Example

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```
Example

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

call_incr2:
```
pushq   %rbx
pushq   $15213
movq    %rdi, %rbx
movl    $3000, %esi
leaq    (%rsp), %rdi
call    incr
addq    %rbx, %rax
addq    $8, %rsp
popq    %rbx
ret
```
Example

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}

call_incr2:
    pushq %rbx
    pushq $15213
    movq %rdi, %rbx
    movl $3000, %esi
    leaq (%rsp), %rdi
    call incr
    addq %rbx, %rax
    addq $8, %rsp
    popq %rbx
    ret
**Example**

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

call_incr2:
```
pushq   %rbx
pushq   $15213
movq    %rdi, %rbx
movl    $3000, %esi
lea     (%rsp), %rdi
call    incr
addq    %rbx, %rax
addq    $8, %rsp
popq    %rbx
ret
```
Example

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

call_incr2:

```assembly
pushq  %rbx
pushq  $15213
movq   %rdi, %rbx
movl   $3000, %esi
lea    (%rsp), %rdi
call   incr
addq   %rbx, %rax
addq   $8, %rsp
popq   %rbx
ret
```

Stack

```
...  
...  
Saved %rbx
15213
%rsp
```
Example

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

call_incr2:

```
    pushq    %rbx
    pushq    $15213
    movq     %rdi, %rbx
    movl     $3000, %esi
    leaq     (%rsp), %rdi
    call     incr
    addq     %rbx, %rax
    addq     $8, %rsp
    popq    %rbx
    ret
```
Example

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

call_incr2:
```
pushq   %rbx
pushq   $15213
movq    %rdi, %rbx
movl    $3000, %esi
leaq    (%rsp), %rdi
    call incr
    addq   %rbx, %rax
    addq   $8, %rsp
popq    %rbx
ret
```
Example

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

call_incr2:
```assembly
  pushq   %rbx
  pushq   $15213
  movq    %rdi, %rbx
  movl    $3000, %esi
  leaq    (%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $8, %rsp
  popq    %rbx
  ret
```

Stack

- call_incr2 needs to save %rbx (callee-saved) because it will modify its value
- It can safely use %rbx after call incr because incr will have to save %rbx if it needs to use it (again, %rbx is callee saved)
Stack Frame: Putting It Together

Caller Frame

%rsp

Callee Frame

Saved Registers
Local Variables
Arguments 7, 8, …
Return Addr
Today: Data Structures and Buffer Overflow

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)

• Structures
  • Allocation
  • Access
  • Alignment

• Buffer Overflow
Array Allocation: Basic Principle

\[ T \text{ A}[L]; \]

- Array of data type \( T \) and length \( L \)
- Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes in memory

char string[12];

\[
\begin{array}{c}
\text{x} \\
\text{x + 12}
\end{array}
\]

int val[5];

\[
\begin{array}{ccccccc}
\text{x} & \text{x + 4} & \text{x + 8} & \text{x + 12} & \text{x + 16} & \text{x + 20}
\end{array}
\]

double a[3];

\[
\begin{array}{cccc}
\text{x} & \text{x + 8} & \text{x + 16} & \text{x + 24}
\end{array}
\]

char* p[3];

\[
\begin{array}{cccc}
\text{x} & \text{x + 8} & \text{x + 16} & \text{x + 24}
\end{array}
\]
Byte Ordering
Byte Ordering

• How are the bytes of a multi-byte variable ordered in memory?
Byte Ordering

• How are the bytes of a multi-byte variable ordered in memory?
• Example
  • Variable x has 4-byte value of 0x01234567
  • Address given by &x is 0x100

0x100 0x101 0x102 0x103
Byte Ordering

• How are the bytes of a multi-byte variable ordered in memory?
• Example
  • Variable x has 4-byte value of 0x01234567
  • Address given by &x is 0x100
• Conventions
  • **Big Endian**: Sun, PPC Mac, IBM z, Internet
    • Most significant byte has lowest address (**MSB first**)
  • **Little Endian**: x86, ARM
    • Least significant byte has lowest address (**LSB first**)

```
0x100 0x101 0x102 0x103
          01  23  45  67
```
Byte Ordering

• How are the bytes of a multi-byte variable ordered in memory?
• Example
  • Variable x has 4-byte value of 0x01234567
  • Address given by &x is 0x100
• Conventions
  • **Big Endian**: Sun, PPC Mac, IBM z, Internet
    • Most significant byte has lowest address (**MSB first**)
  • **Little Endian**: x86, ARM
    • Least significant byte has lowest address (**LSB first**)

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>
Byte Ordering

• How are the bytes of a multi-byte variable ordered in memory?
• Example
  • Variable x has 4-byte value of 0x01234567
  • Address given by &x is 0x100
• Conventions
  • **Big Endian**: Sun, PPC Mac, IBM z, Internet
    • Most significant byte has lowest address (**MSB first**)
  • **Little Endian**: x86, ARM
    • Least significant byte has lowest address (**LSB first**)

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>0x101</td>
<td>0x101</td>
</tr>
<tr>
<td>0x102</td>
<td>0x102</td>
</tr>
<tr>
<td>0x103</td>
<td>0x103</td>
</tr>
</tbody>
</table>

**Big Endian**

- 0x100
- 0x101
- 0x102
- 0x103

**Little Endian**

- 0x100
- 0x101
- 0x102
- 0x103
Byte Ordering

• How are the bytes of a multi-byte variable ordered in memory?
• Example
  • Variable x has 4-byte value of 0x01234567
  • Address given by &x is 0x100

• Conventions
  • **Big Endian**: Sun, PPC Mac, IBM z, Internet
    • Most significant byte has lowest address (**MSB first**)
  • **Little Endian**: x86, ARM
    • Least significant byte has lowest address (**LSB first**)

```plaintext
Big Endian

<table>
<thead>
<tr>
<th></th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

Little Endian

<table>
<thead>
<tr>
<th></th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
```
## Representing Integers

### int A = 15213;

<table>
<thead>
<tr>
<th>Little-E</th>
<th>Big-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>6D</td>
<td>00</td>
</tr>
<tr>
<td>3B</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>3B</td>
</tr>
<tr>
<td>00</td>
<td>6D</td>
</tr>
</tbody>
</table>

### int B = -15213;

<table>
<thead>
<tr>
<th>Little-E</th>
<th>Big-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>FF</td>
</tr>
<tr>
<td>C4</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>C4</td>
</tr>
<tr>
<td>FF</td>
<td>93</td>
</tr>
</tbody>
</table>

Hex: 00003B6D

Hex: FFFFC493

Address Increase
Representing Integers

Hex: 00003B6D

Little-E

| 6D | 3B | 00 | 00 |

Big-E

| 00 | 00 | 3B | 6D |

Hex: FFFFC493

Little-E

| 93 | C4 | FF | FF |

Big-E

| FF | FF | C4 | 93 |

int A = 15213;

int B = -15213;
Representing Integers

Hex: 00003B6D

int A = 15213;

Hex: FFFFFC493

int B = -15213;

Address Increase
Array Access: Basic Principle

\[ T \ A[L]; \]

- Array of data type \( T \) and length \( L \)
- Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>( x )</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>val + ( i )</td>
<td>int *</td>
<td>( x + 4 \ i )</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
</tbody>
</table>
Multidimensional (Nested) Arrays
Multidimensional (Nested) Arrays

- Declaration
  
  \[ T \ A[R][C] ; \]
  
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes

\[
\begin{bmatrix}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & \ddots & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{bmatrix}
\]
Multidimensional (Nested) Arrays

- Declaration
  \[ T \ A[R][C] ; \]
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes

- Array Size
  - \( R \times C \times K \) bytes
Multidimensional (Nested) Arrays

- **Declaration**
  
  \( T \ A[R][C]; \)
  
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes

- **Array Size**
  
  - \( R \times C \times K \) bytes

- **Arrangement**
  
  - Row-Major Ordering in most languages, including C

```c
int A[R][C];
```

<table>
<thead>
<tr>
<th>A</th>
<th>...</th>
<th>A</th>
<th>...</th>
<th>A</th>
<th>...</th>
<th>A</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[0]</td>
<td>[1]</td>
<td>[0]</td>
<td>[1]</td>
<td>[0]</td>
<td>[R-1]</td>
<td>[0]</td>
</tr>
<tr>
<td>[0]</td>
<td>[C-1]</td>
<td>[0]</td>
<td>[C-1]</td>
<td>[C-1]</td>
<td>[C-1]</td>
<td>[R-1]</td>
<td>[C-1]</td>
</tr>
</tbody>
</table>

\( 4\times R \times C \) Bytes
Nested Array Row Access

- \( T \ A[R][C]; \)
  - \( A[i] \) is array of \( C \) elements
  - Each element of type \( T \) requires \( K \) bytes
  - Starting address \( A + i \times (C \times K) \)

```
int A[R][C];
```

![Diagram Illustrating Array Access](image)
Nested Array Element Access

- Array Elements
  - \( A[i][j] \) is element of type \( T \), which requires \( K \) bytes
  - Address \( A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K \)

```c
int A[R][C];
```

```
A[0][0]  ...  A[R-1][0]  ...  A[R-1][C-1]

A+(i*C*4)  A+[i][j]  A+((R-1)*C*4)
```

\( A+(i*C*4)+(j*4) \)
Today: Data Structures and Buffer Overflow

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)

- Structures
  - Allocation
  - Access
  - Alignment

- Buffer Overflow
Structures

```c
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

- Characteristics
  - Contiguously-allocated region of memory
  - Refer to members within struct by names
  - Members may be of different types
Access Struct Members

```c
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

- Given a struct, we can use the . operator:
  - `struct rec r1; r1.i = val;`
Access Struct Members

- Given a struct, we can use the . operator:
  - `struct rec rl; rl.i = val;`
- Suppose we have a pointer `r` pointing to `struct rec`. How to access `rec`’s member using `r`?
Access Struct Members

```c
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

- Given a struct, we can use the . operator:
  - `struct rec rl; rl.i = val;`
- Suppose we have a pointer `r` pointing to `struct res`. How to access `res`’s member using `r`?
  - Using * and . operators: `(*r).i = val;`
Access Struct Members

- Given a struct, we can use the . operator:
  - `struct rec r1; r1.i = val;`
- Suppose we have a pointer `r` pointing to struct `res`. How to access `res`’s member using `r`?
  - Using * and . operators: `(*r).i = val;`
  - Or simply, the -> operator for short: `r->i = val;`
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

![Diagram of structure layout]
Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    double i;
    struct rec *next;
};
```

```
int *get_ap
    (struct rec *r, size_t idx)
{
    return &(r->a[idx]);
}
```
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    double i;
    struct rec *next;
};

int *get_ap (struct rec *r, size_t idx)
{
    return &(r->a[idx]);
}
```

Diagram:
```
\begin{array}{c}
\text{struct rec} & \text{struct rec *next} \\
\text{int a[4]} & \text{i} \\
0 & 16 & 24 & 32 \\
\end{array}
```
Generating Pointer to Structure Member

```c
struct rec {
    int a[4];
    double i;
    struct rec *next;
};

int *get_ap (struct rec *r, size_t idx) {
    return &(r->a[idx]);
}
```

# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret

&((*r).a[idx])
Alignment

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

• Unaligned Data

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

- **Unaligned Data**

- **Aligned Data**
  - If the data type requires $K$ bytes, address must be multiple of $K$

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

• Unaligned Data

\[
\begin{array}{cccc}
  c & i[0] & i[1] & v \\
  p & p+1 & p+5 & p+9 \\
\end{array}
\]

\[p+17\]

• Aligned Data

  • If the data type requires \( K \) bytes, address must be multiple of \( K \)

\[
\begin{array}{cc}
  c & p+0 \\
  \uparrow & \uparrow \\
  \text{Multiple of 8} & \\
\end{array}
\]

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

- **Unaligned Data**

- **Aligned Data**
  - If the data type requires $K$ bytes, address must be multiple of $K$

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

- Unaligned Data

- Aligned Data
  - If the data type requires $K$ bytes, address must be multiple of $K$
Alignment

• Unaligned Data

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

• Aligned Data

  • If the data type requires $K$ bytes, address must be multiple of $K$
Alignment

• Unaligned Data
  - c
  - i[0]
  - i[1]
  - v

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

• Aligned Data
  - If the data type requires $K$ bytes, address must be multiple of $K$

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

• Unaligned Data

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
<td>p+9</td>
<td>p+17</td>
</tr>
</tbody>
</table>

• Aligned Data

- If the data type requires $K$ bytes, address must be multiple of $K$
Alignment Principles

• Aligned Data
  • If the data type requires K bytes, address must be multiple of K
• Required on some machines; advised on x86-64
• Motivation for Aligning Data: Performance
  • Inefficient to load or store data that is unaligned
  • Some machines don’t even support unaligned memory access
• Compiler
  • Inserts gaps in structure to ensure correct alignment of fields
  • sizeof() returns the actual size of structs (i.e., including padding)
Specific Cases of Alignment (x86-64)

• 1 byte: \texttt{char}, ...
  • no restrictions on address

• 2 bytes: \texttt{short}, ...
  • lowest 1 bit of address must be 0_2

• 4 bytes: \texttt{int}, \texttt{float}, ...
  • lowest 2 bits of address must be 00_2

• 8 bytes: \texttt{double}, \texttt{long}, \texttt{char*}, ...
  • lowest 3 bits of address must be 000_2
Satisfying Alignment with Structures
Satisfying Alignment with Structures

• Within structure:
  • Must satisfy each element’s alignment requirement
Satisfying Alignment with Structures

• Within structure:
  • Must satisfy each element’s alignment requirement

• Overall structure placement
  • Structure length must be multiples of \( K \), where:
    • \( K \) = Largest alignment of any element
  • WHY?!
Satisfying Alignment with Structures

• **Within structure:**
  • Must satisfy each element’s alignment requirement

• **Overall structure placement**
  • Structure length must be multiples of $K$, where:
    • $K =$ Largest alignment of any element
  • **WHY?!**

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```
Saving Space

• Put large data types first in a Struct
• This is not something that a C compiler would always do
  • But knowing low-level details empower a C programmer to write more efficient code

```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```
Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Return Struct Values
Return Struct Values

```c
struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n", test.a, test.b);
    // you will get "3, 4" from the terminal
}
```
Return Struct Values

struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n",
    test.a, test.b);
    // you will get “3, 4” from
    the terminal
}
Return Struct Values

```c
struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n", test.a, test.b);
    // you will get "3, 4" from the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in `%rax`??
Return Struct Values

```c
struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n", test.a, test.b);
    // you will get "3, 4" from the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in %rax??
- We don’t have to follow that convention...
Return Struct Values

```c
struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n", test.a, test.b);
    // you will get "3, 4" from the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in `%rax`??
- We don’t have to follow that convention…
- If there are only a few members in a struct, we could return through a few registers.
Return Struct Values

```c
struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n", test.a, test.b);
    // you will get “3, 4″ from the terminal
}
```

- This is perfectly fine.
- A struct could contain many members, how would this work if the return value has to be in `%rax`??
- We don’t have to follow that convention…
- If there are only a few members in a struct, we could return through a few registers.
- If there are lots of members, we could return through memory, i.e., requires memory copy.
Return Struct Values

struct S{
  int a, b;
};

struct S foo(int c, int d){
  struct S retval;
  retval.a = c;
  retval.b = d;
  return retval;
}

void bar() {
  struct S test = foo(3, 4);
  printf(stdout, "%d, %d\n", test.a, test.b);
  // you will get "3, 4" from the terminal
}
Return Struct Values

```c
struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n", test.a, test.b);
    // you will get “3, 4” from the terminal
}
```
Return Struct Values

```c
struct S{
    int a, b;
};

struct S foo(int c, int d){
    struct S retval;
    retval.a = c;
    retval.b = d;
    return retval;
}

void bar() {
    struct S test = foo(3, 4);
    fprintf(stdout, "%d, %d\n", test.a, test.b);
    // you will get "3, 4" from the terminal
}
```

- The entire calling convention is part of what’s called Application Binary Interface (ABI), which specifies how **two binaries** should interact.
- ABI includes: ISA, data type size, calling convention, etc.
- API defines the interface as the **source code** (e.g., C) level.
- The OS and compiler have to agree on the ABI.
- Linux x86-64 ABI specifies that returning a struct with two scalar (e.g. pointers, or long) values is done via `%rax` & `%rdx`
Today: Data Structures and Buffer Overflow

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)

• Structures
  • Allocation
  • Access
  • Alignment

• Buffer Overflow
/* 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;
}
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;
}
```
String Library Code

- Implementation of Unix function `gets()`
  - 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

```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;
}
```
Vulnerable Buffer Code

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

void call_echo() {
    echo();
}
Vulnerable Buffer Code

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

void call_echo()
{
    echo();
}
Vulnerable Buffer Code

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

void call_echo() {
    echo();
}

unix>./bufdemo-nsp
Type a string: 012345678901234567890123
  012345678901234567890123
unix>./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault
Buffer Overflow Stack Example

Before call to gets

Stack Frame for call_echo

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
<td></td>
</tr>
</tbody>
</table>

Stack Frame for echo

20 bytes unused

[3] [2] [1] [0]

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

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:

    ...

4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp
    ...

buf ← %rsp
Buffer Overflow Stack Example #1

After call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 f6</td>
</tr>
<tr>
<td>00 32 31 30</td>
</tr>
<tr>
<td>39 38 37 36</td>
</tr>
<tr>
<td>35 34 33 32</td>
</tr>
<tr>
<td>31 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

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

overflowed buffer, but did not corrupt state

call_echo:

... 4006f1: callq 4006cf <echo>
  4006f6: add $0x8,%rsp
...

unix> ./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012

Overflowed buffer, but did not corrupt state
Buffer Overflow Stack Example #2

After call to gets

Stack Frame for call_echo

| 00 | 00 | 00 | 00 |
| 00 | 40 | 00 | 34 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

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

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...
    4006f1: callq 4006cf <echo>
    4006f6: add $0x8, %rsp
    ...

buf ← %rsp

unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault

Overflowed buffer, and corrupt return address
Buffer Overflow Stack Example #3

After call to gets

Stack Frame
for call_echo

| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | 00 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

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

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...

4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp
    ...

buf ← %rsp

unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123

Overflowed buffer, corrupt return address, but program appears to still work!
Buffer Overflow Stack Example #4

After call to gets

Stack Frame for call_echo

| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | 00 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

buf ← %rsp

register_tm_clones:

```
... 400600: mov %rsp,%rbp
      400603: mov %rax,%rdx
      400606: shr $0x3f,%rdx
      40060a: add %rdx,%rax
      40060d: sar %rax
      400610: jne 400614
      400612: pop %rbp
      400613: retq
```

“Returns” to unrelated code
Could be code controlled by attackers!