Machine Level Programming: Data and Advanced Concepts

Computer Systems Organization (Spring 2015)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska
Teaching Assistants: Paige Connelly & Carlos Guzman

Slides adapted from
Andrew Case, Jinyang Li, Mohamed Zahran, Randy Bryant and Dave O’Hallaron
Today

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

• Structures

• Memory layout

• Buffer overflow
Today

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

• Structures
• Memory layout
• Buffer overflow
Array Allocation

- **Basic Principle**
  \[ T \ A[L] \];
  - Array of data type \( T \) and length \( L \)
  - Contiguously allocated region of \( L * \text{sizeof}(T) \) bytes

char string[12];

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

int val[5];

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

double a[3];

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

char *p[3];

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

\[ x \]
\[ x + 4 \]
\[ x + 8 \]
\[ x + 12 \]
\[ x + 16 \]
\[ x + 20 \]
\[ x + 24 \]
Array Access

• Basic Principle
  \[ T \text{ A}[L] ; \]
  • Array of data type \( T \) and length \( L \)
  • Identifier \( A \) can be used as a pointer to array element 0: Type \( T^\star \)

```c
int val[5];
```

<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>&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>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x + 4 ( i )</td>
</tr>
</tbody>
</table>
Array Example

```c
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig nyu = { 1, 0, 0, 1, 2 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

- Declaration “`zip_dig nyu`” equivalent to “`int nyu[5]`”
- Example arrays were allocated in successive 20 byte blocks (`nyu` is immediately followed by `mit` and then `ucb`) - this is not guaranteed to happen in general
Array Accessing Example

```
int get_digit (zip_digit z, int dig) {
    return z[dig];
}
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx, %eax, 4)
Array Loop Example (IA32)

```c
void zincr(zip_dig z) {
    int i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```assembly
# edx = z
movl $0, %eax  # %eax = i
.L4:            # loop:
    addl $1, (%edx,%eax,4)  # z[i]++
    addl $1, %eax           # i++
    cmpl $5, %eax           # i:5
    jne .L4                # if !==, goto loop
```

# Array Loop Example (IA32)

```c
void zincr(zip_dig z) {
    int i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```assembly
# edx = z
movl $0, %eax  # %eax = i
.L4:            # loop:
    addl $1, (%edx,%eax,4)  # z[i]++
    addl $1, %eax           # i++
    cmpl $5, %eax           # i:5
    jne .L4                # if !==, goto loop
```
2DArray Example

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

- `zip_dig pgh[4]` equivalent to `int pgh[4][5]`
  - Variable `pgh`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously

- “Row-Major” ordering of all elements guaranteed
  - Given a 2D array `int A[R][C]`,
    `A[i][j]` is equivalent to `*(A+i*C+j)`
Nested Array Element Access Code

```c
int get_pgh_digit (int index, int dig)
{
    return pgh[index][dig];
}
```

```assembly
movl 8(%ebp), %eax        # index
leal (%eax,%eax,4), %eax  # 5*index
addl 12(%ebp), %eax       # 5*index+dig
movl pgh(,%eax,4), %eax   # offset 4*(5*index+dig)
```

- **Array Elements**
  - `pgh[index][dig]` is int
  - **address**: `pgh + 20*index + 4*dig`
  - **IA32 Code** computes address as
    
    ```
    pgh + 4*((index+4*index)+dig)
    ```
Today

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

• Structures

• Memory layout

• Buffer overflow
Structure Allocation

```c
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types
Structure Access

```c
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

• Accessing Structure Member
  • Pointer indicates first byte of structure
  • Access elements with offsets

```c
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

IA32 Assembly

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```
Structures & Alignment

• Unaligned Data

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

- Primitive data type requires $K$ bytes
- Address must be multiple of $K$

• Aligned Data

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i[0]</th>
<th>i[1]</th>
<th>4 bytes</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
<td>p+24</td>
<td></td>
</tr>
</tbody>
</table>

- Multiple of 4
- Multiple of 8

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

• Motivation for Aligning Data
  • Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent) – it is inefficient to load or store data elements that span word boundaries

• Compiler
  • Inserts gaps in structure to ensure correct alignment of fields

• Specific cases of alignment (IA32)
  • 1 byte: char - no restrictions on address
  • 2 bytes: short - lowest 1 bit of address must be 02
  • 4 bytes: int, float, char * - lowest 2 bits of address must be 002
  • 8 bytes: double
    • Windows (and most other OS’s & instruction sets): lowest 3 bits of address must be 0002
    • Linux: lowest 2 bits of address must be 002, i.e., treated the same as a 4-byte primitive data type
  • 12 bytes: long double - lowest 2 bits of address must be 002, i.e., treated the same as a 4-byte primitive data type
**Saving Space**

- Put large data types first

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

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

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>d</th>
<th>2 bytes</th>
</tr>
</thead>
</table>
Today

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

• Structures

• Memory layout

• Buffer overflow
IA32 Linux Memory Layout

• Stack
  • E.g., local variables

• Heap
  • Dynamically allocated storage
  • When using `malloc()`, `calloc()`, `new`

• Data
  • Statically allocated data:
    • arrays
    • strings declared in code
    • globals

• Text
  • Executable machine instructions
  • Read-only
Memory Allocation Example

```c
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */
int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    p1 = malloc(1 <<28); /* 256 MB */
    p2 = malloc(1 << 8); /* 256 B */
    p3 = malloc(1 <<28); /* 256 MB */
    p4 = malloc(1 << 8); /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?
IA32 Example Addresses

address range \textasciitilde2^{32}

\begin{itemize}
\item \$esp \quad 0xfffffbc0d0
\item p3 \quad 0x65586008
\item p1 \quad 0x55585008
\item p4 \quad 0x1904a110
\item p2 \quad 0x1904a008
\item \&p2 \quad 0x18049760
\item \&beyond \quad 0x08049744
\item big\_array \quad 0x18049780
\item huge\_array \quad 0x08049760
\item main() \quad 0x080483c6
\item useless() \quad 0x08049744
\item final malloc() \quad 0x006be166
\end{itemize}

\texttt{malloc()} is dynamically linked
address determined at runtime

not drawn to scale
Today

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

• **Structures**

• **Memory layout**

• **Buffer overflow**
Internet Worm and IM War

• November, 1988
  • Internet Worm attacks thousands of Internet hosts.
  • How did it happen?
String Library Code

- 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
Vulnerable Buffer Code

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

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

```
unix> ./bufdemo
Type a string: 1234567
1234567

unix> ./bufdemo
Type a string: 12345678
Segmentation Fault

unix> ./bufdemo
Type a string: 123456789ABC
Segmentation Fault
```
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  0xffffffff8(%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

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
[3] [2] [1] [0]

%ebp
buf

Stack Frame for echo

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

. . .

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

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xfffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xfffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0

Before call to gets

Stack Frame for main

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

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

Before call to gets

Stack Frame for main

0xfffffd688
0xfffffd678
Saved %ebx
Saved %ebx
xx xx xx xx
Stack Frame for echo

call 80485c5 <echo>
leave
Buffer Overflow Example #1

**Before call to gets**

<table>
<thead>
<tr>
<th>Stack Frame for main</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 f0</td>
</tr>
<tr>
<td>ff ff d6 88</td>
</tr>
<tr>
<td>Saved %ebx</td>
</tr>
<tr>
<td>xx xx xx xx</td>
</tr>
</tbody>
</table>

**Stack Frame for echo**

0xffffd678

**Input 1234567**

<table>
<thead>
<tr>
<th>Stack Frame for main</th>
</tr>
</thead>
<tbody>
<tr>
<td>08 04 85 f0</td>
</tr>
<tr>
<td>ff ff d6 88</td>
</tr>
<tr>
<td>00 37 36 35</td>
</tr>
<tr>
<td>34 33 32 31</td>
</tr>
</tbody>
</table>

0xffffd688

Overflow buf, and corrupt %ebx, but no problem
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

08 04 85 f0
ff ff d6 88
Saved %ebx
xx xx xx xx

buf

Stack Frame for echo

Input 12345678

Stack Frame for main

08 04 85 f0
ff ff d6 00
38 37 36 35
34 33 32 31
buf

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

Input 123456789

Return address corrupted

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Desired return point
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When \texttt{bar()} executes \texttt{ret}, will jump to exploit code
Exploits Based on Buffer Overflows

•*Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*

•Internet worm
  •Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    •`finger droh@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.
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
  • Or use `%ns` where `n` is a suitable integer

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

```
unix>./bufdemo-protected
Type a string: 1234
1234
1234
```

```
unix>./bufdemo-protected
Type a string: 12345
*** stack smashing detected ***
```
## Protected Buffer Disassembly echo:

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>804864d:</td>
<td>push %ebp</td>
<td></td>
</tr>
<tr>
<td>804864e:</td>
<td>mov %esp, %ebp</td>
<td></td>
</tr>
<tr>
<td>8048650:</td>
<td>push %ebx</td>
<td></td>
</tr>
<tr>
<td>8048651:</td>
<td>sub $0x14, %esp</td>
<td></td>
</tr>
<tr>
<td>8048654:</td>
<td>mov %gs:0x14, %eax</td>
<td></td>
</tr>
<tr>
<td>8048655a:</td>
<td>mov %eax, 0xfffffffff8(%ebp)</td>
<td></td>
</tr>
<tr>
<td>804865d:</td>
<td>xor %eax, %eax</td>
<td></td>
</tr>
<tr>
<td>804865f:</td>
<td>lea 0xfffffffff4(%ebp), %ebx</td>
<td></td>
</tr>
<tr>
<td>8048662:</td>
<td>mov %ebx, (%esp)</td>
<td></td>
</tr>
<tr>
<td>8048665:</td>
<td>call 80485e1 &lt;gets&gt;</td>
<td></td>
</tr>
<tr>
<td>804866a:</td>
<td>call 804843c <a href="mailto:puts@plt">puts@plt</a></td>
<td></td>
</tr>
<tr>
<td>8048672:</td>
<td>mov 0xfffffffff8(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>8048675:</td>
<td>xor %gs:0x14, %eax</td>
<td></td>
</tr>
<tr>
<td>804867c:</td>
<td>je 8048683 &lt;echo+0x36&gt;</td>
<td></td>
</tr>
<tr>
<td>804867e:</td>
<td>call 804842c &lt;FAIL&gt;</td>
<td></td>
</tr>
<tr>
<td>8048683:</td>
<td>add $0x14, %esp</td>
<td></td>
</tr>
<tr>
<td>8048686:</td>
<td>pop %ebx</td>
<td></td>
</tr>
<tr>
<td>8048687:</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>8048688:</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Setting Up Canary

Before call to `gets`:

Stack Frame for `main`:
- Return Address
- Saved `%ebp`
- Saved `%ebx`
- Canary: `buf`

Stack Frame for `echo`:

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

**echo:**

```assembly
    . . .
    movl %gs:20, %eax  # Get canary
    movl %eax, -8(%ebp) # Put on stack
    xorl %eax, %eax    # Erase canary
    . . .
```
Checking Canary

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary
[3][2][1][0]

Stack Frame for echo

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

echo:
   . . .
   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

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) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00

Benign corruption!
(allow programmers to make silent off-by-one errors)
Summary

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

• Structures

• Memory layout

• Buffer overflow