Machine-Level Programming V: Unions and Memory layout

Slides adapted from Bryant and O’Hallaron
FAQ

Call conventions recap

• The first six integer or pointer argument are stored in RDI, RSI, RDX, RCX, R8, R9.
• The return value is stored in RAX.

Why there are no memory load/store instructions when accessing some variables?

• Compiler code generation
• Compiler optimizations

RBP and RSP

• RBP points to the current stack frame.
• RSP points to the top of the stack.
• Stack frame is not used when doing -O2 (and higher).
Today

- Structs vs. Unions
- Memory layout
C union vs. struct

- All members in a union start at the same location
  - i.e. only one member can contain a value at a given time
C union vs. struct

union my_union {
    unsigned char b[4];
    int x;
}
union my_union U;

sz = sizeof(U);  //sz??
U.b[3] = 0;
U.x = 0x01020304;
U.b[3] = ??

struct my_struct {
    unsigned char b[4];
    int x;
};
struct my_struct S;

sz = sizeof(S);  //sz??
S.b[3] = 0;
S.x = 0x01020304;
S.b[3] = ??
typedef union {
    float f;
    unsigned u;
} bit_float_t;

bit_float_t x;
x.f = 1.1;
//what is the value of x.u?

A. 1
B. 0x03f8cccccd
C. 0xf3ebdddc
D. 0x03000000
E. 0x00000001
Union Allocation

- Allocate according to largest element

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

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

<table>
<thead>
<tr>
<th></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>x+0</td>
<td>x+4</td>
<td>x+8</td>
<td>x+16</td>
<td>x+24</td>
<td></td>
</tr>
</tbody>
</table>
Unions in assembly

typedef union {
    char c;
    int i[2];
    double v;
} u_t;

int get_member(u_t *u,
    int first) {
    if (first) {
        return U->c;
    } else {
        return U->i[1];
    }
}

testl %esi, %esi
je .L2
movsbl (%rdi), %eax
ret
.L2:
movl 4(%rdi), %eax
ret
Today

- Structs vs. Unions
- Memory layout
### x86-64 Linux Memory Layout

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

**Heap**
- Dynamically allocated as needed
- When call `malloc()`, `calloc()`, `new()`

**Data**
- Statically allocated data
- E.g., global vars, `static` vars, string constants

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

Hex Address 400000 000000

Not drawn to scale
Memory Allocation Example

```c
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 */
    ...
}
```

Where does everything go?
x86-64 Example Addresses

address range $\sim 2^{47}$

local
p1  0x00007f7262a1e010
p3  0x00007f7162a1d010
p4  0x0000008359d120
p2  0x0000008359d010
big_array
  0x000000080601060
huge_array
  0x0000000601060
main() 0x0000000040060c
useless() 0x00000000400590
Segmentation Fault

- Each memory segment can be readable, executable, writable (or none at all)
- Segmentation fault occurs when program tries to access illegal memory
  - Read from segment with no permission
  - Write to read-only segments
Segmentation fault example

```c
char str1[100] = "hello world1";

int main ()
{
    char *str2 = "hello world2";
    printf("str1 %p str2 %p\n", str1, str2);
    str1[0] = 'H';
    str2[0] = 'H'
    ...
}
```
Not all Bad Memory Access lead to immediate segmentation

```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;
}
```

<table>
<thead>
<tr>
<th>fun(i)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(0)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(1)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(2)</td>
<td>3.1399998664856</td>
</tr>
<tr>
<td>fun(3)</td>
<td>2.00000061035156</td>
</tr>
<tr>
<td>fun(4)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(6)</td>
<td>Segmentation fault</td>
</tr>
</tbody>
</table>

- Result is system specific
Memory Referencing Bug Example

typedef struct {
    int a[2];
    double d;
} struct_t;

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

struct_t

Critical State

<table>
<thead>
<tr>
<th>Location accessed by fun(i)</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>5</td>
</tr>
<tr>
<td>?</td>
<td>4</td>
</tr>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>
Such problems are a BIG deal

- Generally called a “buffer overflow”
  - when exceeding the memory size allocated for an array

- Why a big deal?
  - It’s the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance

- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing