DISCUSSION – WEEK 3

TA: Mathanky
Email: mathanky04@ucla.edu
REVIEW OF WEEK 3

● Machine Level Programming - Data
  ○ Arrays
  ○ Structures

● Machine Level Programming - Advanced
  ○ Unions
  ○ 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];

int val[5];
double a[3];
char *p[3];
Array Accessing Example

```c
int get_digit
    (zip_dig z, int digit)
{
    return z[digit];
}
```

**IA32**

```assembly
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference (%rdi,%rsi,4)
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

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

```
| A[0][0] | ⋮ | ⋮ | A[0][C-1] |
| ⋮ | ⋮ | ⋮ | ⋮ |
| A[R-1][0] | ⋮ | ⋮ | A[R-1][C-1] |
```
Nested Array Row Access

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

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

![Diagram showing row vectors and their addresses]

- \( A[0] \)
- \( A[i] \)
- \( A[R-1] \)
- \( A[i][0] \)
- \( A[i][C-1] \)
- \( A[R-1][0] \)
- \( A[R-1][C-1] \)
- \( A+(i\times C\times 4) \)
- \( A+((R-1)\times C\times 4) \)
**Nested Array Row Access Code**

```c
int *get_pgh_zip(int index) {
    return pgh[index];
}
```

```assembly
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
lea pgh(,%rax,4),%rax # pgh + (20 * index)
```

- **Row Vector**
  - `pgh[index]` is array of 5 int's
  - Starting address `pgh + 20 * index`

- **Machine Code**
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`
Nested Array Element Access

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

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

![Diagram showing access to nested array elements](image)
Structure Representation

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

- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code
Structures & Alignment

- **Unaligned Data**

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

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K

```
  char c;  // 3 bytes
  int i[2];  // 4 bytes
  double v;  // 8 bytes
```

```
  p+0  p+4  p+8  p+16  p+24
  Multiple of 8  Multiple of 4  Multiple of 8
```
Specific Cases of Alignment (x86-64)

- 1 byte: char, ...
  - no restrictions on address

- 2 bytes: short, ...
  - lowest 1 bit of address must be 0₂

- 4 bytes: int, float, ...
  - lowest 2 bits of address must be 00₂

- 8 bytes: double, long, char *, ...
  - lowest 3 bits of address must be 000₂

- 16 bytes: long double (GCC on Linux)
  - lowest 4 bits of address must be 0000₂
Satisfying Alignment with Structures

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

- **Overall structure placement**
  - Each structure has alignment requirement $K$
    - $K = \text{Largest alignment of any element}$
  - Initial address & structure length must be multiples of $K$

- **Example:**
  - $K = 8$, due to `double` element

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

- For largest alignment requirement K
- Overall structure must be multiple of K

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
How many bytes would the following array declaration allocate on a 64-bit machine?

```c
char *arr[10][6];
```
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
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;
```
Using Union to Access Bit Patterns

typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

Same as (float) u?  Same as (unsigned) f?
**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: 00000000 40000000
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 */
    /* Some print statements ... */
}
```

Where does everything go?
x86-64 Example Addresses

address range \( \sim 2^{47} \)

local
0x00007ffe4d3be87c

p1
0x000007f7262a1e010

p3
0x000007f7162a1d010

p4
0x000000008359d120

p2
0x000000008359d010

big_array
0x0000000080601060

huge_array
0x00000000601060

main()
0x0000000040060c

useless()
0x00000000400590
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;
}

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

- 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

Explanation:

<table>
<thead>
<tr>
<th>Critical State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>struct_t</td>
</tr>
<tr>
<td>?</td>
<td></td>
</tr>
<tr>
<td>d7 ... d4</td>
<td></td>
</tr>
<tr>
<td>d3 ... d0</td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

Before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

20 bytes unused

[3] [2] [1] [0]  buf ← %rsp

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

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

Buffer Overflow Stack Example #1

After call to `gets`

Stack Frame for `call_echo`

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

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: 01234567890123456789012
01234567890123456789012

"01234567890123456789012\0"

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

After call to `gets`

Stack Frame for `call_echo`

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>40</td>
<td>00</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

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

“0123456789012345678901234\0”

Overflowed buffer and corrupted return pointer