Machine-Level Programming IV: Data Structures and Stack Buffer Overflow

Kai Shen

Basic Data Types
- Integer
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on software interpretation and manipulation
    | Intel | ASM  | Bytes | C        |
    |-------|------|-------|----------|
    | byte  | b    | 1     | (unsigned) char |
    | word  | w    | 2     | (unsigned) short |
    | double word | l   | 4     | (unsigned) int |
    | quad word | q   | 8     | (unsigned) long int (x86-64) |
- Floating Point
  - Stored & operated on in floating point registers
    | Intel | ASM  | Bytes | C           |
    |-------|------|-------|-------------|
    | Single | s   | 4     | float       |
    | Double | l   | 8     | double      |
    | Extended | t  | 10/12/16 | long double |

Outline: Data Structures
- Basic data types
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Optimizations
- Structures

Array Allocation
- $T A[l]$:
  - Array of data type $T$ and length $l$
  - Contiguously allocated region of $l \times \text{sizeof}(T)$ bytes

char string[12];
int val[5];
double a[3];
char *p[3];
Array Access

```
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 Accessing Example

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

- Register %edx contains starting address of array
- Register %eax contains array index
- Desired value at 4*%eax + %edx
- Use memory reference (%edx, %eax, 4)

Array Loop Example

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

```
# 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
```

Nested Array Data Layout

```
#define GCOUNT 4
zip_dig grp[GCOUNT] =
{{1, 5, 2, 0, 6},
 (1, 5, 2, 1, 3),
 (1, 5, 2, 1, 7),
 (1, 5, 2, 2, 1)};
```

- “zip_dig grp[4]” equivalent to “int grp[4][5]”
  - Variable grp: array of 4 elements, allocated contiguously
  - Each element is an array of 5 int’s, allocated contiguously
Nested Array Vector Access Code

```c
int *get_grp_zip(int index)
{
    return grp[index];
}
```

- **grp[index]**
  - Array of 5 int's
  - Starting address `grp+20*index`
- Machine code

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

Nested Array Element Access Code

```c
int get_grp_digit
(int index, int dig)
{
    return grp[index][dig];
}
```

- **Array elements**
  - `grp[index][dig]` is int
  - Address: `grp + 20*index + 4*dig`
- Machine code

```c
int *get_grp_zip(int index)
{
    return grp[index];
}
```

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

16 X 16 Matrix Access

```c
typedef int fix_matrix[N][N];

/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
    return a[i][j];
}
```

- **Array Elements**
  - Address `A + i*(N*K) + j*K`
  - `K = 4`

```c
movl 12(%ebp), %edx # i
sall $6, %edx # i*64
movl 16(%ebp), %eax # j
sall $2, %eax # j*4
addl 8(%ebp), %eax # a + j*4
movl (%eax,%edx), %eax # *(a+j*4+i*64)
```

n X n Matrix Access

```c
typedef int fix_matrix[N][N];

/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
    return a[i][j];
}
```

- **Array Elements**
  - Address `A + i*(n*K) + j*K`
  - `K = 4`

```c
movl 8(%ebp), %eax # index
leal (%eax,%eax,4),%eax # 5*index
leal grp(,%eax,4),%eax # offset 4*(5*index+dig)
```

```c
int get_grp_digit
(int index, int dig)
{
    return grp[index][dig];
}
```

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

```c
int *get_grp_zip(int index)
{
    return grp[index];
}
```

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

```c
typedef int fix_matrix[N][N];

/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
    return a[i][j];
}
```

```c
movl 12(%ebp), %edx # i
sall $6, %edx # i*64
movl 16(%ebp), %eax # j
sall $2, %eax # j*4
addl 8(%ebp), %eax # a + j*4
movl (%eax,%edx), %eax # *(a+j*4+i*64)
```

```c
movl 8(%ebp), %eax # index
leal (%eax,%eax,4),%eax # 5*index
leal grp(,%eax,4),%eax # offset 4*(5*index+dig)
```
Outline: Data Structures

- Basic data types
- Arrays
- Structures
  - Allocation
  - Access

Structure Allocation

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

Structure Access

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

A Case Study

- Memory layout and buffer overflow attacks
IA32 Linux Memory Layout

- Stack
  - Local variables, saved registers etc.
- Heap
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only

Internet Worm

- November, 1988
  - Internet Worm attacks thousands of Internet hosts through finger daemon.
  - How did it happen?
    - The finger daemon had a vulnerability – when an attacker supplied certain input string, a buffer overflowed on stack to allow the attacker overwrite function return address, run custom-supplied code, and take over the machine.

Instant Messenger War

- July 1999
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging (AIM) servers
- August 1999
  - Mysteriously, Messenger clients can no longer access AIM servers.
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes
    - At least 13 such skirmishes
  - How did it happen?
    - Note that AOL does not require reinstallation of its own AOL clients

Stack Buffer Overflow Exploits

- The Internet Worm and AOL/Microsoft War were both based on stack buffer overflow exploits!
  - Buffer on stack overflows, then overwrites stack data, then causes trouble or hijacks the machine
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;
  }
  ```

- Does not allow specification of 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);
}
```

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

Buffer Overflow Example #1

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

Before call to `gets`

- Stack Frame for caller
- 0xffffd688
- Saved ebp
- Saved ebx
- [3][2][1][0]
- Stack Frame for echo

Before call to `puts`

- Stack Frame for caller
- 0xffffd688
- Saved ebp
- Saved ebx
- [3][2][1][0]
- Stack Frame for echo

Input 1234567

- Stack Frame for caller
- 0xffffd688
- Saved ebp
- Saved ebx
- [3][2][1][0]
- Stack Frame for echo

Overflow buf, and corrupt %ebx, but no problem
- because %ebx isn’t used by caller
### Buffer Overflow Example #2

**Before call to gets**

- Stack Frame for caller: `0xffffd688`  
  - `08 04 85 10`  
  - `ff ff de 88`  
  - Saved `ebx`  
  - `Stack Frame for echo`  

**Input 12345678**

- Stack Frame for caller: `0xffffd688`  
  - `08 04 85 10`  
  - `ff ff d6 88`  

Base pointer corrupted

### Buffer Overflow Example #3

**Before call to gets**

- Stack Frame for caller: `0xffffd688`  
  - `08 04 85 10`  
  - `ff ff d6 88`  

**Input 123456789ABC**

- Stack Frame for caller: `0xffffd688`  
  - `08 04 85 10`  
  - `ff ff d6 88`  

Return address corrupted

### Malicious Use of Buffer Overflow

```c
void foo(){
    bar();
    ...
}
```

```c
int bar() {
    char buf[64];
    gets(buf);
    ...
    return ...
}
```

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

### Exploits Based on Buffer Overflows

- **Buffer overflows 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 kshen@cycle1.cs.rochester.edu`
    - Worm attacked fingerd server by sending phony argument:
      - `finger “exploit-code padding new-return-address”`
    - Exploit code: launch a shell on the victim machine (as the user of the fingerd process, which is root)
Exploits Based on Buffer Overflows

- Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines
- IM War
  - AOL exploited existing buffer overflow bug in AIM clients
  - Exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
- When Microsoft changed code to match signature, AOL changed signature (WITHOUT requiring reinstallation of its own AOL clients).

Avoiding Overflow Vulnerability

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

- 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

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`

```
unix> gdb bufdemo
(gdb) break echo
(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
unix> ./bufdemo-protected
Type a string:123
123
unix> ./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```
Setting Up Canary for Suspicious Data Structures

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

Checking Canary before Restoring Registers and Returning

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

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.