Machine Level Programming: Data and Advanced Concepts

Today

• Arrays
  • One-dimensional
  • Multi-dimensional (nested)
  • Structures
  • Memory layout
  • Buffer overflow

Array Allocation

• Basic Principle

  $T \mathbf{A}[L];$
  • Array of data type $T$ and length $L$
  • Contiguously allocated region of $L \times \text{sizeof}(T)$ bytes

Array Example

#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 };
Array Accessing Example

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

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

IA32

```asm
# %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
#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 }};
```

Nested Array Element Access Code

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

IA32

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

Structure Allocation

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

Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Structures
  - Memory layout
  - Buffer overflow

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
Structure Access

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

IA32 Assembly

```assembly
rdi = val
leal rax, [rdi]
```

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

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

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

```c
$esp 0xffffbcd0
p3 0x65586008
p1 0x55585008
p4 0x1904a110
p2 0x1904a008
ap2 0x18049760
beyond 0x08049744
big_array 0x18049780
huge_array 0x08049760
main() 0x080483c6
useless() 0x08049744
final malloc() 0x006be166
```

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);
}
```
Buffer Overflow Disassembly

```assembly
004850c5:   55              push    ebp
004850c6:   89 e5            mov     %esp,%ebp
004850c8:   53              push    %ebx
004850c9:   83 ec 14        sub     $0x14,%ebp
004850cc:   8d 5d ff ff     lea     0xffffffff(%ebp),%ebx
004850cf:   89 1c 24        mov     %ebx,%ebp
004850d1:   88 9a ff ff ff  call    0x804857f <gets>
004850d6:   88 05 ff ff ff  call    0x80483e4 <puts@plt>
004850df:   83 c4 14        add     $0x14,%ebp
00485100:   5b              pop     %ebx
00485101:   88 04 33       mov     %esp,%ebp
00485105:   c3              ret
```

call echo:

```assembly
004850b8:   e8 d5 ff ff ff  call    0x8048505 <echo>
004850c0:   c9              leave
004850c1:   c3              ret
```

Buffer Overflow Stack

Before call to gets

```
0x804857f:
buf                # Buffer on stack
ebp                # Saved ebp
esp                # Saved esp
prolog             # Saved prolog
```

Before call to gets

```
0xffffd678:
buf                # Buffer on stack
ebp                # Saved ebp
esp                # Saved esp
prolog             # Saved prolog
```

Buffer Overflow Example #2

```
Address: 0x804854b
Saved ebp: 0x8048505
Saved esp: 0x804850c
```

```
804850c5:   55              push    ebp
804850c6:   89 e5            mov     %esp,%ebp
804850c8:   53              push    %ebx
804850c9:   83 ec 14        sub     $0x14,%ebp
804850cc:   8d 5d ff ff     lea     0xffffffff(%ebp),%ebx
804850cf:   89 1c 24        mov     %ebx,%ebp
804850d1:   88 9a ff ff ff  call    0x804857f <gets>
804850d6:   88 05 ff ff ff  call    0x80483e4 <puts@plt>
804850df:   83 c4 14        add     $0x14,%ebp
80485100:   5b              pop     %ebx
80485101:   88 04 33       mov     %esp,%ebp
80485105:   c3              ret
```

```
804850c5:   55              push    ebp
804850c6:   89 e5            mov     %esp,%ebp
804850c8:   53              push    %ebx
804850c9:   83 ec 14        sub     $0x14,%ebp
804850cc:   8d 5d ff ff     lea     0xffffffff(%ebp),%ebx
804850cf:   89 1c 24        mov     %ebx,%ebp
804850d1:   88 9a ff ff ff  call    0x804857f <gets>
804850d6:   88 05 ff ff ff  call    0x80483e4 <puts@plt>
804850df:   83 c4 14        add     $0x14,%ebp
80485100:   5b              pop     %ebx
80485101:   88 04 33       mov     %esp,%ebp
80485105:   c3              ret
```

Buffer Overflow Example #3

```
Address: 0x804854b
Saved ebp: 0x8048505
Saved esp: 0x804850c
```

```
804850c5:   55              push    ebp
804850c6:   89 e5            mov     %esp,%ebp
804850c8:   53              push    %ebx
804850c9:   83 ec 14        sub     $0x14,%ebp
804850cc:   8d 5d ff ff     lea     0xffffffff(%ebp),%ebx
804850cf:   89 1c 24        mov     %ebx,%ebp
804850d1:   88 9a ff ff ff  call    0x804857f <gets>
804850d6:   88 05 ff ff ff  call    0x80483e4 <puts@plt>
804850df:   83 c4 14        add     $0x14,%ebp
80485100:   5b              pop     %ebx
80485101:   88 04 33       mov     %esp,%ebp
80485105:   c3              ret
```

```
804850c5:   55              push    ebp
804850c6:   89 e5            mov     %esp,%ebp
804850c8:   53              push    %ebx
804850c9:   83 ec 14        sub     $0x14,%ebp
804850cc:   8d 5d ff ff     lea     0xffffffff(%ebp),%ebx
804850cf:   89 1c 24        mov     %ebx,%ebp
804850d1:   88 9a ff ff ff  call    0x804857f <gets>
804850d6:   88 05 ff ff ff  call    0x80483e4 <puts@plt>
804850df:   83 c4 14        add     $0x14,%ebp
80485100:   5b              pop     %ebx
80485101:   88 04 33       mov     %esp,%ebp
80485105:   c3              ret
```
Malicious Use of Buffer Overflow

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

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

Stack after call to `gets()`

- `B` return address
- `A` stack frame
- `foo` stack frame
- `bar` stack frame
- `exploit code`
- `pad`
- `data written by `gets()``

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

```c
/* 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

Protected Buffer Disassembly `echo`:

```
0048644: 55 push %ebp
0048645: 89 e5 mov %esp,%ebp
0048646: 89 45 f8 mov %eax,0xfffffff8(%ebp)
0048649: 31 c0 xor %eax,%eax
004864b: 65 a1 14 00 00 00 mov %gs:0x14,%eax
0048651: 89 1c 24 mov %ebx,(%esp)
0048654: 804843c <puts@plt>
0048657: 80485e1 <gets>
004865a: 89 1c 24 mov %ebx,(%esp)
004865d: e8 a9 fd ff ff call 804842c <FAIL>
0048660: 74 05 je 8048683 <echo+0x36>
0048663: 80 48 05 00 00 00 add 0x48,%eax
0048669: 804842c <FAIL>
004866c: 804843c <puts@plt>
004866f: 80485e1 <gets>
0048672: 804864d: 55 push %ebp
...
```

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> 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
```

- 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

```
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> 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
```
Setting Up Canary

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

Checking Canary

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

Canary Example

```assembly
(void) break echo
(void) run
(void) ste p 3
(void) print \x \{(unsigned \*) $ebp - 2\}
0x3e37d00
```

Summary

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Structures
- Memory layout
- Buffer overflow