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

Kai Shen

1

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`

2

Outline: Data Structures

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

3

Array Allocation

T **A[*L*];**

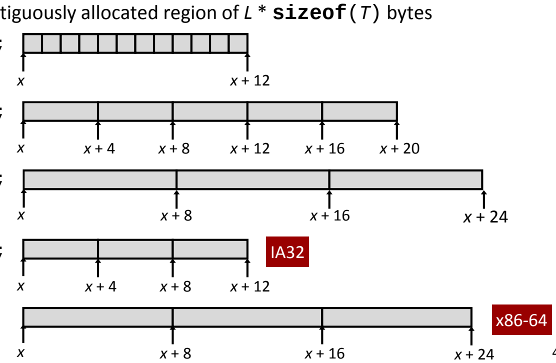
- Array of data type *T* and length *L*
- Contiguously allocated region of *L* * **`sizeof`**(*T*) bytes

`char string[12];`

`int val[5];`

`double a[3];`

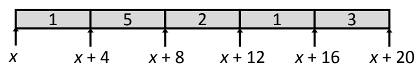
`char *p[3];`



4

Array Access

`int val[5];`

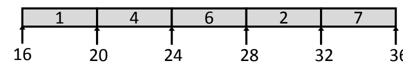


Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

5

Array Accessing Example

`zip_dig z;`




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

```
  # %edx = z
  # %eax = dig
  movl (%edx,%eax,4),%eax  # 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)`**

6

Array Loop Example

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

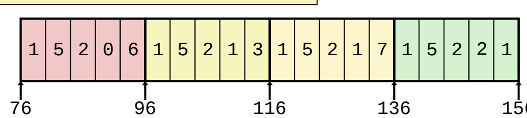
```
  # edx = z
  movl  $0, %eax            #   %eax = i
.L4:                        # loop:
  addl  $1, (%edx,%eax,4)   #   z[i]++
  addl  $1, %eax            #   i++
  cmpl  $5, %eax            #   i:5
  jne   .L4                 #   if !=, goto loop
```

7

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];`



- "**`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

8

Nested Array Vector Access Code

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

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

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

```
# %eax = index
leal (%eax,%eax,4),%eax    # 5 * index
leal grp(,%eax,4),%eax     # grp + (20 * index)
```

9

Nested Array Element Access Code

```
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
  - Computes address **grp + 4*((index+4*index)+dig)**

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

10

16 X 16 Matrix Access

```
#define N 16
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

```
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)
```

11

n X n Matrix Access

```
/* 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

```
movl  8(%ebp), %eax       # n
sall  $2, %eax            # n*4
movl  %eax, %edx          # n*4
imull 16(%ebp), %edx      # i*n*4
movl  20(%ebp), %eax      # j
sall  $2, %eax            # j*4
addl  12(%ebp), %eax      # a + j*4
movl  (%eax,%edx), %eax   # *(a + j*4 + i*n*4)
```

12

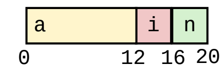
Outline: Data Structures

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

Structure Allocation

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

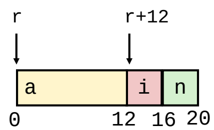
Memory Layout



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

Structure Access

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



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

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

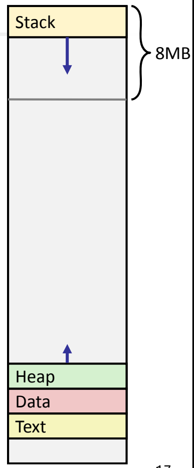
```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```


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



17

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.

18

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

19

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

20

String Library Code

- Implementation of Unix function `gets()`

```
/* 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

21

Vulnerable Buffer Code

```
/* 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
```

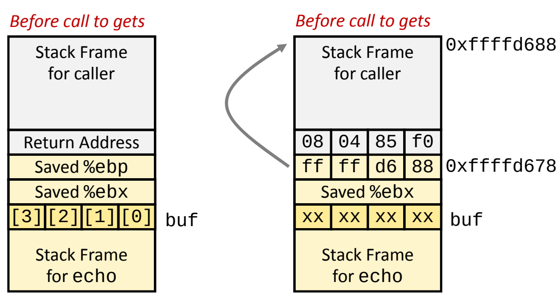
```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

22

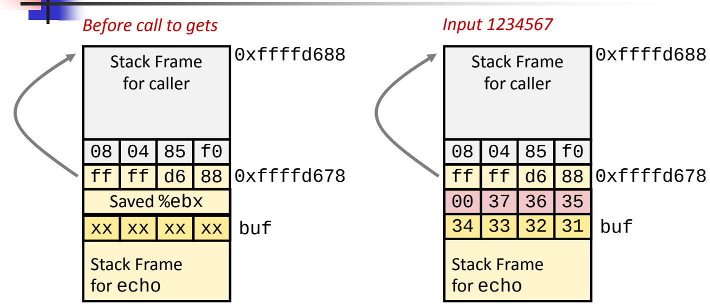
Buffer Overflow Stack Example

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



23

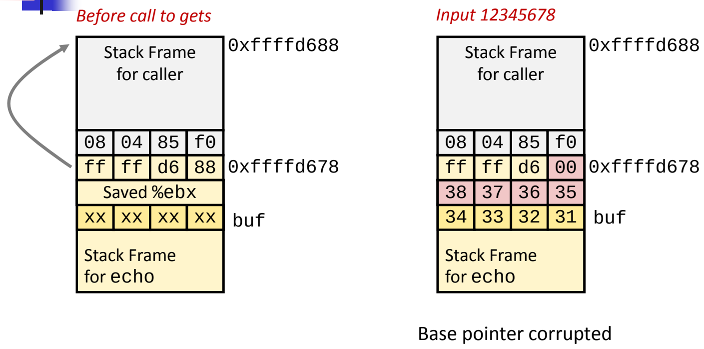
Buffer Overflow Example #1



Overflow buf, and corrupt %ebx, but no problem
- because %ebx isn't used by caller

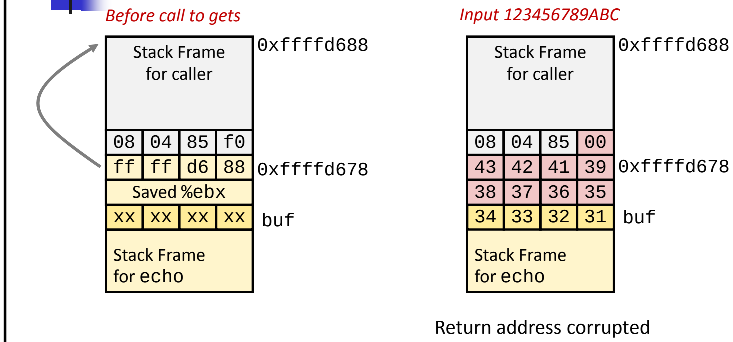
24

Buffer Overflow Example #2



25

Buffer Overflow Example #3



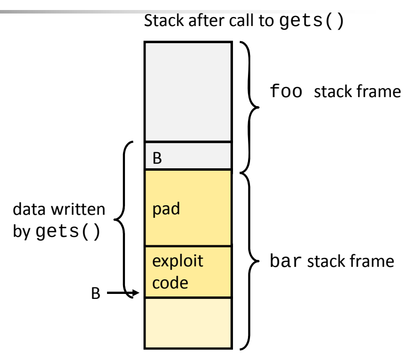
26

Malicious Use of Buffer Overflow

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

return address A

```
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

27

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 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)

28

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).

29

Avoiding Overflow Vulnerability

```
/* 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

30

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

```
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
```

31

Stack Canaries

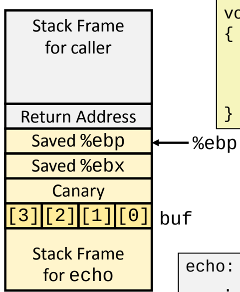
- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - **`-fstack-protector`**

```
unix>./bufdemo-protected
Type a string:123
123
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

32

Setting Up Canary for Suspicious Data Structures

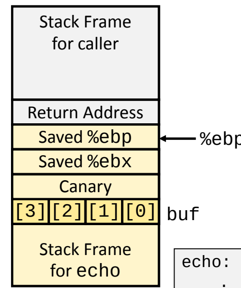



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

33

Checking Canary before Restoring Registers and Returning




```
/* 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:
    . . .
```

34

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.

35