Introduction to Computer Systems
15-213, fall 2009
9th Lecture, Sep. 28th

Instructors:
Majd Sakr and Khaled Harras
Last Time:

- **Structures**

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

- **Alignment**

- **Unions**

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

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

---

Memory Layout:

```
      i  a  p
0 4  16  20
```

```
c  i[0]  i[1]  4 bits
p+0 p+4 p+8
```

```
c  i[0]  i[1]  v
p+0 p+4 p+8 p+16 p+24
```

```
c  i[0]  i[1]  v
up+0 up+4 up+8
```
Summary

- **Arrays in C**
  - 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
Today

- Memory layout
- Buffer overflow, worms, and viruses
IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)

- **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

Upper 2 hex digits = 8 bits of address
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

address range $\sim 2^{32}$

- $\texttt{esp}$: 0xffffbcd0
- $\texttt{p3}$: 0x65586008
- $\texttt{p1}$: 0x55585008
- $\texttt{p4}$: 0x1904a110
- $\texttt{p2}$: 0x1904a008
- $\&\texttt{p2}$: 0x18049760
- beyond: 0x08049744
- big_array: 0x18049780
- huge_array: 0x08049760
- main(): 0x080483c6
- useless(): 0x08049744
- final malloc(): 0x006be166

malloc() is dynamically linked address determined at runtime

not drawn to scale
x86-64 Example Addresses

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

- \$rsp: 0x7fffffff8d1f8
- p3: 0x2aaabaadd010
- p1: 0x2aaaaaad0c010
- p4: 0x000011501120
- p2: 0x000011501010
- &p2: 0x000010500a60
- beyond: 0x000000500a44
- big_array: 0x000010500a80
- huge_array: 0x000000500a50
- main(): 0x0000005000510
- useless(): 0x0000005000500
- final malloc(): 0x00386ae6a170

malloc() is dynamically linked
address determined at runtime
C operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [ ] -</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- + - * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>/ %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; ^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>+= -= *= /= %= &amp;= ^= != &lt;&lt;= &gt;&gt;=</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

- `->` has very high precedence
- `( )` has very high precedence
- monadic `*` just below
C Pointer Declarations: Test Yourself!

```c
int *p

p is a pointer to int

int *p[13]

int *(p[13])

int **p

int (**p)[13]

int *f()

f is a function returning a pointer to int

int (*f)()

f is a pointer to a function returning int

int (**f())[13]()

int (*(*f())[13])()

int (**x[3])() [5]

x is an array[3] of pointers to functions returning pointers to array[5] of ints
```
C Pointer Declarations (Check out guide)

- `int *p`  
  p is a pointer to int

- `int *p[13]`  
  p is an array[13] of pointer to int

- `int *(p[13])`  
  p is an array[13] of pointer to int

- `int **p`  
  p is a pointer to a pointer to an int

- `int (*p)[13]`  
  p is a pointer to an array[13] of int

- `int *(*f)()`  
  f is a function returning a pointer to int

- `int (*f)()`  
  f is a pointer to a function returning int

- `int (*(f()))[13]()`  
  f is a function returning ptr to an array[13] of pointers to functions returning int

- `int (**x[3])() [5]`  
  x is an array[3] of pointers to functions returning pointers to array[5] of ints
Avoiding Complex Declarations

- Use `typedef` to build up the declaration

- Instead of `int (**x[3])(())[5]`:
  ```c
  typedef int fiveints[5];
  typedef fiveints* p5i;
  typedef p5i (*f_of_p5is)();
  f_of_p5is x[3];
  ```

- `x` is an array of 3 elements, each of which is a pointer to a function returning an array of 5 ints
Today

- Memory layout
- Buffer overflow, worms, and viruses
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 Unix functions
  - `strcpy`: Copies string 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);
}

int main()
{
    printf("Type a string:");
    echo();
    return 0;
}
```

```sh
unix> ./bufdemo
Type a string:1234567
1234567
```

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

```sh
unix> ./bufdemo
Type a string:123456789ABC
Segmentation Fault
```
Buffer Overflow Disassembly

080484f0  <echo>:
  80484f0:  55           push   %ebp
  80484f1:  89 e5        mov    %esp,%ebp
  80484f3:  53           push   %ebx
  80484f4:  8d 5d f8     lea    0xffffffff8(%ebp),%ebx
  80484f7:  83 ec 14     sub    $0x14,%esp
  80484fa:  89 1c 24     mov    %ebx,(%esp)
  80484fd:  e8 ae ff ff ff call   80484b0 <gets>
  8048502:  89 1c 24     mov    %ebx,(%esp)
  8048505:  e8 8a fe ff ff call   8048394 <puts@plt>
  804850a:  83 c4 14     add    $0x14,%esp
  804850d:  5b           pop    %ebx
  804850e:  c9           leave
  804850f:  c3           ret

  80485f2:  e8 f9 fe ff ff ff call   80484f0 <echo>
  80485f7:  8b 5d fc     mov    0xfffffffffc(%ebp),%ebx
  80485fa:  c9           leave
  80485fb:  31 c0        xor    %eax,%eax
  80485fd:  c3           ret
Buffer Overflow Stack

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
[3][2][1][0]
Stack Frame for echo

echo:
  pushl %ebp  # Save %ebp on stack
  movl %esp, %ebp
  pushl %ebx  # Save %ebx
  leal -8(%ebp),%ebx  # Compute buf as %ebp-8
  subl $20, %esp  # Allocate stack space
  movl %ebx, (%esp)  # Push buf on stack
  call gets  # Call gets
  . . .

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

Before call to `gets`

Stack Frame for `main`

Return Address

Saved `%ebp`

Stack Frame for `echo`

buf

08 04 85 f7

ff ff c6 58

Before call to `gets`

Stack Frame for `main`

0xffffffffc658

Stack Frame for `echo`

buf

80485f2: call 80484f0 <echo>

80485f7: mov 0xfffffffff(%ebp),%ebx # Return Point

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x $ebp
$1 = 0xffffffffc638
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffffff658
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f7
Buffer Overflow Example #1

Before call to gets

Stack Frame for main

08 04 85 f7
ff ff c6 58
xx xx xx xx

Stack Frame for echo

buf

Input 1234567

Stack Frame for main

08 04 85 f7
ff ff c6 58
00 37 36 35
34 33 32 31

buf

Overflow buf, but no problem
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

0xffffc638

0xffffc658

Stack Frame for echo

buf

Input 12345678

Stack Frame for main

0xffffc638

0xffffc658

Stack Frame for echo

buf

Base pointer corrupted

804850a:  83  c4  14  add     $0x14,%esp  # deallocate space
804850d:  5b          pop      %ebx  # restore %ebx
804850e:  c9          leave  # movl %ebp, %esp;  popl %ebp
804850f:  c3          ret    # Return
Buffer Overflow Example #3

**Before call to gets**

```
Stack Frame for main
0xffffc638
08 04 85 ff
0xffffc658
ff ff c6 58
buf
xx xx xx xx
```

```
Stack Frame for echo
0xffffc658
0xffffc638
```

**Input 12345678**

```
Stack Frame for main
0xffffc658
0xffffc638
0x00 00 00 00
43 42 41 39
38 37 36 35
34 33 32 31
buf
```

```
Stack Frame for echo
```

**Return address corrupted**

```
80485f2: call 80484f0 <echo>
80485f7: mov 0xffffffff (%ebp), %ebx # Return Point
```
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 ...;
}
```
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

/* 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
  - Add explicit “execute” permission

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xfffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xfffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xfffffc6a8
```
Worms and Viruses

- **Worm: A program that**
  - Can run by itself
  - Can propagate a fully working version of itself to other computers

- **Virus: Code that**
  - Add itself to other programs
  - Cannot run independently

- **Both are (usually) designed to spread among computers and to wreak havoc**