RECITATION 4 – THE STACK

15-213-m12
Rick Benua
The Stack

- Region of memory dedicated to local variables and arguments for all functions currently being executed
- Maintained using registers %esp and %ebp (on IA32)
  - %esp points to the top of the stack (actually the lowest address)
  - %ebp points to the base of the current “frame” – section of data associated with current function
- Modern compilers don’t need %ebp for this
  - Omitted by default on x86-64
  - %rbp can be another GPR
  - Can pass compiler flags to omit it on IA32
The Stack

- Memory on the stack can be accessed without checks
  - Callee reaches into caller’s frame to find arguments
  - Caller may pass a pointer into its stack frame to callee
    - (read as input, or write result, or both!)
- Callee may **NOT** return pointers into its stack
  - Stack space is “freed” upon return
  - Reused for next function call
Anatomy of a Stack Frame – IA32

| return address | %ebp | • Just before calling a function  
| old %ebp | %ebp | • arguments to next call pushed on stack in reverse order  
| callee-saved registers |  |  
| locals | %esp |  
| locals |  |  
| locals |  |  
| locals |  |  
| argument 2 |  |  
| argument 1 |  |  

- %ebp
- %esp
Anatomy of a Stack Frame – x86-64

<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>callee-saved registers</td>
</tr>
<tr>
<td>locals</td>
</tr>
<tr>
<td>locals</td>
</tr>
<tr>
<td>locals</td>
</tr>
<tr>
<td>argument 8</td>
</tr>
<tr>
<td>argument 7</td>
</tr>
</tbody>
</table>

- No base pointer – compiler uses offset from %rsp to find return value
- Arguments passed in registers, but can spill over onto the stack
Buffer Lab

• Out now!
• Due Tuesday
• More examination of programs
  • Create buffer overflow exploits for a known program
• READ THE HANDOUT
  • FOR THE LOVE OF GOD, READ THE ENTIRE HANDOUT
• Series of incrementally more complex exploits
Buffer Overflow

• Common idiom in code: Copy input from user into buffer, then process it

• Copy may not check length of input
  • Part of the point of this lab is to teach you to **not** do that

• Can reach beyond buffer into other parts of stack

• Strings generally written in from low – high addresses
  • “up” the stack, including into saved %ebp or return address!

• This is very bad.
Structures

- Structures combine sets of related values that can be passed around together.
- Values not necessarily contiguous in memory:
  - Each value must be aligned to its size.
  - Entire struct must be aligned to the largest constraint of any member.
- Each member is at a constant offset from the beginning of the struct.
Unions

- Structures store values “next to” each other
- Unions store values “on top of” each other
- Casting between types does conversion
- Union access does not
int main(){
    int x = 57;
    int y = 0;
    for(; y < 8; y++){
        x = (x + 1) / 2;
    }
    if(y != 1){
        return 1;
    }
    else{
        return 0;
    }
}
```c
struct{
    int i;
    char c[3];
    struct s *n;
    double d;
    short s;
} s;
```

<table>
<thead>
<tr>
<th>0x00</th>
<th>i</th>
<th>i</th>
<th>i</th>
<th>i</th>
<th>c[0]</th>
<th>c[1]</th>
<th>c[2]</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>0x10</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>0x18</td>
<td>s</td>
<td>s</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>