Buflab

Recitation - 09/20/2010
By sseshadr
Agenda

• Reminders
  – Bomblab should be finished up
  – Exam 1 is on Tuesday 09/28/2010

• Stack Discipline

• Buflab
  – One of you will get lucky

• Datalab Handouts
  – Overall style OK. See comments on ink
Stack?

• What is the stack?
  – It’s NOT
    • The memory regions returned by malloc()
    • The memory where your program itself is loaded
    • Where the bits for general purpose register are stored
    • Where the return value of a function is stored
  – It IS
    • Where you can often find a function’s local variables
    • How parameters are passed in 32-bit x86
    • Where the return ADDRESS is stored
    • A highly structured (easily corruptible) data structure essential to the execution of your code!
Virtual Address Space

The stack starts very close to 0xFFFFFFFF (for 32-bit) and grows DOWN.
Stack Discipline

- Each function has a stack frame
  - Local variables
  - Saved registers
  - Anything that function wants to put in its stack

- Functions CALL other functions
  - Arguments
  - Return address
  - Base pointer
Stack Discipline

Stack Frame

Arguments
- Return address
- Old %ebp
- Saved Registers +
- Local variables
- Argument Build
- Return address

...
Stack Discipline

• What happens when `ret` is executed?
  – Pop the stack and “`jmp`” to that address
  – In Java, think: `eip = stack.pop();`
• Where can I find the return address of my function?
  – `*(ebp + 4)`
• How can I find the second parameter to my function?
  – `*(ebp + 12) == the second parameter`
• How can I find the return address of the function that CALLED me?
  – Remember that `*ebp == old ebp`
  – `*(ebp + 4)`
• How are arguments pushed onto the stack?
  – Reverse order! Stacks are LIFO
Stack Discipline

• How will the “transition stack” look like for a function `foo` which makes the function call

\[
\text{printf(“str = \%s, num = \%d”, name, 16)}
\]

<table>
<thead>
<tr>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Stack frame for <code>foo</code></td>
</tr>
<tr>
<td>0x00000010</td>
<td>Arguments pushed in REVERSE order</td>
</tr>
<tr>
<td><code>&lt;address of name&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;address of format string in .rodata&gt;</code></td>
<td>All hard-coded strings are put into .rodata before runtime</td>
</tr>
<tr>
<td>return address (where EIP should return in <code>foo</code> after the printf call)</td>
<td>Return address</td>
</tr>
<tr>
<td><code>foo’s ebp</code></td>
<td>Old base pointer</td>
</tr>
<tr>
<td>...</td>
<td>Stack frame for <code>printf</code></td>
</tr>
</tbody>
</table>
Buflab

• It’s a hack.
  – Overflow the buffer to write over the return address

• We will go over how to solve the first phase.

• Whoever can answer this next question will get a head start
  – Only answer if you haven’t yet started buflab
**x86 Review Question:**

- $%eax$ at the start of this has the value $0x01000000$
- What will $%eax$ have after executing this code?

```
mov 4(%eax), %eax     \Rightarrow \quad eax = *(eax+4) = 0x10203040
lea 4(%eax), %eax     \Rightarrow \quad eax = (eax+4) = 0x10203044
```

<table>
<thead>
<tr>
<th>Memory</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00FFFFFFFC</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>0x01000000</td>
<td>0x01020304</td>
</tr>
<tr>
<td>0x01000004</td>
<td>0x10203040</td>
</tr>
<tr>
<td>0x10203040</td>
<td>0x12345678</td>
</tr>
<tr>
<td>0x10203044</td>
<td>0xBEEFBABE</td>
</tr>
</tbody>
</table>
Buflab Demo
Pick up your datalabs!