ANITA’S SUPER AWESOME RECITATION SLIDES
15/18-213: Introduction to Computer Systems
Stacks and Buflab, 23 Sept 2013
Anita Zhang
WHAT’S NEW (OR NOT)

- Bomb Lab is due Tuesday (tomorrow), 11:59 PM
  - Your late days are wasted here
  - Student: “But if you wait until the last minute, then it only takes a minute!”
    - Not (quite) true

- Buf Lab out Tuesday (tomorrow), 11:59 PM
  - Hacking the stack

- Stacks will be on the exams
  - They’re tough at first, but I believe in you 😊
Speaking of the Exam...

- **Midterm: Wed, 16 Oct – Sat, 19 Oct 2013**
- Covers everything up to, and including, caches.
  - Chapters 1-3 and 6 of textbook.
  - Up to and including Cache Lab.
  - Lectures up to and including Caches (1 Oct 2013).
- Recitation exam review the week of exam.
- “Read each chapter 3 times, work the practice problems, and do previous exams.”
  - Do enough midterms until you feel comfortable with the material (at least 5 recent ones).
    - Depending on the semester, caches can be found in Exam 2.
TO THOSE WHO WANT A COOL SHELL

- [http://www.contrib.andrew.cmu.edu/~anitazha/15213_tips.html](http://www.contrib.andrew.cmu.edu/~anitazha/15213_tips.html)
  - Scroll down to the part about “Shell of Choice”
  - Follow the directions
  - Your terminal will look something like this:

```
$ catshark 5:30 AM
$ hammerheadshark 6:00 AM
$ houndshark 6:30 AM
$ lemonshark 7:00 AM
$ makoshark 7:30 AM

---

< Do your best Anita! >
```

```
//  ^  ^
//    (oo)\___________)
//    (\|_______w
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
//     |        |
```
Journey Through Time

- Basic Assembly Review
  - Jump Tables vs. Sparse Switch
  - Terminology

- Stacks
  - IA32 Stack Discipline
  - Function Call Overview
  - Stack Walkthrough
  - Extras on x86(_64) stacks

- Buf Lab Quick Start
  - Essential Items of Business
  - Miscellany

- Demo...?
Assembly Coverage: Jump Tables

- Jump tables
  - Think of it as an array of addresses in memory
    - Use jump instructions to execute from these addresses
  - Using assembly it is possible to index into the array
  - Each entry of will hold addresses of instructions
JUMP TABLE EXAMPLE

- The tip-off is something like this:
  - `jmpq *0x400600(,%rax,8)`
    - Empty base means implied 0
    - `%rax` is the “index”
    - 8 is the “scale” (64-bit machine addresses are 8 bytes)
    - `*` indicates a dereference (like in C notation)
      - Like `leal`: does not do a dereference with parenthesis
  - Put it all together: “Jump to the address stored in the address 0x400600 + %rax*8”

- Using GDB (example output): `x/8g 0x400600`
  
  0x400600: 0x00000000000004004d1 0x00000000000004004c8
  0x400610: 0x00000000000004004c8 0x00000000000004004be
  0x400620: 0x00000000000004004c1 0x00000000000004004d7
  0x400630: 0x00000000000004004c8 0x00000000000004004be
Assembly Coverage: Sparse Switch

- Sparse switch vs. jump tables
  - Jump tables work if every entry has a jump location
  - Sparse switches cover cases where there are less densely packed cases
    - Does not make sense to allocate space for 100 entries if only 1 and 100 are used as cases
    - In the following example: uses labels to go to the next instruction
Sparse Switch Example

```c
int div111(int x) {
    switch(x) {
        case 0: return 0;
        case 111: return 1;
        case 222: return 2;
        case 333: return 3;
        case 444: return 4;
        case 555: return 5;
        case 666: return 6;
        case 777: return 7;
        case 888: return 8;
        case 999: return 9;
        default: return -1;
    }
}
```

```c
movl 8(%ebp),%eax # get x
cmpl $444,%eax # x:444
je L8
jg L16
cmpl $111,%eax # x:111
je L5
jg L17
testl %eax,%eax # x:0
je L4
jmp L14
```

```
L5:
    movl $1,%eax
    jmp L19
L6:
    movl $2,%eax
    jmp L19
L7:
    movl $3,%eax
    jmp L19
L8:
    movl $4,%eax
    jmp L19
    .
```
“In order to support general recursion, a language needs a way to allocate different activation records for different invocations of the same function. That way, local variables allocated in one recursive call can coexist with local variables allocated in a different call.” (credits to stack overflow)
DEFINITIONS AND CONVENTIONS

• Register
  - Some place in hardware that stores bits
    - Like boxes on the side of memory

• Caller save
  - Saved by the caller of a function
  - Before a function call, the caller must save any caller save register values it wants preserved

• Callee save
  - Saved by the callee of a function
  - The callee is required to save/restore values in these registers if it is using these registers in the function
Aside: Why both?

- Why do we have both caller and callee save?
  - Performance
  - Not all registers need to be saved
IA32 Registers

6 general purpose registers
  • Caller save
    o %eax, %ecx, %edx
    o Saved by the caller of a function
  • Callee save
    o %ebx, %edi, %esi
    o Saved by the callee of a function
**Special IA32 Registers**

- **Base Pointer**
  - `%ebp`
  - Points to the “bottom” of the stack frame
    - The location of old `%ebp` that gets pushed on entry

- **Stack Pointer**
  - `%esp`
  - Points to the “top” of the stack
    - *Usually* whatever was last pushed on the stack

- **Instruction Pointer (Program Counter)**
  - `%eip`
  - Points to the next instruction to be executed
IA32 TERMINOLOGY

Higher addresses (ie. 0xFFFFFFFF)  "bottom"

%esp ➔

"top"

Lower addresses (ie. 0x00000000)

Direction of stack growth
Aside: Technology Note (Again!)

- This class is (strictly) x86(_64)
  - Other architectures may have different conventions
    - May not use stacks at all (weird, I know)
  - Stacks grow down/ up depending on implementation
    - Very confusing to those new to stacks
Aside: Direction of Growth

- Stack direction REALLY doesn’t matter
  - Direction of growth is dependent on the processor
  - May be selectable for up/down
  - ...Or some other direction...?
BAM! CIRCULAR STACK!

SPARC (scalable processor architecture) Architecture
Aside: Direction of Growth

- Examples from StackOverflow
  - x86 - down
  - SPARC - in a circle
  - System z - in a linked list (down, at least for zLinux)
  - ARM - selectable
  - PDP11 - down
WHAT HAPPENS IN IA32: PUSH

- Pushing on the stack

In general, `pushl` translates to (in AT&T syntax):

- `subl $0x4, %esp`
- `movl src, (%esp)`
WHAT HAPPENS IN IA32: POP

- Popping off the stack

In general, `popl` translates to (in AT&T syntax):
- `movl (%esp), dest`
- `addl $0x4, %esp`
**Stack Frames WhatChamacallits?**

- Every function call gets a “stack frame”
- All the useful stuff can go on the stack!
  - Local variables (scalars, arrays, structs)
    - What the compiler couldn’t fit into registers
  - Callee/caller save registers
  - Temporary variables
  - Arguments
- Stacks can make recursion work!
- **Key idea:** “Storage for each *instance* of procedure call” (stolen out of 15-410 slides)
So THAT’S WHAT IT LOOKS LIKE…

- Increasing Addresses
- Frame Pointer \( %ebp \rightarrow \)
- Stack Pointer \( %esp \rightarrow \)
- \( \ldots \) Earlier Frames
- \( \ldots \)
- Argument n
- \( \ldots \)
- Argument 1
- Return Address
- Saved (old) \( %ebp \)
- Saved registers, local variables, and temporaries
- Argument build area
- Caller’s frame
- Current (callee) frame
Roles of a (Function) Caller

- Caller
  - Save (push) relevant caller save registers
  - Push arguments
  - Call function

- Caller after function return
  - “Remove” (add to %esp or pop) arguments
  - Restore (pop) saved caller save registers
Roles of a (Function) Callee

Callee

- Push `%ebp` (save stack frame)
- Copy (move) `%esp` into `%ebp`
- Save (push) callee save registers it wants to use

Callee before return

- Restore (pop) callee save registers previously saved
- Copy (move) `%ebp` into `%esp`
  - Moves stack pointer to the saved `%ebp`
- Restore (pop) `%ebp`
FUNCTION CALLS, OTHER OPERATIONS

- Implied operations
  - "call" implicitly pushes return address
    - Return address is always of the instruction after the call
  - "ret" implicitly pops return address into %eip
    - Becomes the next instruction to execute!
## Stack Frames in Action

<table>
<thead>
<tr>
<th>C Code</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main() {</td>
<td>08048394 &lt;main&gt;:</td>
</tr>
<tr>
<td>return addition(5, 6);</td>
<td>push %ebp</td>
</tr>
<tr>
<td>}</td>
<td>8048394:</td>
</tr>
<tr>
<td>int addition(int x, int y) {</td>
<td>8048395:</td>
</tr>
<tr>
<td>return x+y;</td>
<td>8048397:</td>
</tr>
<tr>
<td>}</td>
<td>804839a:</td>
</tr>
<tr>
<td></td>
<td>804839d:</td>
</tr>
<tr>
<td></td>
<td>80483a4:</td>
</tr>
<tr>
<td></td>
<td>80483a5:</td>
</tr>
<tr>
<td></td>
<td>80483ac:</td>
</tr>
<tr>
<td></td>
<td>80483b1:</td>
</tr>
<tr>
<td></td>
<td>80483b2:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**BREAKDOWN: ARGUMENT BUILD**

- **Build the arguments** *(note: 2 instructions are executed in this example)*

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp = 0x104</td>
<td>%esp = 0x104</td>
</tr>
<tr>
<td>%ebp = 0x200</td>
<td>%ebp = 0x200</td>
</tr>
<tr>
<td>%eip = 0x804839d</td>
<td>%eip = 0x80483ac</td>
</tr>
</tbody>
</table>

**Example Instructions**

- `main():
  movl $0x6,0x4(%esp)
  movl $0x5,(%esp)`

**Before Addresses**

- `%esp = 0x104`
- `%ebp = 0x200`
- `%eip = 0x804839d`
**BREAKDOWN: FUNCTION CALL**

- **Call the function**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp = 0x104</td>
<td>%esp = 0x100</td>
</tr>
<tr>
<td>%ebp = 0x200</td>
<td>%ebp = 0x200</td>
</tr>
<tr>
<td>%eip = 0x80483ac</td>
<td>%eip = 0x80483b3</td>
</tr>
</tbody>
</table>
**BREAKDOWN: CALLEE SETUP**

- **Stack frame set up** (note: 2 instructions are executed in this example)

  Before
  - `%esp = 0x100`
  - `%ebp = 0x200`
  - `%eip = 0x80483b3`

  After
  - `%esp = 0xFC`
  - `%ebp = 0xFC`
  - `%eip = 0x80483b6`

  **addition()**:  
  ```
  push %ebp
  mov %esp,%ebp
  ```

  **Flow**:
  - `%esp` moves to `0x100` (argument 2)
  - `%esp` moves to `0xFC` (argument 2)
  - `%esp` moves to `0x104` (argument 1)
  - `%esp` moves to `0x5` (argument 1)
  - `%esp` moves to `0x80483b1` (return address)
Breaking from the example: Kind of

- Accessing an argument

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Argument</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>0x6</td>
<td>0x6</td>
<td>0xC(%ebp)</td>
</tr>
<tr>
<td>0x104</td>
<td>0x5</td>
<td>0x5</td>
<td>0x8(%ebp)</td>
</tr>
<tr>
<td>0x100</td>
<td>0x80483b1</td>
<td>0x80483b1</td>
<td></td>
</tr>
<tr>
<td>0xFC</td>
<td>0x200</td>
<td>0x200</td>
<td></td>
</tr>
</tbody>
</table>

- In the current frame, arguments are accessed via references to %ebp
  - Notice how argument 1 is at 0x8(%ebp), not 0x4(%ebp)
## Let’s Review the Code Again

<table>
<thead>
<tr>
<th>C Code</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int main() {</strong>&lt;br&gt;  return addition(5, 6);&lt;br&gt;}&lt;br&gt;<strong>int addition(int x, int y)</strong>&lt;br&gt;  {&lt;br&gt;  return x+y;&lt;br&gt;}</td>
<td><strong>08048394 &lt;main&gt;:</strong>&lt;br&gt;  8048394: 55  push %ebp&lt;br&gt;  8048395: 89 e5  mov %esp,%ebp&lt;br&gt;  8048397: 83 e4 f0  and $0xfffffffff0,%esp&lt;br&gt;  804839a: 83 ec 10  sub $0x10,%esp&lt;br&gt;  804839d: c7 44 24 04 06 00 00  movl $0x6,0x4(%esp)&lt;br&gt;  80483a4: 00&lt;br&gt;  80483a5: c7 04 24 05 00 00 00  movl $0x5,(%esp)&lt;br&gt;  80483ac: e8 02 00 00 00  call 80483b3 &lt;addition&gt;&lt;br&gt;  80483b1: c9  leave&lt;br&gt;  80483b2: c3  ret</td>
</tr>
</tbody>
</table>
BREAKDOWN: PREPARING TO RETURN

Preparing to return from a function

Before
\[
\begin{align*}
%\text{esp} &= 0x\text{FC} \\
%\text{ebp} &= 0x\text{FC} \\
%\text{eip} &= 0x80483bf
\end{align*}
\]

After
\[
\begin{align*}
%\text{esp} &= 0x100 \\
%\text{ebp} &= 0x200 \\
%\text{eip} &= 0x80483c0
\end{align*}
\]
**BREAKDOWN: RETURN**

- Return from a function

Before:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0xFc</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x200</td>
</tr>
<tr>
<td>%eip</td>
<td>0x80483c0</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x200</td>
</tr>
<tr>
<td>%eip</td>
<td>0x80483b1</td>
</tr>
</tbody>
</table>

Diagram:

```
%esp→ 0x100
   0x80483b1 (return address)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>0x6</td>
</tr>
<tr>
<td>0x104</td>
<td>0x5</td>
</tr>
</tbody>
</table>

addition():

```

ret

(=equivalent to: popl %eip)
Stacks and Stuff on x86_64

- Arguments (≤ 6) are passed via registers
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Extra arguments passed via stack!
    - IA32 stack knowledge still matters!

- Don’t need %ebp as the base pointer
  - Compilers are smarter now

- Overall less stack use
  - Potentially better performance
AND FLOATING POINT?

- Floating point arguments are complicated
  - Out of the scope of this course
  - Some chips have a separate floating point stack

- Example of complication
  - x86_64 stack on function entry needs to be 16 byte aligned for floating point
  - And other potential issues you shouldn’t worry about
Buflab

- A series of exercises asking you to overflow the stack and change execution
  - You do this with inputs that are super long and write over key stack values
  - Incorrect inputs will not hurt your score
- Seminal paper on stack corruption
  - *Smashing the Stack for Fun and Profit*
**Basic Approach**

- Examine the provided C code/ disassembly
  - Disassembling
    - `> objdump -d bufbomb > outfile`
  - Don’t forget that GDB is still used for this lab!
- Find out how long to make your inputs
- Write exploits to divert program execution
- Profit
**Buflab Tools**

- `./makecookie andrewID`
  - Makes a unique “cookie” based on your Andrew ID
- `./hex2raw`
  - Use the hex generated from assembly to pass raw strings into `bufbomb`
  - Use with `-n` in the last stage
- `./bufbomb -t andrewID`
  - The actual program to attack
  - Always pass in with your Andrew ID so your score is logged
  - Use with `-n` in the last stage
HOW TO INPUT ANSWERS

- Put your byte code exploit into a text file
  - Then feed it through hex2raw

- Later stages: write (corruption) assembly
  - Compiling
    - gcc -m32 -c example.S
  - Get the byte codes
    - objdump -d example.o > outfile
  - Then feed it through hex2raw
Ways to Feed Byte Codes

- Option 1: Pipes
  - > cat exploitfile | ./hex2raw | ./bufbomb -t andrewID

- Option 2: Redirects
  - > ./hex2raw < exploitfile > exploit-rawfile
  - > ./bufbomb -t andrewID < exploit-rawfile

- Option 3: Redirects in GDB
  - > gdb bufbomb
  - (gdb) run -t andrewID < exploit-rawfile
**Potential Points of Failure**

- Don’t use byte value 0A in your exploit
  - ASCII for newline
  - `gets()` will terminate early if it sees this

- Multiple exploits submitted for the same level always takes the latest submission
  - So if you pass correctly once, but accidently pass the wrong exploit later, just pass the correct one again

- If you manage to execute your exploit....
  - GDB will say weird things
    - “Can’t access memory...” etc.
    - Just ignore it and keep going

- Don’t forget the –n flag on the last level
The writeup contains all the lab knowledge
- How to use the tools
- How to write corruption code
- Even tells you how to solve the level (at a high level)!

Please don’t ask questions answered by the writeup
- Or I will make this sad face: TT_TT

The writeup is on Autolab
- Couple links down from the handout
**A Lesson on Endianness**

- We’re working with little endian
  - Least significant byte is at the lower address

<table>
<thead>
<tr>
<th></th>
<th>Caller stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher addresses</td>
<td>...</td>
</tr>
<tr>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>Saved %ebp</td>
<td>← %ebp</td>
</tr>
<tr>
<td>Saved %ebx</td>
<td>← Potential way to detect stack corruption</td>
</tr>
<tr>
<td>Canary</td>
<td></td>
</tr>
<tr>
<td>MSB [7] [6] [5] [4]</td>
<td>buf string (each char is a byte)</td>
</tr>
<tr>
<td>[3] [2] [1] [0] LSB</td>
<td></td>
</tr>
<tr>
<td>Lower addresses</td>
<td></td>
</tr>
</tbody>
</table>

...
**Example of Endian in Buf Lab**

- **Example byte code input:**
  - 01 02 03 04
  - 05 06 07 08
  - 09 AA BB CC
  - 55 44 04 08

- **Little Endian**
  - Addresses will look as they normally do when they end up on the stack.
  - Here, value 0x08044455 reads as 0x08044455 on the stack.

<table>
<thead>
<tr>
<th>Higher addresses...</th>
<th>08 04 44 55</th>
<th>$\leftarrow$ Potentially overwritten return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC BB AA 09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08 07 06 05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04 03 02 01</td>
<td>$\leftarrow$ Input string address</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lower addresses</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
**Miscellany but Necessary**

- **Canaries**
  - Attempts to detect overrun buffers
  - Sits at the end of the buffer (array)
    - If the array overflows, *hopefully* we detect this with a change in the canary value....

- **NOP sleds**
  - The **nop** instruction means “no-op/ no operation”
    - In computer architecture it’s like “pipeline bubbles”
  - Used to “pad” instructions (or exploits!)
    - Place your exploits at the end of the **nop** sled
    - Allows you to be “sloppier” in providing the return address of your exploit
DEMO TIME (IF CLASS ISN’T OVER YET)

- Byte code format
- Byte code feeding
- Example assembly
- Compiling assembly
  - Not assembling
- Assembly to byte code
STOLEN CREDITS & QUESTIONS SLIDE

- xkcd: Tabletop Roleplaying
- StackOverflow: Supporting Recursion
- StackOverflow: Direction of Stack Growth
- Understanding the SPARC Architecture
- CS:APP p. 220 – Stack Frame Structure
- *Smashing the Stack for Fun and Profit*
- CS:APP p.262 – NOP sleds
- CS:APP p.263 – Stack Frame with a canary
- Double Mocha Latte Picture