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!”

- Buf Lab out Wednesday, 11:59 PM
  - Hacking the stack
  - One week long lab

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

- **Midterm: Week of 3 March 2014**
- Covers everything up to, and including, caches.
  - Chapters 1-3 and 6 of textbook.
  - Lectures up to and including Caches (18 Feb 2014).
- 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
  - **WARNING: THIS SITE IS MESSY AND OUTDATED**

```
< Do your best Anita! >
--------------------
\   ^__^   \_Ob_/_
(oo)
(____)
 ||-\_w_\_||
 ||     ||
 anitazha@houndshark ~]$ 
```
JOURNEY THROUGH TIME

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

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

- Demo...?
“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
    - %eax, %ecx, %edx
    - Saved by the caller of a function
  - Callee save
    - %ebx, %edi, %esi
    - 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
Higher addresses (ie. 0xFFFFFFFF)

Lower addresses (ie. 0x00000000)
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
ASYDE: 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 ➔
- Stack Pointer %esp ➔

<table>
<thead>
<tr>
<th>Earlier Frames</th>
<th>Current (callee) frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Argument build area</td>
</tr>
<tr>
<td>...</td>
<td>Saved registers, local variables, and temporaries</td>
</tr>
<tr>
<td>Argument n</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Argument 1</td>
<td></td>
</tr>
<tr>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>Saved (old) %ebp</td>
<td></td>
</tr>
</tbody>
</table>

%ebp ➔ Saved (old) %ebp ➔ Current (callee) frame ➔ Argument build area ➔ Increasing Addresses ➔ Earlier Frames ➔ ... ➔ ... ➔ Frame Pointer %ebp ➔ Stack Pointer %esp ➔
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>8048394: 55</td>
</tr>
<tr>
<td>}</td>
<td>push %ebp</td>
</tr>
<tr>
<td></td>
<td>8048395: 89 e5</td>
</tr>
<tr>
<td></td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>int addition(int x, int y)</td>
<td>8048397: 83 e4 f0</td>
</tr>
<tr>
<td>{</td>
<td>and $0xfffffffff0,%esp</td>
</tr>
<tr>
<td>return x+y;</td>
<td>804839a: 83 ec 10</td>
</tr>
<tr>
<td>}</td>
<td>sub $0x10,%esp</td>
</tr>
<tr>
<td></td>
<td>804839d: c7 44 24 04 06 00 00</td>
</tr>
<tr>
<td></td>
<td>movl $0x6,0x4(%esp)</td>
</tr>
<tr>
<td></td>
<td>80483a4: 00</td>
</tr>
<tr>
<td></td>
<td>80483a5: c7 04 24 05 00 00 00</td>
</tr>
<tr>
<td></td>
<td>movl $0x5,(esp)</td>
</tr>
<tr>
<td></td>
<td>80483ac: e8 02 00 00 00</td>
</tr>
<tr>
<td></td>
<td>call 80483b3 &lt;addition&gt;</td>
</tr>
<tr>
<td></td>
<td>80483b1: c9</td>
</tr>
<tr>
<td></td>
<td>leave</td>
</tr>
<tr>
<td></td>
<td>80483b2: c3</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td>080483b3 &lt;addition&gt;:</td>
</tr>
<tr>
<td></td>
<td>80483b3: 55</td>
</tr>
<tr>
<td></td>
<td>push %ebp</td>
</tr>
<tr>
<td></td>
<td>80483b4: 89 e5</td>
</tr>
<tr>
<td></td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>80483b6: 8b 45 0c</td>
</tr>
<tr>
<td></td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>80483b9: 8b 55 08</td>
</tr>
<tr>
<td></td>
<td>mov 0x8(%ebp),%edx</td>
</tr>
<tr>
<td></td>
<td>80483bc: 8d 04 02</td>
</tr>
<tr>
<td></td>
<td>lea (%edx,%eax,1),%eax</td>
</tr>
<tr>
<td></td>
<td>80483bf: c9</td>
</tr>
<tr>
<td></td>
<td>leave</td>
</tr>
<tr>
<td></td>
<td>80483c0: c3</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>
BREAKDOWN: ARGUMENT BUILD

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

![Argument Build Diagram]

**Before**
- ESP = 0x104
- EBP = 0x200
- EIP = 0x804839d

**After**
- ESP = 0x104
- EBP = 0x200
- EIP = 0x80483ac

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

0x6 (argument 2)
0x5 (argument 1)
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>

main():
call 80483b3 <addition>

0x6 (argument 2)
0x5 (argument 1)
0x80483b1 (return address)
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`
- `%esp = 0x200`
- `%eip = 0x80483b1`
Break From the Example.. Kind of

- Accessing an argument

<table>
<thead>
<tr>
<th>Address</th>
<th>Argument</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>0x6</td>
<td>0xC(%ebp)</td>
</tr>
<tr>
<td>0x104</td>
<td>0x5</td>
<td>0x8(%ebp)</td>
</tr>
<tr>
<td>0x100</td>
<td>0x80483b1</td>
<td></td>
</tr>
<tr>
<td>0xFC</td>
<td>0x200</td>
<td>(Prev. %ebp)</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><code>int main() {</code></td>
<td>08048394 &lt;main&gt;:</td>
</tr>
<tr>
<td><code>return addition(5, 6);</code></td>
<td>8048394:      55</td>
</tr>
<tr>
<td><code>}</code></td>
<td>push %ebp</td>
</tr>
<tr>
<td></td>
<td>8048395:      89 e5</td>
</tr>
<tr>
<td></td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td><code>int addition(int x, int y)</code></td>
<td>8048397:      83 e4 f0</td>
</tr>
<tr>
<td><code>{</code></td>
<td>and $0xffffffff0,%esp</td>
</tr>
<tr>
<td><code>return x+y;</code></td>
<td>804839a:      83 ec 10</td>
</tr>
<tr>
<td><code>}</code></td>
<td>sub $0x10,%esp</td>
</tr>
<tr>
<td></td>
<td>804839d:      c7 44 24 04 06 00 00</td>
</tr>
<tr>
<td></td>
<td>movl $0x6,0x4(%esp)</td>
</tr>
<tr>
<td></td>
<td>80483a4:      00</td>
</tr>
<tr>
<td></td>
<td>80483a5:      c7 04 24 05 00 00 00</td>
</tr>
<tr>
<td></td>
<td>movl $0x5,(%esp)</td>
</tr>
<tr>
<td></td>
<td>80483ac:      e8 02 00 00 00 00</td>
</tr>
<tr>
<td></td>
<td>call 80483b3 &lt;addition&gt;</td>
</tr>
<tr>
<td></td>
<td>80483b1:      c9</td>
</tr>
<tr>
<td></td>
<td>leave</td>
</tr>
<tr>
<td></td>
<td>80483b2:      c3</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td>080483b3 &lt;addition&gt;:</td>
</tr>
<tr>
<td></td>
<td>80483b3:      55</td>
</tr>
<tr>
<td></td>
<td>push %ebp</td>
</tr>
<tr>
<td></td>
<td>80483b4:      89 e5</td>
</tr>
<tr>
<td></td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>80483b6:      8b 45 0c</td>
</tr>
<tr>
<td></td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>80483b9:      8b 55 08</td>
</tr>
<tr>
<td></td>
<td>mov 0x8(%ebp),%edx</td>
</tr>
<tr>
<td></td>
<td>80483bc:      8d 04 02</td>
</tr>
<tr>
<td></td>
<td>lea (%edx,%eax,1),%eax</td>
</tr>
<tr>
<td></td>
<td>80483bf:      c9</td>
</tr>
<tr>
<td></td>
<td>leave</td>
</tr>
<tr>
<td></td>
<td>80483c0:      c3</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>
**BREAKDOWN: PREPARING TO RETURN**

- Preparing to return from a function

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp = 0xF0C</td>
<td>%esp = 0x100</td>
</tr>
<tr>
<td>%ebp = 0xF0C</td>
<td>%ebp = 0x200</td>
</tr>
<tr>
<td>%eip = 0x80483bf</td>
<td>%eip = 0x80483c0</td>
</tr>
</tbody>
</table>
**BREAKDOWN: RETURN**

- Return from a function

**Before**

- `%esp = 0xFC`
- `%ebp = 0x200`
- `%eip = 0x80483c0`

**After**

- `%esp = 0x104`
- `%ebp = 0x200`
- `%eip = 0x80483b1`

---

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%esp = 0xFC</code></td>
<td><code>%esp = 0x104</code></td>
</tr>
<tr>
<td><code>%ebp = 0x200</code></td>
<td><code>%ebp = 0x200</code></td>
</tr>
<tr>
<td><code>%eip = 0x80483c0</code></td>
<td><code>%eip = 0x80483b1</code></td>
</tr>
</tbody>
</table>
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
Buflab

- 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>Higher addresses</th>
<th>Caller stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</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></td>
</tr>
<tr>
<td>Canary</td>
<td>↯ Potential way to detect stack corruption</td>
</tr>
<tr>
<td><strong>LSB</strong> [3] [2] [1] [0]</td>
<td>buf string (each char is a byte)</td>
</tr>
<tr>
<td>...</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></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Potentially overwritten return address</td>
</tr>
<tr>
<td>08 04 44 55</td>
<td></td>
</tr>
<tr>
<td>CC BB AA 09</td>
<td>Input string address</td>
</tr>
<tr>
<td>08 07 06 05</td>
<td></td>
</tr>
<tr>
<td>04 03 02 01</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Lower addresses</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lower addresses</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</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)

- Walking stacks
- 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