Announcements

• Clickers available through the bookstore and TAP
• Lab 0 due today
• Reading quizzes count from this week
• Midterm dates on edstem later today
Reading Quiz
Today

• Design Principles of Security
• Software Vulnerabilities
• Recap functions and the stack
• Recap assembly instructions
• Stack Buffer Overflow
Last Class: Design Principles of Security

• Least Privilege
• Use Fail-Safe Defaults
• Separation of Privilege/Separation of responsibility
• Defense in Depth
• Complete Mediation: check access to every object
• Security *not* through obscurity
• Design Security as a core principal
• Keep it simple silly
• Ease of use

-Saltzer, J. “Protection and the Control of Information Sharing in MULTICS”, CACM - 1974
Defense in Depth

• The notion of layering multiple types of protection together

• e.g., the Theodosian Walls of Constantinople:
  • Moat -> wall -> depression -> even bigger wall
  • Idea: attacker needs to breach all the defenses to gain access

• But defense in depth isn't free
  • You are throwing more resources at the problem
Password authentication

• People have a hard time remembering multiple strong passwords, so they reuse them on multiple sites

• Consequence: security breach of one site causes account compromise on other sites

• Solution: password manager
  • Remember one strong password, which unlocks access to site passwords

• Solution: two-factor authentication
  • Need both correct password and separate device to access account

• Free advice: to protect yourself, use a password manager and two-factor authentication
Least Privilege

- Every program and every user of the system should operate using the least set of privileges necessary to complete the job
- A subject should be given only those privileges necessary to complete its task
  - Function, not identity, controls
  - Rights added as needed, discarded after use
  - Minimal protection domain
Does this follow the principle of least privilege?

A. Yes
B. No
C. Maybe (Be prepared to explain)
Ensuring Complete Mediation

• To secure access to some capability/resource, construct a reference monitor
  • Single point through which all access must occur
    • E.g.: a network firewall
    • Desired properties: • Un-bypassable (“complete mediation”) •
      • Tamper-proof (is itself secure)
      • Verifiable (correct)
  • One subtle form of reference monitor flaw concerns race conditions
A Failure of Complete Mediation

Every security-relevant action must be checked for authenticity, integrity and authorization
Time of Check to Time of Use Vulnerability: Race Condition

procedure withdraw(w)
    // contact central server to get balance
    1. let b := balance

    2. if b < w, abort

    // contact server to set balance
    3. set balance := b - w

    4. dispense $w to user

Suppose that here an attacker arranges to suspend first call, and calls withdraw again concurrently.

TOCTTOU = Time of Check To Time of Use
Security Reviews

• Least Privilege
• Use Fail-Safe Defaults
• Separation of Privilege/Separation of responsibility
• Defense in Depth
• Complete Mediation: check access to every object
• Security *not* through obscurity
• Design Security as a core principal
• Keep it simple silly
• Ease of use
When is a program secure?

A. When it does what we want it to do
B. When we ensure that bad inputs do not result in unintended functionality
C. We need B + some more safeguards (be prepared to explain)
D. We can never have a secure program
Software Security

• Secure design and implementation
• Popular approach to software: black box approach
• Build defenses around vulnerable software – easily circumvented
When is a program secure?

• Formal approach: When it does exactly what it should
  • not more
  • not less

• But how do we know what it is supposed to do?
When is a program secure?

- Formal approach: When it does exactly what it should
  - not more
  - not less
- But how do we know what it is supposed to do?
  - somebody tells us (do we trust them?)
  - we write the code ourselves (what fraction of s/w have you written?)
When is a program secure?

• Pragmatic approach: when it doesn’t do bad things
• Often easier to specify a list of “bad” things:
  • delete or corrupt important files (integrity)
  • crash my system (availability)
  • send my password over the internet (confidentiality)
  • send phishing email
When is a program secure?

• But .. what if the program doesn’t do bad things, but could?

• is it secure?
Weird machines

• complex systems contain unintended functionality

• attackers can trigger this unintended functionality
  • i.e. they are exploiting vulnerabilities
What is a software vulnerability?

• A bug in a program that allows an unprivileged user capabilities that should be denied to them.

• There are a lot of types of vulnerabilities
  • bugs that violate “control flow integrity”
  • why? lets attacker run code on your computer!

• Typically these involve violating assumptions of the programming language or its run-time
Exploiting vulnerabilities (the start)

• Dive into low level details of how exploits work
  • How can a remote attacker get a victim program to execute their code?

• Threat model: victim code is handling input that comes from across a security boundary
  • what are examples of this?

• Security policy: want to protect integrity of execution and confidentiality of data from being compromised by malicious and highly skilled users of our system.
Stack buffer overflows

• Understand how buffer overflow vulnerabilities can be exploited
• Identify buffer overflows and assess their impact
• Avoid introducing buffer overflow vulnerabilities
• Correctly fix buffer overflow vulnerabilities
Buffer Overflows

• An anomaly that occurs when a program writes/reads data beyond the boundary of a buffer

• Canonical software vulnerability
  • ubiquitous in system software
  • OSes, web servers, web browsers

• If your program crashes with memory faults, you probably have a buffer overflow vulnerability
Common Vulnerabilities and Exposures (CVE): security flaw that is publicly known

Critical Systems are written in C/C++
- OS kernels
- High-performance servers
  - Apache, MySQL
- Embedded Systems
  - IoT devices, “smart” vehicles, the MARs rover..

https://nvd.nist.gov/vuln/search
CS 31 Recap
Memory

• Abstraction goal: make every process think it has the same memory layout.
  • MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.

• Reality: there’s only so much memory to go around, and no two processes should use the same (physical) memory addresses.

OS (with help from hardware) will keep track of who’s using each memory region.
**Virtual (logical) Memory:** The abstract view of memory given to processes. Each process gets an independent view of the memory.

**Physical Memory:** The contents of the hardware (RAM) memory. Managed by OS. Only ONE of these for the entire machine!

**Address Space:** Range of addresses for a region of memory.

The set of available storage locations.

**Virtual address space (VAS):** fixed size.

(Determined by amount of installed RAM.)
Memory

• Behaves like a big array of bytes, each with an address (bucket #).

• By convention, we divide it into regions.

• The region at the lowest addresses is usually reserved for the OS.
NULL: A special pointer value.

• NULL is equivalent to pointing at memory address 0x0. This address is NEVER in a valid segment of your program’s memory.
  • This guarantees a segfault if you try to deref it.
  • Generally a good idea to initialize pointers to NULL.
What happens if we launch an attack where we load an instruction to execute at 0x0

A. Nothing will happen, this region is mapped to the NULL pointer, which does not have any effect
B. There will be some effect, but not necessarily devastating
C. This will have a devastating effect.
Memory - Text

• After the OS, we store the program’s code.

• Instructions generated by the compiler.
Memory – (Static) Data

• Next, there’s a fixed-size region for static data.

• This stores static variables that are known at compile time.
  • Global variables
Memory - Stack

• At high addresses, we keep the stack.

• This stores local (automatic) variables.
  • The kind we’ve been using in C so far.
  • e.g., int x;
Memory - Stack

• The stack grows upwards towards lower addresses (negative direction).

• Example: Allocating array
  • int array[4];
Memory - Heap

- The heap stores dynamically allocated variables.
- When programs explicitly ask the OS for memory, it comes from the heap.
  - `malloc()` function
Instructions in Memory

0x0

Operating system

Text

Data

Heap

Stack

0xFFFFFFFF

funcA:
... 
call funcB
...

funcB:
pushl %ebp
movl %esp, %ebp
...

Function A

Function B

...
Process memory layout

.text
  • Machine code of executable

.data
  • Global initialized variables

.bss
  • Below Stack Section
    global uninitialized vars

heap
  – Dynamic variables

stack
  – Local variables
  – Function call data

Env
  – Environment variables
  – Program arguments

0x0

Operating system
 .text
 .data
 .bss

Heap

Stack

X:

Command line arguments

Environment variables

0xFFFFFFFF
Process memory layout

.text
  • Machine code of executable

.data
  • Global initialized variables

.bss
  • Below Stack Section
    global uninitialized vars

heap
  – Dynamic variables

stack
  – Local variables
  – Function call data

Env
  – Environment variables
  – Program arguments

    int i = 0;
    int main()
    {
      char *ptr = malloc(sizeof(int));
      char buf[1024]
      int j;
      static int y;
    }
Process memory layout

.text
  • Machine code of executable
.data
  • Global initialized variables
.bss
  • Below Stack Section
global uninitialized vars
heap
  – Dynamic variables
stack
  – Local variables
  – Function call data
Env
  – Environment variables
  – Program arguments

int i = 0;
int main()
{
  char *ptr = malloc(sizeof(int));
  char buf[1024]
  int j;
  static int y;
}

• i -> data segment
• ptr -> stack
  • data allocated on heap
• buf -> stack
• j -> stack
• y -> bss
X86: The De Facto Standard

- Extremely popular for desktop computers
- Alternatives
  - ARM: popular on mobile
  - MIPS: very simple
  - Itanium: ahead of its time
- CISC
  - 100 distinct opcodes
- Register poor
  - 8 registers of 32 bits
  - only 6 general purpose
- instructions are variable length
  - not aligned at 4 byte boundaries
- lots of backward compatibilities
  - defined in late 70s
  - exploit code that noone pays attention to
- we will use 32 bit because its more convenient.
Compilation Steps (.c to a.out)

You can see the results of intermediate compilation steps using different gcc flags.

- **C program** (p1.c) → **Compiler** (gcc -m32 -S)
- **Assembly program** (p1.s) → **Assembler** (gcc -c (or as = gcc’s assembler))
- **Object code** (p1.o) → **Linker** (gcc (or ld))
- **Executable code** (a.out)
  - Other object files (p2.o, p3.o, ...)
  - Library obj. code (libc.a)

Machine code instructions.
Machine Code

Binary (0’s and 1’s) Encoding of ISA Instructions
- some bits: encode the instruction (opcode bits)
- others encode operand(s)
  (eg) 01001010 \textbf{opcode} operands
  01 001 010
  ADD %r1 %r2

- different bits fed through different CPU circuitry:
Assembly Code

- **C program (p1.c)**
- **Assembly program (p1.s)**
- **Object code (p1.o)**
- **Executable code (a.out)**

**Compiler:** (gcc -m32 -S)

**Assembler:** (gcc -c (or as = gcc’s assembler))

**Linker:** (gcc (or ld))

**Human Readable Form of Machine Code**
What is “assembly”? 

Assembly is the “human readable” form of the instructions a machine can understand.

```
push %ebp
mov %esp, %ebp
sub $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), %eax
addl %eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

```
objdump -d a.out
```
### Object / Executable / Machine Code

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Machine Code (Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push %ebp</code></td>
<td>55</td>
</tr>
<tr>
<td><code>mov %esp, %ebp</code></td>
<td>89 E5</td>
</tr>
<tr>
<td><code>sub $16, %esp</code></td>
<td>83 EC 10</td>
</tr>
<tr>
<td><code>movl $10, -8(%ebp)</code></td>
<td>C7 45 F8 0A 00 00 00</td>
</tr>
<tr>
<td><code>movl $20, -4(%ebp)</code></td>
<td>C7 45 FC 14 00 00 00</td>
</tr>
<tr>
<td><code>movl -4(%ebp), $eax</code></td>
<td>8B 45 FC</td>
</tr>
<tr>
<td><code>addl $eax, -8(%ebp)</code></td>
<td>01 45 F8</td>
</tr>
<tr>
<td><code>movl -8(%ebp), %eax</code></td>
<td>B8 45 F8</td>
</tr>
<tr>
<td><code>leave</code></td>
<td>C9</td>
</tr>
</tbody>
</table>

Almost a 1-to-1 mapping to Machine Code  
Hides some details like num bytes in instructions
Object / Executable / Machine Code

**Assembly**

```assembly
push %ebp
mov %esp, %ebp
sub $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), %eax
addl %eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

```c
int main() {
    int a = 10;
    int b = 20;
    a = a + b;
    return a;
}
```
Processor State in Registers

- Information about currently executing program
  - Temporary data (%eax - %edi)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%eip, ...)
  - Status of recent tests %EFLAGS
    (CF, ZF, SF, OF)

General purpose registers

- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

Current stack top
- %esp
- %ebp

Current stack frame
- %edi

Program Counter (PC)
- %eip

Condition codes
- CF
- ZF
- SF
- OF
General purpose Registers

Six are for instruction operands

Can store 4 byte data or address value

The low-order 2 bytes \( %ax \) is the low-order 16 bits of \( %eax \)

Two low-order 1 bytes \( %al \) is the low-order 8 bits of \( %eax \)

May see their use in ops involving shorts or chars

<table>
<thead>
<tr>
<th>Register name</th>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
<th>%eip</th>
<th>%EFLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bits: 31 16 15 8 7 0
Assembly Programmer’s View of State

Registers:
- PC: Program counter (%eip)
- Condition codes (%EFLAGS)
- General Purpose (%eax - %ebp)

Memory:
- Byte addressable array
- Program code and data
- Execution stack

CPU

32-bit Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td>next instr addr (PC)</td>
</tr>
<tr>
<td>%EFLAGS</td>
<td>cond. codes</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Program:</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>instrs</td>
</tr>
<tr>
<td></td>
<td>stack</td>
</tr>
<tr>
<td>0xffffffff</td>
<td></td>
</tr>
</tbody>
</table>
Types of IA32 Instructions

• Data movement
  • Move values between registers and memory
  • Example: `movl`

• Load: move data from memory to register

• Store: move data from register to memory
Instruction Syntax

Examples:

`subl $16, %ebx`

`movl (%eax), %ebx`

- Instruction ends with data length
- Opcode, src, dst
- Constants preceded by $
- Registers preceded by %
- Indirection uses ( )
Addressing Mode: Memory

• Accessing memory requires you to specify which address you want.
  • Put address in a register.
  • Access with () around register name.

• `movl (%ecx), %eax`
  • Use the address in register ecx to access memory, store result in register eax
Addressing Mode: Memory

- **movl (%ecx), %eax**
  
  - Use the address in register ecx to access memory, store result in register eax

### CPU Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
</tbody>
</table>

Slide 58
Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register `ecx` to access memory, store result in register `eax`

**CPU Registers**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1. Index into memory using the address in `ecx`.
Addressing Mode: Memory

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%eax</code></td>
<td>42</td>
</tr>
<tr>
<td><code>%ecx</code></td>
<td>0x1A68</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Index into memory using the address in ecx.

2. Copy value at that address to eax.

Slide 60
Addressing Mode: Displacement

- Like memory mode, but with constant offset
  - Offset is often negative, relative to %ebp

- `movl -12(%ebp), %eax`
  - Take the address in ebp, subtract twelve from it, index into memory and store the result in eax
Addressing Mode: Displacement

- `movl  -12(%ebp), %eax`
  - Take the address in ebp, subtract twelve from it, index into memory and store the result in eax

**CPU Registers**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1A70</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Access address:
   - `0x1A70 - 12 => 0x1A64`
Addressing Mode: Displacement

• `movl -12(%ebp), %eax`
  • Take the address in ebp, subtract three from it, index into memory and store the result in eax

### CPU Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>11</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1A70</td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>

1. Access address: 0x1A70 – 12 => 0x1A64

2. Copy value at that address to eax.
What will memory look like after these instructions?

x is 2 at %ebp-8, y is 3 at %ebp-12, z is 2 at %ebp-16

```
movl -16(%ebp), %eax
sall $3, %eax
imull $3, %eax
movl -12(%ebp), %edx
addl -8(%ebp), %edx
addl %edx, %eax
movl %eax, -8(%ebp)
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>value</td>
</tr>
<tr>
<td>%eax</td>
<td>?</td>
</tr>
<tr>
<td>%edx</td>
<td>?</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1270</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>2</td>
</tr>
<tr>
<td>0x1264</td>
<td>3</td>
</tr>
<tr>
<td>0x1268</td>
<td>2</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>
What will memory look like after these instructions?

\[ x \text{ is 2 at } %ebp-8, \ y \text{ is 3 at } %ebp-12, \ z \text{ is 2 at } %ebp-16 \]

\[
\begin{align*}
&\text{movl } -16(\%ebp), %eax \\
&\text{sall } 3, %eax \\
&\text{imull } 3, %eax \\
&\text{movl } -12(\%ebp), %edx \\
&\text{addl } -8(\%ebp), %edx \\
&\text{addl } %edx, %eax \\
&\text{movl } %eax, -8(\%ebp)
\end{align*}
\]
x is 2 at %ebp-8, y is 3 at %ebp-12, z is 2 at %ebp-16

cmp $16(%ebp), %eax
snmp $3, %eax
imull $3, %eax
ncmp $12(%ebp), %edx
addl $8(%ebp), %edx
addl %edx, %eax
ncmp %eax, $-8(%ebp)

Equivalent C code:
x = z*24 + y + x;
Stack Frame Contents

• What needs to be stored in a stack frame?
  • Alternatively: What must a function know?

• Local variables
• Previous stack frame base address
• Function arguments
• Return value
• Return address

• Saved registers
• Spilled temporaries
Stack Frame Contents

• What needs to be stored in a stack frame?
  • Alternatively: What must a function know?

• Local variables
• Previous stack frame base address
• Function arguments
• Return value
• Return address

• Saved registers
• Spilled temporaries

Slide 68
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Must adjust %esp, %ebp on call / return.

![Diagram showing %esp and %ebp pointers within a stack frame and an arrow labeled "caller"]
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  1. `pushl %ebp`
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  1. pushl %ebp
  2. Set %ebp = %esp
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  1. `pushl %ebp`
  2. Set %ebp = %esp
  3. Subtract N from %esp

Callee can now execute.
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  1. set %esp = %ebp
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp

IA32 has another convenience instruction for this: leave

Back to where we started.
Frame Pointer: Function Call

Initial state

pushl %ebp (store caller’s frame pointer)

movl %esp, %ebp (establish callee’s frame pointer)

subl $SIZE, %esp (allocate space for callee’s locals)
Frame Pointer: Function Return

Want to restore caller’s frame.

IA32 provides a convenience instruction that does all of this:
leave
Functions and the Stack

1. pushl %eip
2. jump funcB
3. (execute funcB)

Stack Memory Region

Function A

Function B

Stored PC in funcA

Text Memory Region

funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
... call funcB
addl %eax, %ecx
...

funcB:
pushl %ebp
movl %esp, %ebp
...
movl $10, %eax
leave
ret
Functions and the Stack

1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip

Text Memory Region

```assembly
funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
...  
call funcB
addl %eax, %ecx
...

funcB:
pushl %ebp
movl %esp, %ebp
...  
movl $10, %eax
leave
ret
```
Functions and the Stack

6. (resume funcA)

```
funcA:
    addl $5, %ecx
    movl %ecx, -4(%ebp)
    ...
    call funcB
    addl %eax, %ecx
    ...

funcB:
    pushl %ebp
    movl %esp, %ebp
    ...
    movl $10, %eax
    leave
    ret
```
Functions and the Stack

1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip
6. (resume funcA)

Text Memory Region

```
funcA:
  addl $5, %ecx
  movl %ecx, -4(%ebp)
  ...
  call funcB
  addl %eax, %ecx
  ...
funcB:
  pushl %ebp
  movl %esp, %ebp
  ...
  movl $10, %eax
  leave
  ret
```
Functions and the Stack

1. `pushl %eip`  
   - call
2. `jump funcB`  
3. `(execute funcB)`  
4. `restore stack`  
   - leave
5. `popl %eip`  
   - ret
6. `(resume funcA)`

**Return address:**

Address of the instruction we should jump back to when we finish (return from) the currently executing function.
Implementing a function call

**main:**

```
...  
subl $8, %esp
movl $2, 4(esp)
movl $1, (%esp)
call foo
lea $8, %esp
...  
```

**foo:**

```
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl 8(%ebp), %eax
addl $9, %eax
leave
ret
```
void bar(char * in){
    char name[5];
    strcpy(name, in);
}

bar:
    pushl %ebp
    movl %esp, %ebp
    subl $5, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%ebp)
    leal -5(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    leave
    ret
Data types / Endianness

• x86 is a little-endian architecture
Putting it all together…

Callee’s frame.

Callee’s local variables.

Caller’s Frame Pointer

Return Address

First Argument to Callee

…

Final Argument to Callee

Caller’s local variables.

Caller’s frame.

Older stack frames.

Callee Code
1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

Caller Code
1. save address of next instruction
2. push arguments
Register Convention

• Caller-saved: %eax, %ecx, %edx
  • If the caller wants to preserve these registers, it must save them prior to calling callee
  • callee free to trash these, caller will restore if needed

• Callee-saved: %ebx, %esi, %edi
  • If the callee wants to use these registers, it must save them first, and restore them before returning
  • caller can assume these will be preserved

This is why I’ve told you to only use these three registers.
Buffer Overflows
When is a program secure?

• Formal approach: When it does exactly what it should
  • not more
  • not less
• But how do we know what it is supposed to do?
Example 1

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv){
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv){
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s
", name, nice);
    return 0;
}

A. Nothing bad will happen
B. Something nonsensical will result
C. Something terrible will result
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv){
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}

It it is not null terminated it can read a lot more of the stack!
HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).

User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 51309857343.

SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

User Meg wants these 4 letters: BIRD. User Brendan uploaded the files in /tmp/files-3843. There are currently 64 connections open.

POTATO

HMM...

BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long). User Karen wants it to include "authentic".
# include <stdio.h>
# include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbbb, argv[1]);
    return 0;
}
```

Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAAAAA"

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is “AAAAAAAAAAAAAAAAAAA”

```c
#include <stdio.h>
#include <string.h>

void foo() { 0x08049b95
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xaaaaaaaaa, argv[1]);
    return 0;
}
```
Better Hijacking Control

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
```

Jump to attacker supplied code where?
- put code in the string
- jump to start of the string
Better Hijacking Control

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```

Jump to attacker supplied code where?
• put code in the string
• jump to start of the string