Announcements

• Clickers available through the bookstore and TAP
  • No more paper hand-ins next week 😊
Today

• CS 31 Recap:
  • functions and the stack
  • assembly instructions
• Stack Buffer Overflow
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
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!

Virtual address space (VAS): fixed size. 0x0F000000

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!

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Virtual address space (VAS): fixed size. 0x0F000000

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

Slide 16
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

0xFFFFFFFF

funcA:
...
call funcB
...

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

Function B

Function A

...
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
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;
}
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)

- **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)

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

Other object files (p2.o, p3.o, ...)

Library obj. code (libc.a)

Machine code instructions
Compilers

• Computers don't execute source code
  • Instead, they use machine code
• Compilers translate code from a higher level to a lower one
• In this context, C $\rightarrow$ assembly $\rightarrow$ machine code
Machine Code

• Binary (0’s and 1’s) Encoding of ISA Instructions
  • some bits: encode the instruction (opcode bits)
  • others encode operand(s)
    • 01001010 opcode operands
    • 01 001 010 ADD %r1 %r2
  • different bits fed through different CPU circuitry:
Assembly Code

1. **C program (p1.c)**
2. **Assembly program (p1.s)**
3. **Object code (p1.o)**
4. **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**

- **text**
- **binary**
- **executable**

**machine code instructions**
**What is “assembly”?**

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

```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
```

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

**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`

**Machine Code (Hexadecimal)**
- `55`
- `89 E5`
- `83 EC 10`
- `C7 45 F8 0A 00 00 00`
- `C7 45 FC 14 00 00 00`
- `8B 45 FC`
- `01 45 F8`
- `B8 45 F8`
- `C9`

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

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

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 (\( %eip, \ldots \) )
- Status of recent tests %EFLAGS (CF, ZF, SF, OF )

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</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>Current stack top</td>
</tr>
<tr>
<td>%ebp</td>
<td>Current stack frame</td>
</tr>
<tr>
<td>%eip</td>
<td>Program Counter (PC)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Register name</th>
<th>bits:</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%EFLAGS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
Assembly Programmer's View of State

**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></td>
</tr>
</tbody>
</table>

Program:
- data
- instrs
- stack

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

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

<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>

CPU Registers

(Memory)

- 0x0:
- 0x4:
- 0x8:
- 0xC:
- ...
- 0x1A64
- 0x1A68 42
- 0x1A6C
- 0x1A70
- ...
- 0xFFFFFFFF:
Addressing Mode: Memory

- \texttt{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.

(Memory)
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>42</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.
2. Copy value at that address to eax.
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>

### Memory

- 0x1A70 – 12 => 0x1A64

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

<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?

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

\[
\begin{align*}
\text{movl} & -16(\%\text{ebp}), \%\text{eax} \\
\text{sall} & \$3, \%\text{eax} \\
\text{imull} & \$3, \%\text{eax} \\
\text{movl} & -12(\%\text{ebp}), \%\text{edx} \\
\text{addl} & -8(\%\text{ebp}), \%\text{edx} \\
\text{addl} & \%\text{edx}, \%\text{eax} \\
\text{movl} & \%\text{eax}, -8(\%\text{ebp})
\end{align*}
\]

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%edx</td>
</tr>
<tr>
<td>%edx</td>
<td>%eax</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` 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>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>53</td>
</tr>
<tr>
<td>0x1264</td>
<td>3</td>
</tr>
<tr>
<td>0x1268</td>
<td>24</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></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>16</td>
</tr>
<tr>
<td>0x1268</td>
<td>24</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>53</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>

<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>53</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>
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
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.
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:
  • 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:
  • pushl %ebp
  • 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:
  • pushl \%ebp
  • Set \%ebp = \%esp
  • 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:
  • 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:
  - set `%esp = %ebp`
  - `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:
  • set %esp = %ebp
  • 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`

`popl %ebp` (restore caller’s frame pointer)
Functions and the Stack

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

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

Program Counter (%eip)

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

Stack Memory Region

Function A
...

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

6. (resume funcA)

Text Memory Region

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

`call funcB`

`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)

Stack Memory Region

Program Counter (%eip)

Text Memory Region

```asm
funcA:
  addl $5, %ecx
  movl %ecx, -4(%ebp)
  ...  \(\text{call funcB}\)
  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)

Return address:
Address of the instruction we should jump back to when we finish (return from) the currently executing function.
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
Putting it all together…

Callee’s local variables.

Caller’s Frame Pointer

Return Address

First Argument to Callee

…

Final Argument to Callee

Caller’s local variables.

…

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
Implementing a function call

```
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $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);
}
Implementing a function call

main:
  ...
  eip  subl   $8, %esp
  eip  movl   $2, 4(%esp)
  eip  movl   $1, (%esp)
  eip  call   foo
  eip  addl   $8, %esp
  ...

foo:
  eip  pushl  %ebp
  eip  movl   %esp, %ebp
  eip  subl   $16, %esp
  eip  movl   $3, -4(%ebp)
  eip  movl   8(%ebp), %eax
  eip  addl   $9, %eax
  eip  leave
  eip  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(%esp)
    leal -5(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    leave
    ret
Data types / Endianness

• x86 is a little-endian architecture
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
", name, nice);
    return 0;
}
```
What happens if we read a long name?

A. Nothing bad will happen
B. Something nonsensical will result
C. Something terrible will result
HOW THE HEARTBLEED BUG WORKS:

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

User Meg wants these 6 letters: POTATO. User Ioana 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. There are currently 99 connections open. User Brendan uploaded the file "guess in car why". Note: Files with ID 375.381. 22.37 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 99 connections open. User Brendan uploaded the file "guess in car why". Note: Files with ID 375.381. 22.37 are in /tmp/files-3843.
SERVER, ARE YOU STILL THERE? IF SO, REPLY "NAT" (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 the snakes but not too long. User Karen wants to change account password to "blabla".
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(0xffffffff,0xbfffffff,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(0xffffffff, 0xbbbbbbbb, argv[1]);
    return 0;
}
```

Load function arguments starting with the last argument.
#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

```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(0xaaaaaaaaa, 0xbbbbbbbbbb, 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,0xbbbbbbbbbb,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, 0xbbbbbbbbbb, 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,0xbbbbbbbbbb,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,0xbbbbbbbbbb,argv[1]);
    return 0;
}
```
#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(0xaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
#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

```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(0xaaaaaaaaa,0xbbbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAAAA"

```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,0xbbbbbbbbbb,argv[1]);
    return 0;
}
```

<table>
<thead>
<tr>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>buf[0-3]</td>
</tr>
<tr>
<td>saved ebp</td>
<td>0xdeadbeef</td>
</tr>
<tr>
<td>saved ret: eip</td>
<td>argv[1]</td>
</tr>
<tr>
<td>0xaaaaaaaa</td>
<td>0xbbbbbbbbbb</td>
</tr>
<tr>
<td>argv[1]</td>
<td>0xaaaaaaaa</td>
</tr>
</tbody>
</table>
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAA"

```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 “AAAAAAAAAAAAAAAAA”

```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,0xbbbbbbbbbb,argv[1]);
    return 0;
}
```

%esp  →  0x41414141
%ebp  →  0x41414141
saved →  0x41414141
eip    →  0x41414141

0x41414141
0x41414141
0x41414141
0x41414141
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAA"

```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, 0bbbbbbbb, 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,0xbbbbbbbbbb,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, 0xbbbbbbbbbb, argv[1]);
    return 0;
}
```

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

- **Shellcode**: small code fragment that receives initial control in an control flow hijack exploit
  - Control flow hijack: taking control of instruction pointer
- Earliest attacks used shellcode to exec a shell
  - Target a setuid root program, gives you root shell
Shellcode

```c
int main(void) {
    char* name[1];
    name[0] = "\texttt{/bin/sh}";
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}
```

take the compiled output?
Shellcode

There are some restrictions

1. Shellcode cannot contain null characters ‘\0’
   ➢ Why?

2. If payload is via gets() must also avoid line-breaks
   ➢ Why?
Payload is not always robust

Exact address of the shellcode start is not always easy to guess

Miss? Segfault
Fix? NOP Sled!
NOP Sled!

• NOP instruction: 0x90
• Instruct eip to move on to the next instruction
Build your very own NOP sled

Information we have

- char buffer[100] <- overflow this buffer
- address of char buffer is 0xffffd88c
- eip = 50 byte offset from char buffer[100]
- shellcode = 20 bytes
Other overflow targets

• Format strings in C
• Heap management structures used by malloc
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];
    if(argc < 3){
        return -1;
    }
    i = atoi(argv[1]);
    s = i;
    if(s >= 80) {
        /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }
    printf("s = %d\n", s);
    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);
    return 0;
}
Integer overflows

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

int main(int argc, char *argv[])
{
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3)
        return -1;

    i = atoi(argv[1]);
    s = i;

    if(s >= 80) {
        /* [w1] */
        printf("Oh no you don't!\n");
        return -1;
    }

    printf("s = %d\n", s);
    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
```

Output

$ ./overflow 5 hello
s = 5
hello

$ ./overflow 80 hello
Oh no you don’t

$ ./65536 hello
s = 0
Segmentation fault (core dumped)
What’s wrong with this code?

```c
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

void *memcpy(void *dest, const void *src, size_t n);

typedef unsigned int size_t;

A. Nothing
B. Buffer overflow
C. Integer overflow
D. Race Condition
Variable arguments in C

- In C, we can define a function with a variable number of arguments

```c
void printf(const char* format, ....)
```

Usage:

```c
printf("hello world");
printf("length of %s = %d \n", str, str.length());
```

format specification encoded by special % characters
Sloppy use of printf

```c
void main(int argc, char* argv[])
{
    printf( argv[1] );
}
```
Sloppy use of printf

```c
void main(int argc, char* argv[])
{
    printf( argv[1] );
}
```

argv[1] = "%s%s%s%s%s%s%s%s%s%s"

Attacker controls format string gives all sorts of control:
- Print stack contents
- Print arbitrary memory
- Write to arbitrary memory
Implementation of printf

- Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time

```c
void printf(const char* format, ...) {
    int i; char c; char* s; double d;
    va_list ap; /* declare an “argument pointer” to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != ‘\0’; p++) {
        if (*p == ‘%’) {
            switch (*++p) {
                case ‘d’:
                    i = va_arg(ap, int); break;
                case ‘s’:
                    s = va_arg(ap, char*); break;
                case ‘c’:
                    c = va_arg(ap, char); break;
            }
            /* etc. for each % specification */
        }
    }
    /* etc. for each % specification */
}
```

printf has an internal stack pointer
Viewing the stack

We can show some parts of the stack memory by using a format string like this:

C code

```c
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

Output

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

instruct printf:
• retrieve 5 parameters
• display them as 8-digit padded hexademical numbers
Closer look at the stack

```c
printf("Numbers: %d,%d", 5, 6);
```

```
printf("Numbers: %d,%d");
```

- Local variables
- Args
- Caller's frame
- Internal stack pointer starts here
- Addr 0xFF...F
Format specification encoded by special % characters

### Format Specifiers

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Passed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>decimal (int)</td>
<td>value</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned decimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%x</td>
<td>hexadecimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%s</td>
<td>string ((const) (unsigned) char *)</td>
<td>reference</td>
</tr>
<tr>
<td>%n</td>
<td>number of bytes written so far, (* int)</td>
<td>reference</td>
</tr>
</tbody>
</table>
The %n format specifier

```c
void main()
{
    int count = 0;
    printf("Hello%n", &count);
}
```
The \texttt{\%n} format specifier

- \texttt{\%n} format symbol tells \texttt{printf} to write the number of characters that have been printed

\begin{verbatim}
printf("Overflow this!\%n", &myVar);
\end{verbatim}

- Argument of \texttt{printf} is interpreted as destination address
- This writes 14 into \texttt{myVar} (“Overflow this!” has 14 characters)

- What if \texttt{printf} does \textbf{not} have an argument?

\begin{verbatim}
char buf[16]="Overflow this!\%n";
printf(buf);
\end{verbatim}

- Stack location pointed to by \texttt{printf}’s internal stack pointer will be \textbf{interpreted as address} into which the number of characters will be written.