Lab 3 (Bomb) Due 1:15pm Tomorrow

Lab 4 (Attack) Starts Tomorrow — New Partner!

Take-Home Midterm available by 5pm Tomorrow Afternoon
(75-minute exam due 5pm next Friday)
Security: The Story So Far
Observation

The program crashed because the code "returned" (jumped) to address 0x400034, which didn't contain valid machine code.

And by typing in a carefully-chosen 32-character string, we can make echo() "return" (jump) to any address we want!
Code Injection Attacks

Input string includes bytes encoding machine code

Overwrite return address A with address of that code!

```
int Q() {
    char buf[64];
    gets(buf);
    ... return ...;
}
```

```
void P(){
    Q();
    ...
}
```

Stack before call to `gets()`

Stack after call to `gets()`

What happens when Q returns?
2. System-Level Protections can help

- Non-executable code segments
  - In previous x86, could mark region of memory as either “read-only” or “writeable”... could execute anything readable
  - X86-64 added explicit “execute” permission
  - Stack marked as non-executable

Any attempt to execute this code will fail
Are We Still in Danger?

If the stack is marked "don't execute"

- we can't write machine code into the buffer and jump to it.
- but we can still overwrite the return address
- we can force a "return" (jump!) anywhere in the code that is running.

Is that really so bad?

Yes!
There are lots of instructions in a typical program.

Suppose that at address 0x410000 there are two consecutive instructions:

```
inc %ebp
ret
```

Suppose we overwrite the return address with 0x410000.

What happens when function Q returns?
Question 2

There are lots of instructions in a typical program.

Suppose that at address 0x410000 there are two consecutive instructions:

\[ \text{incl }\%\text{ebp} \]
\[ \text{retq} \]

Suppose we overwrite the return address with three copies of 0x410000.

What happens when function Q returns?
Return-Oriented Programming (ROP)

Idea:
• Find existing machine code instructions followed by retq
  (These are called gadgets)
• Put a sequence of gadgets addresses on the stack.
  (where the sequence of gadgets does our evil work)

The computer returns (jumps) from each gadget to the next!
• It reads addresses from the stack, but executes code in the text segment.

But most of our retq instructions immediately follow addq $..., %rsp.
• Can attacker find enough gadgets to do evil? 
  
  Yes!
We don't need `retq`; we need `0xc3`!

Unintended instructions — `ecb_crypt()`

```
movl $0x00000001, -44(%ebp)
test $0x00000007, %edi
setnzb -61(%ebp)
add %dh, %bh
movl $0x0F000000, (%edi)
xchg %ebp, %eax
inc%ebp
ret
```

Have Fun with Lab 4!
Review Topics

- Bits
- And/Or/Not/Xor
- Arithmetic & logical shifts
  - Unsigned ints
  - 2's complement
  - Max/min values
- Integers
- Negating a signed int
- Signed/unsigned compare
- Zero- vs. sign-extension
- Casting
- Overflow
- Mult/Div vs. Shifting
- IEEE float & double
- Normal, special, and denormal fp numbers
- Memory vs. registers
- Machine code vs. assembly
- x86 assembly
  - arithmetic
  - movq vs. leaq
  - comparisons
  - condition codes
  - conditional jumps
  - conditional moves
- Implementing if, do, while loops using jumps & labels
- Stack frames & %rsp
- Return address
- Arrays, Structs, Unions
  - Padding/alignment
- Buffer overflows
  - Identifying
  - Security implications
  - Prevention techniques
expt mantissa

single 32-bit

double 64-bit

\[
\begin{align*}
\text{ign} & = + \\
& = - \\
\text{expt} - \text{bias} & = +127 \\
1 \cdot \text{mantissa} \times 2 &  \\
+ & \frac{1}{1} \times 2^{-128} \\
\end{align*}
\]

- If expt is all 0's then normals
  \[+0. \text{mantissa} \times 2^{-126}\]
- If expt is all 1's then
  \[-0. \text{mantissa} \times 2^{-126}\]
- Special
  \[+\infty, -\infty, \text{or} \text{NaN}, \text{depending on mantissa bits.}\]
if $x = 0$ then $-x \leq 0$ ✓

- For 8 bit signed
  
  $-128 \leq x \leq 127$

if $x \leq 0$ then $-x > 0$ ✗

$-( -128 ) = -128$

$n$ bits

<table>
<thead>
<tr>
<th></th>
<th>unsigned</th>
<th>signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{max}}$</td>
<td>$2^n - 1$</td>
<td>$2^{n-1} - 1$</td>
</tr>
<tr>
<td>$T_{\text{max}}$</td>
<td>$2^{n-1}$</td>
<td>$2^{n-1} - 1$</td>
</tr>
<tr>
<td>$U_{\text{min}}$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$T_{\text{min}}$</td>
<td>$-2^{n-1}$</td>
<td>$-2^{n-1}$</td>
</tr>
</tbody>
</table>
callq f

Code for f:
  subq $n, %rsp
  callq f
  addq $n, %rsp
  retq

stack space

used

return addr

1 n bytes

return addr

1 n bytes

not (yet) used

_%esp - %esp

f’s stack frame

f’s stack frame

f’s stack frame
caller-save registers — functions are allowed to use these w/o putting them back the way they function must put the original values back before they return.
callee-save registers —
Floats

- **Sign**: 0 = +, 1 = -
- **Exponent**: "exp" (biased by 127 for single precision, 1023 for double precision)
- **Mantissa**: "mantissa" (for normal numbers)

**Single-Precision**

- 32 bits total
- **Exponent**: 8 bits (1 bit for bias)
- **Mantissa**: 23 bits

**Double-Precision**

- 64 bits total
- **Exponent**: 11 bits (1 bit for bias)
- **Mantissa**: 52 bits

**Normals**: $\pm 1 \cdot "mantissa" \times 2^{expt - "bias"}$

**Denormals**: $\pm 0 \cdot "mantissa" \times 2^{1 - "bias"}$ (if "expt" = 0000000)

**Special**

- (if "expt" = 1111...1)
- $+\infty$, $-\infty$, NaN depending on "mantissa" bit pattern
Structs & alignment

Struct {
    char c;
    int i;
}

Goal: 4 byte primitive data should live/start at an address that is a multiple of 4

8-byte primitive data ... 8-byte primitive data ...

Fact: structs will be placed so they start at a nice address (multiple of 4, 8, 16, ...)

Struct {
    char c;
    char d;
    int i;
}

Struct {
    c, d, 2 bytes, i
}