Announcements

• Things Due:
  – Homework #1: Due Friday (tomorrow)

• Lab 1 out
  – If you haven’t created a group and gotten access, please do so ASAP

• It will be hard to do Lab 1 without:
  – Reading (see course schedule):
    • Smashing the Stack for Fun and Profit
    • Exploiting Format String Vulnerabilities
  – Attending section this week and next
Review: `printf()` and the Stack

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

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

Printf’s internal stack pointer starts here

Addr 0xFF...F

Printf’s internal stack pointer starts here

Addr 0xFF...F
Summary ofPRINTF Risks

-PRINTF takes a variable number of arguments
  - E.g., printf(“Here’s an int: %d”, 10);

- Assumptions about input can lead to trouble
  - E.g., printf(buf) when buf=“Hello world” versus when buf=“Hello world %d”
  - Can be used to advance printf’s internal stack pointer
  - Can read memory
    - E.g., printf(“%x”) will print in hex format whatever printf’s internal stack pointer is pointing to at the time
  - Can write memory
    - E.g., printf(“Hello%n”); will write “5” to the memory location specified by whatever printf’s internal SP is pointing to at the time
How Can We Attack This?

```c
foo() {
    char buf[...];
    strncpy(buf, readUntrustedInput(), sizeof(buf));
    printf(buf); //vulnerable
}
```

What should the string returned by `readUntrustedInput()` contain??

Canvas -> Quizzes -> Oct 7

Note: Different compilers / compiler options / architectures might vary
Using `%n` to Overwrite Return Address

In `foo()`'s stack frame:

- Buffer with attacker-supplied input “string”
- “... attackString%n...”, attack code
- &RET
- SFP
- RET
- Return execution to this address

Why is “in” in quotes? C allows you to concisely specify the “width” to print, causing `printf` to pad by printing additional blank characters without reading anything else off the stack.

Example: `printf("%5d%n", 10)` will print three spaces followed by the integer: “   10”

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte.

(4x `%n` to write into &RET, &RET+1, &RET+2, &RET+3)

This portion contains enough `%` symbols to advance `printf`'s internal stack pointer

Number of characters “in” attackString must be equal to ... what?

When `%n` happens, make sure the location under `printf`'s internal stack pointer contains address of RET; `%n` will write the number of characters printed so far into RET

Return execution to this address
Buffer Overflow: Causes and Cures

• Classical memory exploit involves code injection
  – Put malicious code at a predictable location in memory, usually masquerading as data
  – Trick vulnerable program into passing control to it

• Possible defenses:
  1. Prevent execution of untrusted code
  2. Stack “canaries”
  3. Encrypt pointers
  4. Address space layout randomization
  5. Code analysis
  6. ...
Defense: Executable Space Protection

• Mark all writeable memory locations as non-executable
  – Example: Microsoft’s Data Execution Prevention (DEP)
  – This blocks many code injection exploits

• Hardware support
  – AMD “NX” bit (no-execute), Intel “XD” bit (executed disable) (in post-2004 CPUs)
  – Makes memory page non-executable

• Widely deployed
  – Windows XP SP2+ (2004), Linux since 2004 (check distribution), OS X 10.5+ (10.4 for stack but not heap), Android 2.3+
Question

What might an attacker be able to accomplish even if they cannot execute code on the stack?
What Does “Executable Space Protection” Not Prevent?

• Can still corrupt stack …
  – … or function pointers
  – … or critical data on the heap

• As long as RET points into existing code, executable space protection will not block control transfer!
  ➔ return-to-libc exploits
return-to-libc

• Overwrite saved ret (IP) with address of any library routine
  – Arrange stack to look like arguments

• Does not look like a huge threat
  – ... Right?
  – We can call any function we want!
  – Say, exec 😊
return-to-libc++

• Insight: Overwritten saved EIP need not point to the *beginning* of a library routine
• *Any* existing instruction in the code image is fine
  – Will execute the sequence starting from this instruction
• What if instruction sequence contains RET?
  – Execution will be transferred... to where?
  – Read the word pointed to by stack pointer (SP)
    • Guess what? Its value is under attacker’s control!
  – Use it as the new value for IP
    • Now control is transferred to an address of attacker’s choice!
  – Increment SP to point to the next word on the stack
Chaining RETs

• Can chain together sequences ending in RET
  – Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)

• What is this good for?

• Answer [Shacham et al.]: everything
  – Turing-complete language
  – Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  – Attack can perform arbitrary computation using no injected code at all – return-oriented programming
Return-Oriented Programming