Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Admin

• Assignments:
  – Homework 1: Due today at 11:59pm
  – Lab 1: Sign up, granting access ~once per day, see forum
Summary of printf 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??

If format string contains % then printf will expect to find arguments here...

What should the string returned by `readUntrustedInput()` contain??
Using %n to Overwrite Return Address

In foo()'s stack frame:

Buffer with attacker-supplied input “string”

“... attackString%n”, attack code

&RET

SFP

RET

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

When %n happens, make sure the location under printf’s stack pointer contains address of RET; %n will write the number of characters in attackString into RET

Return execution to this address

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”, 10) will print three spaces followed by the integer: “   10”
That is, %n will print 5, not 2.

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)
Recommended Reading

• 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 next week
Buffer Overflow: Causes and Cures

• Typical 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. ...
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+
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 EIP with address of any library routine
  – Arrange stack to look like arguments

• Does not look like a huge threat
  – Attacker cannot execute arbitrary code
  – But … ?
    • Can still call critical functions, like exec

• See lab 1, sploit 8 (extra credit)
return-to-libc on Steroids

• 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 (ESP)
    • Guess what? Its value is under attacker’s control!
  – Use it as the new value for EIP
    • Now control is transferred to an address of attacker’s choice!
  – Increment ESP to point to the next word on the stack
Chaining RETs for Fun and Profit

• 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
Run-Time Checking: StackGuard

- Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary
Run-Time Checking: StackGuard

- Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

- Choose random canary string on program start
  - Attacker can’t guess what the value of canary will be

- Terminator canary: “\0”, newline, linefeed, EOF
  - String functions like strcpy won’t copy beyond “\0”
StackGuard Implementation

• StackGuard requires code recompilation
• Checking canary integrity prior to every function return causes a performance penalty
  – For example, 8% for Apache Web server at one point in time
• StackGuard can be defeated
  – A single memory write where the attacker controls both the value and the destination is sufficient
Defeating StackGuard

- Suppose program contains `strcpy(dst,buf)` where attacker controls both `dst` and `buf` – Example: `dst` is a local pointer variable

```
buf &dst canary sfp RET

Return execution to this address
```

```
BadPointer, attack code &RET canary sfp RET
```

Overwrite destination of `strcpy` with RET position

strcpy will copy BadPointer here
ASLR: Address Space Randomization

- Randomly arrange address space of key data areas for a process
  - Base of executable region
  - Position of stack
  - Position of heap
  - Position of libraries
- Introduced by Linux PaX project in 2001
- Adopted by OpenBSD in 2003
- Adopted by Linux in 2005
ASLR: Address Space Randomization

• Deployment (examples)
  – Linux kernel since 2.6.12 (2005+)
  – Android 4.0+
  – iOS 4.3+ ; OS X 10.5+
  – Microsoft since Windows Vista (2007)

• Attacker goal: Guess or figure out target address (or addresses)

• ASLR more effective on 64-bit architectures
Attacking ASLR

• **NOP slides** and **heap spraying** to increase likelihood for custom code (e.g., on heap)

• **Brute force attacks** or memory disclosures to map out memory on the fly
  – Disclosing a single address can reveal the location of all code within a library, depending on the ASLR implementation
Other Possible Solutions

• Use safe programming languages, e.g., Java
  – What about legacy C code?
  – (Though Java doesn’t magically fix all security issues 😊)

• Static analysis of source code to find overflows

• Dynamic testing: “fuzzing”