Admin

- Lab 1 out, groups being formed
  - Come to quiz section this week!

- Looking forward
  - This week: More buffer overflows + beyond
  - Also this week: Transition to crypto
Buffer Overflow: Causes and Cures

• Typical memory exploit involves **code injection**
  – Classic approach: 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 “saved EIP” points into existing code, executable space protection will not block control transfer

• This is the basis of return-to-libc exploits
  – 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 … ?
return-to-libc

• Can still call critical functions, like exec

• See lab 1, sploit 8
return-to-libc on Steroids

• 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

  ![Stack Frame Diagram]

• 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
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  canary  sfp  RET
```

Defeating StackGuard

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

Return execution to this address

Overwrite destination of strcpy with RET position
Defeating StackGuard

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

```plaintext
strcpy
-----
BadPointer, attack code &RET &canary sfp RET

Return execution to this address

Overwrite destination of `strcpy` with RET position

strcpy will copy BadPointer here
```
More on Defeating StackGuard

• Attacker sets buf to contain (first) a pointer to another region in buf with the attack code, and then (second) the attack code

• Attacker sets dst, to contain the address where RET is stored (recall the assumption that the attacker can also set dst)

• When the strcpy happens, memory beginning at the address of RET is overwritten with the contents of buf
  – This puts “BadPointer” in the location of RET
  – Recall that “BadPointer” is a value for the address at which the attack code starts (in buf)
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) (not by default)

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

• ASLR more effective on 64-bit architectures
ASLR Issues

• 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”
• Modern compiler options, e.g., incorporate stack canaries
Beyond Buffer Overflows...
Another Type of Vulnerability

• Consider this code:

```c
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

• **Goal:** Open only regular files (not symlink, etc)
• **What can go wrong?**
TOCTOU (Race Condition)

• TOCTOU == Time of Check to Time of Use:

```c
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISRREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

• **Goal:** Open only regular files (not symlink, etc)
• Attacker can change meaning of `path` between `stat` and `open` (and access files he or she shouldn’t)
This TOCTOU Example

• In call to open, pass O_NOFOLLOW to not follow symbolic links
• Call fstat on open file descriptor
• ...
Another Type of Vulnerability

- Consider this code:

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

```c
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```
Implicit Cast

• Consider this code:

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```
Another Example

```c
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

(from www-inst.eecs.berkeley.edu—implflaws.pdf)
Integer Overflow

- What if `len` is large (e.g., `len = 0xFFFFFFFF`)?
- Then `len + 5 = 4` (on many platforms)
- Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

```c
size_t len = read_int_from_network();
char *buf;
buf = malloc(len + 5);
read(fd, buf, len);
```

(from [www-inst.eecs.berkeley.edu—implflaws.pdf](http://www-inst.eecs.berkeley.edu/~implflaws.pdf))