Announcements

• Lab 1
  – Part 1a due Friday

• Homework 1
  – Also Friday
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 or check integrity of pointers
  4. Address space layout randomization
  5. Code analysis
  6. …
Defense: 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
Defense: 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

- Canary contains: "\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
Defeating StackGuard

• StackGuard can be defeated
  – A single memory write where the attacker controls both the value and the destination is sufficient

• Suppose program contains `copy(buf, attacker-input)` and `copy(dst, buf)`
  – Example: `dst` is a local pointer variable
  – Attacker controls both `buf` and `dst`
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 sleds** and **heap spraying** to increase likelihood for adversary’s code to be reached (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
Defense: Shadow Stacks

• Idea: don’t store return addresses on the stack!

• Store them on... a different stack!
  – A hidden stack

• On function call/return
  – Store/retrieve the return address from shadow stack

• Or store on both main stack and shadow stack, and compare for equality at function return

• 2020/2021 Hardware Support emerged (e.g., Intel Tiger Lake, AMD Ryzen PRO 5000)
Challenges With Shadow Stacks

• Where do we put the shadow stack?
  – Can the attacker figure out where it is? Can they access it?

• How fast is it to store/retrieve from the shadow stack?

• How big is the shadow stack?

• Is this compatible with all software?

• (Still need to consider data corruption attacks, even if attacker can’t influence control flow.)
Other Big Classes of Defenses

• Use safe programming languages, e.g., Java, Rust
  – 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”
Fuzz Testing

• Generate “random” inputs to program
  – Sometimes conforming to input structures (file formats, etc.)

• See if program crashes
  – If crashes, found a bug
  – Bug may be exploitable

• Surprisingly effective

• Now standard part of development lifecycle
Other Common Software Security Issues...
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);
}

void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```
Another Example

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

Canvas -> Quizzes -> January 18
• 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);
}
```

If `len` is negative, may copy huge amounts of input into `buf`.

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

(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](www-inst.eecs.berkeley.edu—implflaws.pdf))
Another Type of Vulnerability

- Consider this code:

```c
if (access("file", W_OK) != 0) {
    exit(1); // user not allowed to write to file
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- **Goal:** Write to file only with permission
- **What can go wrong?**
TOCTOU (Race Condition)

• TOCTOU = “Time of Check to Tile of Use”

```c
if (access("file", W_OK) != 0) {
    exit(1); // user not allowed to write to file
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

• **Goal:** Write to file only with permission
• Attacker (in another program) can change meaning of “file” between access and open:
  symlink("/etc/passwd", "file");
Password Checker

• Functional requirements
  – PwdCheck(RealPwd, CandidatePwd) should:
    • Return TRUE if RealPwd matches CandidatePwd
    • Return FALSE otherwise
  – RealPwd and CandidatePwd are both 8 characters long
Password Checker

• Functional requirements
  – $\text{PwdCheck}(\text{RealPwd}, \text{CandidatePwd})$ should:
    • Return TRUE if RealPwd matches CandidatePwd
    • Return FALSE otherwise
  – RealPwd and CandidatePwd are both 8 characters long

• Implementation (like TENEX system)

```
PwdCheck(RealPwd, CandidatePwd) // both 8 chars
for i = 1 to 8 do
  if (RealPwd[i] != CandidatePwd[i]) then
    return FALSE
return TRUE
```

• Clearly meets functional description
Attacker Model

- Attacker can guess CandidatePwds through some standard interface
- Naive: Try all \(256^8 = 18,446,744,073,709,551,616\) possibilities
- Is it possible to derive password more quickly?
Timing Attacks

• Assume there are no “typical” bugs in the software
  – No buffer overflow bugs
  – No format string vulnerabilities
  – Good choice of randomness
  – Good design
• The software may still be vulnerable to timing attacks
  – Software exhibits input-dependent timings
• Complex and hard to fully protect against
• Even possible over a network
  – “Remote timing attacks are possible” (Brumley & Boneh, 2005)
Other Examples

- Plenty of other examples of timings attacks
  - Timing cache misses
    - Extract cryptographic keys...
    - Recent Spectre/Meltdown attacks
  - Duration of a rendering operation
    - Extract webpage information
  - Duration of a failed decryption attempt
    - Different failures mean different thing (e.g., Padding oracles)

- Plenty of other side channels... We’ll return to this later in the course
Software Security:
So, what do we do?
General Principles

• Check inputs
• Check all return values
• Least privilege
• Securely clear memory (passwords, keys, etc.)
• Failsafe defaults
• Defense in depth
  – Also: prevent, detect, respond
General Principles

• Reduce size of trusted computing base (TCB)
• Simplicity, modularity
  – But: Be careful at interface boundaries!
• Minimize attack surface
• Use vetted components
• Security by design
  – But: tension between security and other goals
• Open design? Open source? Closed source?
  – Different perspectives
Vulnerability Analysis and Disclosure

• What do you do if you’ve found a security problem in a real system?
• Say
  – A commercial website?
  – UW grade database?
  – Boeing 787?
  – TSA procedures?

What would you do? What ethical questions come up?
Suppose companies A, B, and C all have a vulnerability, but have not made the existence of that vulnerability public

Company A has a software update prepared and ready to go that, once shipped, will fix the vulnerability; but B and C are still working on developing a patch for the vulnerability

Company A learns that attackers are exploiting this vulnerability in the wild

Should Company A release their patch, even if doing so means that the vulnerability now becomes public and other actors can start exploiting Companies B and C?

Or should Company A wait until Companies B and C have patches?