1 Overview

Last lecture we discussed some of the principles of security:

- Open Design
- Policy of Least Privilege
- Fail-safe Defaults
- Complete Mediation

This lecture we discuss the remaining principles and delve into buffer overflow vulnerabilities.

2 Principles of Security (continued)

2.1 Economy of Mechanism

The Economy of Mechanism principle states that the design of a security system should be simple. While minimizing the number of lines of code in a system does not guarantee its simplicity (ex: PERL), systems with a large amount of code are invariably complex:

More code → more bugs → more security vulnerabilities

This logic is behind current trends to ship devices with services turned off by default; if a program is not running, the number of bugs it has is irrelevant.

2.1.1 Least Common Mechanism

To maximize security, sharing and dependencies between security mechanisms should be minimized. This is necessary to prevent unintended interaction effects.

An extreme example of such minimization is Orthogonal Security in which security mechanisms are completely independent from the rest of the system and each other. Examples include HTTPS, Firewalls, and HD Encryption. Each of these, in most cases, act as independent modules that add to the security of a computer system without unintended negative consequences.

Negative Interaction Example: Paging

Paging occurs when requested data is not in memory. In an era where computers were very, very slow, hard drives were even slower than they are now. Since the HDD must be accessed to bring a page into memory, this leaves an opportunity for various attacks:
bool check(char password[8], char input[8]) {
    for (i = 0; i < 8; i++)
        if (password d[i] != input[i])
            return false;
    return true;
}

- Password can be 8 characters long → 256^8 passwords to guess.
- Attacker creates a page break right after the first character of the password
- A delay (due to a page fault) alerts the attacker that the character they guessed was correct

This process can then be repeated, placing the page break after the second character and so forth. The new worst case running time is 256*8. This is known as a **Side-Channel Attack** since it involves the use of information gained from the physical implementation of the system. Fast hard disks make it difficult to observe paging today.

### 2.2 Psychological Acceptability

Users must understand (at a high level) and participate in the security mechanisms of a system in order for those mechanisms to work properly.

### 2.3 Separation of Privilege

When possible, requiring multiple keys to access a protected mechanism is more secure than requiring a single key. This prevents a single security breach from compromising a mechanism. The most common example of this principle is the separation of nuclear launch codes.

**Question:** doesn’t the ”root” user break many of these principles? Yes. Especially before new improvements to UNIX-like systems that separate different root capabilities.

### 3 Buffer Overflow Vulnerabilities

In this lecture, we began discussing the details behind various buffer overflow attacks that involve abusing the structure of the stack. More details can be found in the assigned reading: [http://insecure.org/stf/smashstack.html](http://insecure.org/stf/smashstack.html).

#### 3.1 Memory Layout

To begin our discussion, we went over the basics of where local, static, global, and dynamically allocated variables are stored for any given process (in addition to program code, environmental variables, and command line arguments):

Insert Memory-Layout Figure from [http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/Memory-Layout.gif](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/Memory-Layout.gif)

#### 3.2 The Stack

The stack is where local variables are stored and is the focus of many buffer overflow attacks.
3.2.1 C calling convention

In C, the simplified calling convention for functions/procedures is:

- Push arguments (last first)
- Push the return address

This is critical because, given the use of a single stack, an attacker can manipulate the return address and arguments to run their own code or do something malicious with your code.

3.3 Using gets() maliciously

given a buffer such as

```c
char buf[8]
```

and a statement such as

```c
gets(buf);
```

which, due to c’s lack of mandatory bounds checking, will allow input into ”buf” past its allotted memory, an attacker can then write past buf to both overwrite the return address AND inject their own code to run (most typically a shell of some sort).

Insert Buffer Overflow figure from

http://www.read.seas.harvard.edu/~kohler/class/05f-osp/notes/fig19-01.jpg

3.4 Defensive Strategies

- Mandatory bounds checking
- Non-executable stack

3.5 Remaining Vulnerabilities

While the defenses above prevent the attacker from running their own code, they may still be able to run your code or return to a lib-C.

3.6 Remaining Defenses

- Address space randomization
- Address space layout randomization