CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 24th, 2011

Strategies to make software secure

Design level

Today we will discuss some general principles of **secure software design**. We will discuss design patterns that seem very useful at increasing the odds that your software is secure.

Major principles of secure system design

The main source of inspiration is the following paper by Saltzer & Schroeder: The Protection of Information in Computer Systems, Jerome H. Saltzer, Michael D. Schroeder

This also looks interesting: Fundamental practices for secure software development

Economy of mechanism

KISS principle: keep it simple, stupid.

Having a **small trusted computing base** (TCB), the software and hardware critical to security, is a good thing.

- What's a TCB? The OS is part of the TCB, but Minesweeper is not.
- *"You don't have to trust code you don't run"* (code that is not part of the TCB)

Smaller code means fewer bugs, easier to verify and possible prove the code's security:

- Some folks in Australia recently proved that an OS is "correct" (no memory errors, etc.). The OS was a simplified version of the L7 microkernel (scheduling, memory management, file systems, device drivers, networking: a microkernel tries to move a lot of that out of the kernel into the user-space)

Failsafe defaults

When there is no explicit policy, then you should make one up with security as your guiding policy: In most cases this means denying access:

- **The `system` function:** sees invalid return address on the stack, it should terminate the program and not just return since terminating is safer in this case, once the attacker succeeding in modifying the stack contents.
- **ACLs:** if you get to the end of the ACL and the access has not been granted by any of the ACEs, permission should be denied.
- **Firewall rules:** if there is no explicit rule to allow access through a port on a specific address, connection is refused
- **Default install configuration:** when you install an OS, it comes with a lot of "stuff". Your installer does not set `/usr/bin/ls` to be setuid root. It doesn't set `/usr/bin` to bee world-writable. It has sensible defaults. In Windows 95 or Windows NT, for backwards compatibility reasons, the `C:\Windows\` directory was world-writable.

What happens if the **failsafe configuration is wrong?** Someone who should be able to access the system will get access denied. Worst case is they complain and you fix it.

Too **strict rules** will *lead* to **complaints** will *lead* to **fixes**. To **lax rules** will *lead* to **no complaints** and to **security holes**.

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 24th, 2011

"fail loudly" vs. "fail silently." When something fails you should inform the user why it failed without disclosing too much information (think SQL query failures which print the query to the user thereby disclosing a lot of information about the query itself and about the table structure in the database)

Least privilege

Each process or system should have **least privilege** to do its job (the minimum set of privileges needed to do its job).

- We've seen this in our printing client program
- Really cool paper on how to design web applications to improve the security of the users by applying the least privilege principle: CLAMP: Practical Prevention of Large-Scale Data Leaks

Web application structure

Communication: Database <-> Web-server (PHP) <-> Browser (all these are often separate computers)

Imagine the attacker takes over the web-server machine. He can get data from the DB, from the users, modify the DB, etc. The problem is that most web apps are all powerful with respect to the database.

For example, if you take Amazon, there are Amazon customers and Amazon employees.

There is an internal web app for Amazon employees and it communicates with the DB, and there is also a customer web app for Amazon customers which also talks to the DB. Amazon uses user accounts to give different privileges to the internal web app and the customer web app (like RW access to the Products table).

What the CLAMP paper proposes:
When you initially connect to the webserver, you connect to a "nobody server" and they introduce a new component called the "query filter" or "view". Instead of allowing the server to talk to the DB directly, you let the "query filter" intermediate their communication.

When you log in, a VM is spawned that runs the web application and there is a new view setup that allows the user (Brijesh) to see everything in the initial nobody view plus rows in the DB tables that are for his user. This way Brijesh will not be able to see everyone's credit card info in the billing table for instance, should he do an SQL injection attack.

Avoid redundant parsing

This is not part of the principles in the Saltzer & Schroeder paper.

Example: XSS content-sniffing attacks were possible because of redundant parsing done by the browser.

Least shared mechanism

If you have two systems, you would want to share code to save time. This kind of contradicts the **economy of mechanism principle.**

Least shared state: Just because you have an account on Facebook, it doesn't mean you should have an account on the Facebook.com Apache webserver.

- If you have two subsystems, don't share state between them if they have different security goals

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 24th, 2011

Software diversity: Because we have a **monoculture of software** on the Internet (everybody uses the same software, think Windows), when someone writes a virus or a worm, it can affect a huge percentage of all the computers. A more diverse software ecosystem would prevent this from happening.

Example: You're writing a MMO, such that people can start a lot of servers and clients connect to them. A worm would spread from clients to servers and then from servers to other client. If the client and server use some shared library that has a bug then the worm can exploit it in both programs and spread easily. However, if you had two teams to write two separate libraries, one for the client and one for the server, the bug might only be in one of them.

Complete mediation

If you're going to control access to a resource then you have to **control every possible way of accessing that resource**.

If you are writing a web app and want to prevent XSS attacks then you **need to filter input at every possible place** that the user can enter it.

The **challenge is not knowing** that you need to do this, **it's to make sure** you did it in every place.

Linux syscalls

There are **about 300 system calls** in Linux that you can call inside the kernel, and every one of those should have a **permission check inside them**. When you make a syscall the user process specifies the syscall number, like 294. This transfers control into the kernel to `syscall_number(294)`. The syscall has to check permissions. Since there are 300 of these syscalls, it is conceivable for someone to forget to do the checking and wreak havoc.

An alternative design is to change the interface to a system call. You can have the user process push the syscall number on the stack and then do a syscall 0 which goes into the kernel to `syscall_number(0)`. The kernel checks permissions in `syscall_number(0)` (one place) and then it gets the syscall number from the stack and executes `syscall_number(294)`.

The data problem

Sometimes mediation is about the way data flows in your program and not about how control in your code flows. For instance, with XSS attacks, you don't want input to be outputted to the user unless it was sanitized: you are interested in the data flow.

A kernel example

When you call a function `int read_file(int fd, char * buf, int len)`, you pass a pointer into the kernel. Inside the kernel there is a `sys_read` function which gets called and passed `read_file` 's parameters.

Virtual memory of Linux application
Upper 1GB is kernel
Lower 3GB is application

A lot of verification has to be done in the `sys_read` function, and other functions alike:

```
int sys_read(int fd, char * buf, int len)
{
	//	Don't trust that fd is valid
	//	Make sure len is not too large
	//	Make sure buf points to user space and is not zero
	//	Make sure that buf is a valid mapped virtual address
```

```
        memcpy(buf, tmp, len);
}
```

Kernel developers knew users would pass evil pointers, so they created a function that safely copies data (verifies the pointer properly):

```
void copy_to/from_user(char * dest, char * src, int len);
```

The issue is that it's **still possible to forget to use that function**. You can still do an unsafe `memcpy` and your program will still work. This contradicts complete mediation.

> *"It's not enough to make the secure behavior easy. You have to make the insecure behavior hard or impossible."* – Rob Johnson

You can do a quick hack to prevent people from accidently calling `memcpy` inside `sys_read`. You can have a `struct` that represents a user pointer and change the `sys_read` function as follows:

```
int sys_read(int fd, struct user_ptr buf, int len);
```

Now, if you want to call `memcpy` inside `sys_read` you will get a compiler error, so you've made it hard for the programmer to do the wrong thing.

```
struct user_ptr {
        int dummy;
        void * p;
};
```

This illustrates the **"get the compiler to help" principle**: Think of how to structure your program so that security errors become compiling errors.

For instance, you can store dangerous input data, such as XSS vulnerable input, into a `DangerousInput` which does not have a `ToString` method and can only be outputted by a `DataSanitizer` object which sanitizes it.