Design Patterns for Building Secure Systems

In these notes we aim to capture some important patterns for building secure systems, and, in particular, what you can do at design time to improve security. How can you choose an architecture that will help reduce the likelihood of flaws in your system, or increase the likelihood that you will be able to survive such flaws? We begin with a powerful concept, the notion of a trusted computing base (TCB).

The Trusted Computing Base (TCB).

A *trusted* component is a part of the system that we rely upon to operate correctly, if the whole system is to be secure; to turn it around, a trusted component is one that is able to violate our security goals if it misbehaves. A *trustworthy* component is a part of the system that we would be justified in trusting, i.e., where we'd be justified in expecting it to operate correctly. For instance, on Unix systems the super-user (root) is trusted; hopefully people with access to this account are also trustworthy, or else we are in trouble.

In any system, the *trusted computing base* (TCB) is that portion of the system that must operate correctly in order for the security goals of the system to be assured. We have to rely on every component in the TCB to work correctly. However, anything that is outside the TCB isn't relied upon in any way: even if it misbehaves or operates maliciously, it cannot defeat the system's security goals. Indeed, we can take the latter statement as our definition of the TCB: the TCB must be large enough so that nothing outside the TCB can violate security.

Example: Suppose the security goal is that only authorized users are allowed to log into my system using SSH. What is the TCB? Well, the TCB includes the SSH daemon, since it is the one that makes the authentication and authorization decisions—if it has a bug (say, a buffer overrun), or if it was programmed to behave maliciously (say, the SSH implementor has included a backdoor in it), then it will be able to violate my security goal (e.g., by allowing access to unauthorized users). That's not all. The TCB also includes the operating system, since the operating system has the power to tamper with the operation of the SSH daemon (e.g., by modifying its address space). Likewise, the CPU is in the TCB, since we are relying upon the CPU to execute the SSH daemon's machine instructions correctly. Suppose a web browser application is installed on the same machine; is the web browser in the TCB? Hopefully not! If we've built the system in a way that is at all reasonable, the SSH daemon is supposed to be protected (by the operating system's memory protection) from interference by unprivileged applications, like a web browser.

Another example: Suppose that we deploy a firewall at the network perimeter to enforce the security goal that only authorized external connections should be permitted into our internal network. Then in this case the firewall is in the TCB for this security goal. In addition, the deployment of the firewall reflects an assumption that all external connections will pass through it. Thus, the network's physical topology, forwarding and routing configurations, and controls over its access points will also be part of the TCB.

A third example: When we build *access control* into a system, there is always some mechanism that is responsible for enforcing the access control policy. We term such a mechanism as a **reference monitor**. The reference monitor is the TCB for the security goal of ensuring that the access control policy is followed. Basically, the notion of a reference monitor is just the idea of a TCB, specialized to the case of access control.

TCB Design Principles.

Several principles guide us when designing a TCB:

- *Unbypassable:* There must be no way to breach system security by bypassing the TCB.
- *Tamper-resistant:* The TCB should be protected from tampering by anyone else. For instance, other parts of the system outside the TCB should not be able to modify the TCB's code or state. The integrity of the TCB must be maintained.
- *Verifiable:* It should be possible to verify the correctness of the TCB. This usually means that the TCB should be as simple as possible, as generally it is beyond the state of the art to verify the correctness of subsystems with any appreciable degree of complexity.

Keeping the TCB **simple and small** is good (excellent) practice. The less code you have to write, the fewer chances you have to make a mistake or introduce some kind of implementation flaw. Industry standard error rates are 1–5 defects per thousand lines of code. Thus, a TCB containing 1,000 lines of code might have 1–5 defects, while a TCB containing 100,000 lines of code might have 100–500 defects. If we need to then try to make sure we find and eliminate any defects that an adversary can exploit, it's pretty clear which one to pick![1] The lesson is to shed code: design your system so that as much code as possible can be *moved outside* the TCB.

Benefits of TCBs.

Who cares about all this esoteric stuff about TCBs? The notion of a TCB is in fact a very powerful and pragmatic one. The concept of a TCB allows a primitive yet effective form of modularity. It lets us separate the system into two parts: the part that is security-critical (the TCB), and everything else.

[1] Windows XP consisted of about 40 million lines of code—all of which were in the TCB. Yikes!

This separation is a big win for security. Security is hard. It is really hard to build systems that are secure and correct. The more pieces the system contains, the harder it is to assure its security. If we are able to identify a clear TCB, then we will know that only the parts in the TCB must be correct for the system to be secure. Thus, when thinking about security, we can focus our effort where it really matters. And, if the TCB is only a small fraction of the system, we have much better odds at ending up with a secure system: the less of the system we have to rely upon, the less likely that it will disappoint.

Let's do a concrete example. You've been hired by the National Archives to help with their email retention system. They're chartered with saving a copy of every email ever sent by government officials. They want to ensure that, once a record is saved, it cannot be subsequently deleted or destroyed. For instance, if someone is investigated, they are worried about the threat that someone might try to destroy embarassing or incriminating documents previously stored in the archives. The security goal is to prevent this kind of after-the-fact document destruction.[2] So, you need to build a document storage system which is "append-only": once a document is added to the collection, it cannot be removed. How are you going to do it?

One possible approach: You could augment the email program sitting on every government official's desktop computer to save a copy of all emails to some special directory on that computer. What's the TCB for this approach? Well, the TCB includes every copy of the email application on every government machine, as well as the operating systems, other privileged software, and system administrators with root/Administrator-level privilege on those machines. That's an awfully large TCB. The chances that everything in the TCB works correctly, and that no part of the TCB can be subverted, don't sound too good. After all, any system administrator could just delete files from a special directory after the fact. It'd be nice to have a better solution.

A different idea: We might set up a high-speed networked printer, one that prints from continuous rolls of paper. An email will be considered added to the collection when it has been printed. Let's feed a giant roll of blank paper into the printer. Once the paper is printed, the paper might spool out into some giant canister. We'll lock up the room to make sure no one can tamper with the printouts. What's the TCB in this system? The TCB includes the physical security of the room. Also, the TCB includes the printer: we're counting on it to be impossible for the printer to be driven in reverse and overwrite previously printed material.

This scheme can be improved if we add a ratchet in the paper spool, so that the spool can only rotate in one direction. Thus, the paper feed cannot be reversed: once something is printed on a piece of paper and it scrolls into the canister, it cannot be later overwritten. Given such a ratchet, we no longer need to trust the printer. The TCB includes only this one little ratchet gizmo, and the physical security for the room, but nothing else. Neat! That sounds like something we could secure.

[2] Assume that you don't have to worry about the problem of making sure that documents are entered into the archive in the first place. Maybe users will mostly comply initially, and we're only really worried about a "change of mind." Or, maybe it is someone else's job to ensure that the necessary documents get into the archive.

One problem with this one-way ratcheted printer business is that it involves paper. A *lot* of paper. (Government bureaucrats can generate an awful lot of email.) Also, paper isn't keyword-searchable. Instead, let's try to find an electronic solution.

An all-electronic approach: We set up a separate computer that is networked and runs a special email archiving service. The service accepts connections from anyone; when an email is sent over such a connection, the service adds the email to its local filesystem. The filesystem is carefully implemented to provide write-once semantics: once a file is created, it can never be overwritten or deleted. We might also configure the network routers so that hosts cannot connect to any other port or service on that computer. What's in the TCB now? Well, the TCB includes that computer, the code of this server application, the operating system and filesystem and other privileged code on this machine, the system administrators of this machine, the packet firewall, the physical security mechanisms (locks and so on) protecting the machine room where this computer is located, and so on. The TCB is bigger than with a printer—but it is vastly better than an approach where the TCB includes all the privileged software and privileged users on every government machine. This sounds manageable.

In summary, some good principles are:

- Know what is in the TCB. Design your system so that the TCB is clearly identifiable.
- Try to make the TCB unbypassable, tamper-resistant, and as verifiable as possible.
- Keep It Simple, Stupid (KISS). The simpler the TCB, the greater the chances you can get it right.
- Decompose for security. Choose a system decomposition/modularization based not just on functionality or performance grounds—choose an architecture that makes the TCB as simple and clear as possible.

TOCTTOU Vulnerabilities.

It is worth knowing about a type of concurrency risk that often has particular relevance when enforcing access control policies such as when using a reference monitor. Consider the following code:

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

This code is trying to open a file, but only if it is a regular file (e.g., not a symlink, not a directory, not a special device). On Unix, the `stat()` call is used to extract meta-data about the file, including whether it is a regular file or not. Then, the `open()` call is used to open the file.

The flaw in the above code is that it assumes the state of the filesystem will remain unchanged between the `stat()` and the `open()`. However, this assumption may be faulty if there is any other code that might execute *concurrently.* Suppose an attacker can change the file that `path` refers to after the call to `stat()` completes, but *before* `open()` is invoked. If `path` refers to a regular file when the `stat()` is executed, but refers to some other kind of file when the `open()` is executed, this bypasses the check in the code! If that check was there for a security reason, the attacker may be able to subvert system security.

This is known as a *Time-Of-Check To Time-Of-Use* (TOCTTOU) vulnerability, because the meaning of `path` changed from the time when it is checked (the `stat()`) and the time when it is used (the `open()`). In Unix, this often comes up with filesystem calls, because sequences of system calls do not execute in an atomic fashion, and the filesystem is where most long-lived state is stored. However, the risk is not specific to files. In general, TOCTTOU vulnerabilities can arise anywhere that there is mutable state that is shared between two or more entities. For instance, multi-threaded Java servlets and applications are at risk for this kind of flaw.

Leveraging Modularity.

A well-designed system will be decomposed into modules, where modules interact with each other only through well-defined interfaces. Each module should perform a clear function; the essence is conceptual clarity of what it does (what functionality it provides), not how it does it (how it is implemented).

Sound modular design can also significant strengthen the security properties of a system by providing forms of *isolation*—keeping potential problems localized, and minimizing the assumptions made between components.

For example, consider a network server that listens on a port below 1024. A good design might partition the server into two distinct pieces, each its own process: a small start-up wrapper, and the application itself. Because binding to a port in the range 0–1023 requires root privileges on Unix systems, the wrapper could run as root, bind to the desired port to some file descriptor, and then spawn the application as a separate process and pass it the file descriptor. The application itself could then run as a non-root user, limiting the damage if the application is compromised. We can write the wrapper in just a few dozen lines of code, so we should be able to validate it quite thoroughly.

As another example, consider structuring a web server as a composition of two modules. One module might be responsible for interacting with the network; it could handle incoming network connections and parse them to identify the requested URL. The second module might translate the URL into a filename and read it from the filesystem. Note that the first module can be run with no privileges at all (assuming it is started by a root wrapper that binds to port 80). The second module might be run as some special userid (e.g., `www`), and we might ensure that only documents intended to be publicly visible are readable by user `www`. This approach then leverages the file access controls provided by the operating system so that even if the second module is subverted, the attacker cannot harm the rest of the system.

These practices are often known under the name *privilege separation*, because we split the architecture up into multiple modules, some privileged and some unprivileged.