

Patterns for Building Secure Software

This lecture will show you a number of 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? You will also see a powerful concept, the notion of a trusted computing base (TCB).

1 The Trusted Computing Base (TCB)

Earlier in this class we introduced the notion of *trusted* and *trustworthy* components. 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 she is 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, 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 connections should be permitted into our internal network. Then, in this case, the firewall is the TCB for this security goal.

A third example: When we build access control into a system, there is always some mechanism that is re-

sponsible for enforcing the access control policy. As you may remember from the lecture notes on firewalls, this mechanism is known as a *reference monitor*. The reference monitor is the TCB for 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.

In fact, the three guiding principles for a reference monitor also apply to TCB. We repeat them here. A TCB should be:

- *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 practice. The less code you have to write, the less 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 1000 lines of code might have 1–5 defects, while a TCB containing 100,000 lines of code might have 100–500 defects. I know which I'd pick. (Windows XP consists of about 40 million lines of code, all of which is in the TCB. Yikes!) The lesson is to shed code: design your system so that as much of the code can be moved outside the TCB.

2 TCBs: What are they good for?

Who cares about all this esoteric stuff about TCBs? Actually, the notion of a TCB is 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.

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 security. 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 embarrassing or incriminating documents previously stored in the archives. The security goal is to prevent this kind of after-the-fact document destruction.¹ 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

¹Assume 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.

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 line-printer. 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 later be 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.

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 a little 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. I think you've earned your consulting fee.

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 unby-passable, 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.

3 TOCTTOU Vulnerabilities

It is worth knowing about a concurrency risk, one that is often particularly relevant when enforcing access control policies. Consider the following code:

```

int openregularfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to 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 referred to by `path` 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 system calls are not atomic and the filesystem is where most long-lived state is stored. However, this 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.

4 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).

The granularity of modules is dependent on the system and language. A module typically has state and code. For instance, in an object-oriented language like Java, a module might consist of a class (or a few closely related classes). In C, a module might be in its own file and contain some clear external interface, along with many internal functions that are not externally visible or callable.

Module design is as much about interface design as anything else. The interface is the contract between caller and callee; hopefully, it should change less often than the implementation of the module itself. A caller should only need to understand the interface. Modules should interact only through the defined interface; for instance, you shouldn't use global variables to communicate information from caller to callee. Think of a module as a blob; the interface is its surface area, and the implementation is its volume. Thoughtful design is often characterized by narrow and conceptually clean interfaces and modules with a low surface area to volume ratio.

When you decompose the system into modules, here are some suggestions that will improve security:

- *Minimize the harm that could be caused by failure of a module.* Ensure that even if one module is

penetrated (e.g., by a buffer overrun) or behaves unexpectedly (e.g., due to a bug in its implementation), then the damage is contained as much possible. Draw a security perimeter around each module. Protect modules from each other, so that one misbehaving module cannot cause other modules's behavior to deviate from what was expected by the programmer. Plan for failure: think in advance about what the consequences of a compromise of each module might be, and structure the system to reduce these consequences.

For instance, a monolithic architecture that places all modules in a common address space is an unnecessary security risk, because if one module is compromised then all others can be penetrated as well. Some languages (e.g., Java) provide mechanisms for isolating modules from each other using type-safety; with legacy languages (like C), you may need to place each module in its own process to protect it.

- *Follow the principle of least privilege at a module granularity.* Provide each module the least privilege that is necessary to get its job done. Architect the system so that most modules need only minimal privileges.

Think about whether there is a way to structure the system so that the complex computations that will require a lot of code are isolated in modules with few privileges. Modules with extra privileges should have very little code. The more privilege a module is given, the greater the confidence we will want to have that it is correct, and more confidence generally requires less code.

Example: A network server that listens on a port below 1024 might be broken up into two pieces: a small start-up wrapper, and the application itself. Because binding to a port in the range 0–1023 requires root privileges, the wrapper could run as root, bind to the desired port to some file descriptor, and then spawn the application 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. The wrapper can be written in only a few dozen lines of code, so we should be able to validate it quite thoroughly.

Example: A web server might be structured 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 then leverages the file access controls provided by the operating system so that even if the second module is penetrated, 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.

5 Optional: Defensive Consistency

We've discussed *defensive programming* before, but now I'd like to look at twist on the basic concept. As mentioned before, the simplest situation is where we are writing a module M that provides functionality to a single client. In this case, M should strive to provide useful responses as long as the client provides valid inputs to M . However, if the client provides an invalid input to M , then M is released from any obligation to provide useful output. The contract between M and its client determines what inputs are valid and what inputs are invalid.

The situation becomes more elaborate if the module M provides some functionality to *multiple* clients that do not necessarily trust each other. In this case, it is important for M to defend itself against malicious clients. It is also frequently helpful for M to ensure that one malicious client cannot disrupt other clients. Thus, when M is performing some function on behalf of a client, there are two cases:

- *If a well-behaved client supplies valid inputs, M should provide correct and useful results to that client.* When M is invoked with a valid and meaningful request, M must respond correctly. This is primarily a functionality requirement. It may also be relevant to security, because the client may be relying upon M to do its job correctly.
- *If a misbehaving client supplies invalid inputs, M does not need to provide useful service to this client, but other clients should not be disrupted.* When M is invoked with meaningless, unexpected, or malicious input, there is no requirement that M provide a useful response to this client. The misbehaving client has violated its contract with M , and thus has no right to expect any particular response from M . However, M should protect itself from such requests, and M should not allow its internal state to become corrupted or harmful side effects to occur. M should maintain the consistency of its internal data structures no matter what inputs it receives. Also, M should ensure that other clients are not disrupted by requests from a malicious client, and that all well-behaved clients continue to receive correct and useful results.

Following these principles makes it easier to ensure that the resulting system will be secure. If we didn't follow these principles, then each client of M would be relying upon the proper behavior and security of all of M 's other clients, which would make it hard to reason about system security. For instance, if Alice and Bob are two clients of M , and Alice becomes compromised, it would be nice to know that Alice cannot attack M in a way that violates Bob's security; and that's exactly what the principles above are intended to achieve.

There is a special case where we do not have to worry about multiple clients. Suppose M computes a pure function, with no internal state and performing no I/O, so that its output depends deterministically on its input. In this case, we do not need to worry about one client disrupting another client or corrupting M 's state. Thus, functional programming can simplify the task of defensive programming.