Principles of Secure Software

This lecture will show you a number of principles for building secure systems. First, we will show one powerful concept, the notion of a trusted computing base (TCB); then, we will describe thirteen useful principles. The principles are neither necessary nor sufficient to ensure the design of a secure system, but they are often very helpful nonetheless.

This lecture focuses primarily on 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? That's the focus of this lecture. We reserve the question of what to do at implementation time until next lecture.

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. 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 responsible 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 embarassing 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":

[1] 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.

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. We'd better find 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. We might have a giant roll of blank paper, which feeds through 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, it includes the printer. More specifically, suppose we arrange to add a ratchet in the paper spool so that the spool can only rotate in one direction (once printed, the paper feed cannot be reversed). Then the rest of the printer doesn't need to be trusted. Wow! 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 server. The computer accepts connections to that server; when an email is sent over that connection, the computer 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 set up a packet filter so that no other connections to that server are allowed. What's in the TCB now? Well, the TCB includes the computer running this server, 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 as unbypassable, tamper-resistant, and 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 Principles for Secure Systems

Here are some general principles for secure system design.[2]

- *Security is economics.* No system is completely, 100% secure against all attacks. Rather, systems may only need to resist a certain level of attack. There is no point buying a $10,000 firewall to protect $1,000 worth of trade secrets.

  Also, it is often helpful to quantify the level of effort that an attacker would need to expend to break the system. Adi Shamir once wrote, "There are no secure systems, only degrees of insecurity." A lot of the science of computer security comes in measuring the degree of insecurity.

  Analogy: Safes come with a rating of their level of security. For instance, a consumer-grade safe might indicate that it will resist attack for up to 5 minutes by anyone without tools. A high-end safe might be rated TL-30: it is secure against a burglar with safecracking tools and limited to 30 minutes access to the safe. (With such a safe, we know that we need to hire security guards who are able to respond to any intrusion within 30 minutes.)

  A corollary of this principle is you should focus your energy on securing the weakest links. Security is like a chain: it is only as secure as the weakest link. Attackers follow the path of least resistance, and they will attack the system at its weakest point. There is no sense putting an expensive high-end deadbolt on a screen door; an attacker isn't going to bother trying to pick the lock when he can just rip out the screen and step through.

- *Least privilege.* Give a program the set of access privileges that it legitimately needs to do its job—but nothing more. Try to minimize how much privilege you give each program and system component.

  Least privilege is an enormously powerful approach. It doesn't reduce the probability of failure, but it can reduce the expected cost of failures. The less privilege that a program has, the less harm it can do if it goes awry or runs amok. You can think of this as the computer-age version of the shipbuilder's notion of "watertight compartments": even if one compartment is breached, we want to minimize the damage to the integrity of the rest of the system.

  For instance, the principle of least privilege can help reduce the damage caused by buffer overruns. If a program is compromised by a buffer overrun attack, then it will probably be completely taken over by an intruder, and the intruder will gain all the privileges the program had. Thus, the fewer privileges that a program has, the less harm is done if it should someday be penetrated by a buffer overrun attack.

  Example: How does Unix do, in terms of least privilege? Answer: Pretty lousy. Every program gets all the privileges of the user that invokes it. For instance, if I run a editor to edit a single file, the editor receives all the privileges of my user account, including the powers to read, modify, or delete all my files. That's much more than is needed; strictly speaking, the editor probably only needs access to the file being edited to get the job done.

  Example: How is Windows, in terms of least privilege? Answer: Just as lousy. Arguably worse, because many users run under an Administrator account, and many Windows programs require that you be Administrator to run them. In this case, every program receives total power over the whole computer. Folks on the Microsoft security team have recognized the risks inherent in this, and are taking many steps to warn people away from running with Administrator privileges, so things are getting better in this respect.

[2]Many of them are due to Saltzer and Schroeder, who wrote a classic paper in the 1970s with advice on this topic.

- *Use fail-safe defaults.* Use default-deny polices. Start by denying all access, then allow only that which has been explicitly permitted. Ensure that if the security mechanisms fail or crash, they will default to secure behavior, not to insecure behavior.

  Example: A packet filter is a router. If it fails, no packets will be routed. Thus, a packet filter fails safe. This is good for security. It would be much more dangerous if it had fail-open behavior, since then all an attacker would need to do is wait for the packet filter to crash (or induce a crash) and then the fort is wide open.

  Example: Long ago, SunOS machines used to ship with + in their `/etc/hosts.equiv`, which allowed anyone with root access on any machine on the Internet to log into your machine as root. Irix machines used to ship with `xhost +` in their X Windows configuration files by default. This violates of the principle of fail-safe defaults, since the machines came with an out-of-the-box configuration that was insecure by default.

- *Separation of responsibility.* Split up privilege, so no one person or program has complete power. Require more than one party to approve before access is granted.

  Examples: In a nuclear missile silo, two launch officers must agree before the missile can be launched.

  Example: In a movie theater, you pay the teller and get a ticket stub; then when you enter the movie theater, a separate employee tears your ticket in half and collects one half of it, putting it into a lockbox. Why bother giving you a ticket that 10 feet later is going to be collected from you? One answer is that this helps prevent insider fraud. Tellers are low-paid employees, and they might be tempted to under-charge a friend, or to over-charge a stranger and pocket the difference. The presence of two employees helps keep them both honest, since at the end of the day, the manager can reconcile the number of ticket stubs collected against the amount of cash collected and detect some common shenanigans.

  Example: In many companies, purchases over a certain amount must be approved both by the requesting employee and by a separate purchasing office. This control helps prevent fraud, since it is less likely that both will collude and since it is unlikely that the purchasing office will have any conflict of interest in the choice of vendor.

- *Defense in depth.* This is a closely related principle. There's a saying that you can recognize a security guru who is particularly cautious if you see someone wearing both a belt and a set of suspenders. (What better way to avoid getting caught with your trousers around your ankles?) The principle is that if you use multiple redundant protections, then all of them would need to be breached before the system's security will be endangered.

- *Psychological acceptability.* It is important that your users buy into the security model.

  Example: Suppose the company firewall administrator gains a reputation for capriciously, for no good reason, blocking applications that the engineers need to use to get their job done. Pretty soon, the engineers are going to view the firewall as damage and route around it, maybe setting up tunnels, or bypassing it in any number of other ways. This is not a game that the firewall administartor is going to win. No system can remain secure for long when all its users actively seek to subvert it.

  Example: The system administrator issues an edict that, henceforth, all passwords will be automatically generated unmemorizable strings that are at least 17 characters long, and must be changed once a month. What's likely to happen is that users will simply write down their password on a yellow sticky attached to their monitor, visible to anyone who looks. Such well-intentioned edicts can ultimately turn out to be counter-productive.

- *Usability.* A related topic: Security systems must be usable by ordinary people, and must be designed to take into account the role humans will play.

  Example: Your web browser pops up security warnings all the time, with vague alarming warnings but no clear indication of what steps you can take and no guidance on how to handle the risk. What are you going to do? If you're like most of the user population, you're soon going to learn to always click "Ok" any time a security dialogue box pops up.

  Example: The NSA's cryptographic equipment stores its key material on a small physical token. This token is built in the shape of an ordinary door key. To activate an encryption device, you insert the key into a slot on the device and turn the key. This interface is intuitively understandable, even for 18-year-olds soldiers out in the field with minimal training in cryptography.

- *Ensure complete mediation.* When enforcing access control policies, make sure that you check *every* access to *every* object.

  Caching is a slightly sticky subject. In some cases, you can get away with not checking every access and allowing security decisions to be cached, but beware. If the context relevant to the security decision changes, and the cache entry isn't invalidated, then someone might get away with accessing something they shouldn't.

- *Least common mechanism.* Be careful with shared code. The assumptions originally made might no longer be valid. The threat model may have changed.

  Example: In the early days, the Internet was populated only by researchers, who mostly trusted each other. Many networking protocols designed during those days made assumptions that all other network participants were benign and would not try to harm others. Of course, today the Internet is populated by millions of users, who do not always have such benign intent; consequently, many network protocols designed long ago are now suffering under the strain of attack. Spam is one well-known example of this syndrome.

- *Detect if you can't prevent.* If you can't prevent break-ins, at least detect them (and, where possible, provide a way to recover or to identify the perpetrator). Save audit logs so that you have some way to analyze break-ins after the fact.

  Example: FIPS 140-1 sets out a federal standard on tamper-resistant hardware. Type III devices—the highest level of security in the standard—are intended to be tamper-resistant. However, Type III devices are very expensive. Type II devices are only required to be tamper-evident, so that if someone tampers with them, this will be visible (e.g., a seal will be visibly broken). This means they can be built more cheaply and used in a broader array of applications.

- *Orthogonal security.* We saw this one before: security mechanisms that are implemented orthogonally (transparently) to the rest of the system are useful in protecting legacy systems, and allow to improve assurance by composing multiple such mechanisms in series.

- *Don't rely on security through obscurity.* The phrase 'security through obscurity' has come to be understood to refer to systems that rely on the secrecy of their design, algorithms, or source code to be secure[3]. The problem with this is that it is often very hard to keep the design of the system secret from a dedicated adversary. For instance, every running installation is going to have binary

[3] One might hear reasoning like: "this system is so obscure, only 100 people around the world understand anything about it, so what are the odds that an adversary will bother attacking it?" One problem with such reasoning is that such an approach is self-defeating. As the system becomes more popular, there will be more incentive to attack it, and then we cannot rely on its obscurity to keep attackers away.

executable code, and it is tedious but not all that difficult to disassemble and reverse-engineer such code. Also problematic is that it is very difficult to assess, with any confidence, the chances that the secret will leak or the difficulty of learning the secret. Moreover, it's a real bummer if this secret ever leaks: it is often hard to update widely-deployed systems, so there may be no recourse if someone ever succeeds in reverse-engineering the code. Historically, security through obscurity has a lousy track record: many systems that have relied upon the secrecy of their code or design for security have failed miserably.

  This doesn't mean that open-source applications are necessarily more secure than closed-source applications. But it does mean that you shouldn't trust any system that relies on security through obscurity, and you should probably be skeptical about claims that keeping the source code secret makes the system significantly more secure.

- *Design security in, from the start.* Trying to retrofit security to an existing application after it has already been spec'ed, designed, and implemented is usually a very difficult proposition. At that point, you're stuck with whatever architecture has been chosen, and you don't have the option of decomposing the system in a way that ensures least privilege, separation of privilege, complete mediation, defense in depth, and other good properties. Backwards compatibility is often particularly painful, because you are stuck with supporting the worst insecurities of all previous versions of the software.

Finally, I'll end with three principles that are widely accepted in the cryptographic community (although not often articulated), and that seem useful in computer security as well.

- *Conservative design.* Systems should be evaluated according to the worst security failure that is at all plausible, under assumptions favorable to the attacker[4]. If there is any plausible circumstance under which the system can be rendered insecure, then it would be prudent to seek a more secure system.
- *Kerkhoff's principle.* Cryptosystems should remain secure even when the attacker knows all internal details of the system. The key should be the only thing that must be kept secret. If your secrets are leaked, it is a lot easier to change the key than to change the algorithm.
- *Proactively study attacks.* We should devote considerable effort to trying to break our own systems; this is how we gain confidence in their security. Also, because security is a game where the attacker gets the last move, and where it can be very costly if a security hole is discovered after a system is widely deployed, it pays to try to identify attacks before the bad guys find them, so that we have some lead time to close the security holes before they are exploited in the wild.

[4] I thank Doug Gwyn for this formulation.