

An online version is available at <https://textbook.cs161.org>.

Textbook by [David Wagner](#), [Nicholas Weaver](#), [Peyrin Kao](#),
Fuzail Shakir, Andrew Law, and [Nicholas Ngai](#)

Additional contributions by Noura Alomar, Sheqi Zhang, and [Shomil Jain](#)

Last update: June 20, 2021

Contact for corrections: cs161-staff@berkeley.edu

The security principles module of this course is the least technically involved and is essentially covered in the first one or two lectures. Basically, it's a list of "things to remember" when thinking about security. Vulnerabilities will usually violate one or more of these rules. Your job on exams is to determine which security principle best fits a given scenario.

We teach these security principles because they are prevalent across all avenues of the security field and you may hear about them in academic literature, or just in informal conversation in the class.

1 Principles for Building Secure Systems

In this chapter, we will look at some general principles for secure system design.¹ These ideas also allow us to examine existing systems to understand their security properties.

1.1 Know your threat model

A threat model is a model of who your attacker is and what resources they have. Attackers target systems for various reasons, be it money, politics, fun, etc. Some aren't looking for anything logical, some attackers just want to watch the world burn.

Take, for example your own personal security. Understanding your threat model has to do with understanding who and why might someone attack you; criminals, for example, could attack you for money, teenagers could attack you for laughs (or to win a dare), governments might spy on you to collect intelligence (but you probably are not important enough for that just yet), or intimate partners could spy on you.

Once you understand who your attacker is and what resources they might possess, there are some common assumptions that we take into account for attackers:

¹Many of these principles are due to Saltzer and Schroeder, who wrote a classic paper in the 1970s with advice on this topic.

1. The attacker can interact with your systems without anyone noticing, meaning that you might not always be able to detect the attacker tampering with your system before they attack. 2. The attacker has some general information about your system, namely the operating system, any potential software vulnerabilities, etc. 3. The attacker is persistent and lucky; for example, if an attack is successful 1/1,000,000 times, the attacker will try 1,000,000 times. 4. The attacker has the resources required to undertake the attack (up to an extent). This will be touched on in "Securities is Economics", but depending on who your threat model is, assume that the attacker has the ability and resources to perform the attack. 5. The attacker can coordinate several complex attacks across various systems, meaning that the attacker does not have to mount only a single attack on one device, but rather can attack your entire network at the same time. 6. Infrastructure is pretty difficult to compromise, meaning that an attack that requires the attacker to break the internet is a lot less worrisome than one that requires the attacker to break a single computer. 7. Every system is a potential target. For example, a casino was once hacked because a fish-tank thermometer was hacked within the network.

Finally, be extremely vigilant when dealing with old code as the assumptions that were originally made might no longer be valid and the threat model might have changed. When the internet was first created, for example, it was mostly populated by academics who (mostly) trusted one another. As such, several networking protocols made the assumption that all other network participants could be trusted and were not malicious. Today however, the Internet is populated by billions of devices, some of whom are malicious. As such, many network protocols that were designed a long time ago are now suffering under the strain of attack.

1.2 Consider Human Factors

The key idea here is that security systems must be usable by ordinary people, and therefore must be designed to take into account the role that humans will play. As such, you must remember that programmers make mistakes and will use tools that allow them to make mistakes (like C and C++). Similarly, users like convenience; if a security system is unusable and not user-friendly, no matter how secure it is, it will go unused. Users will find a way to subvert security systems if it makes their lives easier.

No matter how secure your system is, it all comes down to people. Social engineering attacks, for example, exploit other people's trust and access for personal gain. The takeaway here is to consider the tools that are presented to users, and try to make them fool-proof and as user-friendly as possible.

For example, your computer pops up with a notification that tells you it needs to restart to "finish installing important updates"; if you are like a majority of the user population, you likely click "remind me later", pushing off the update. If the computer is attempting to fix a security patch, the longer the update gets pushed, the more time your computer is vulnerable to an attack. However, since the update likely inconveniences the user, they forego the extra security for convenience.

Another example is that 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.

1.3 Security is economics

No system is completely, 100% need to be protected against a certain level of attacks. Since more security usually costs more money to implement, the expected benefit of your defense should be proportional to the expected cost of the attack. Essentially, there is no point putting a \$100 lock on a \$1 item.

To understand this concept, we can think about physical safes, which come with a rating of their level of security. For instance, a consumer grade safe, a TL-15, might indicate that it will resist attacks for up to 15 minutes by anyone with common tools, and might cost around \$3,000, while a TL-30, a safe that would resist attacks for up to 30 minutes with common tools might cost around \$5000. Finally, a TXXL-60 (a super high-end safe), might resist attacks for up to 60 minutes with common tools, a cutting torch, and up to 4 oz of explosives, and would cost upwards of \$50,000. The idea is that security usually comes at a cost. A more secure safe is going to cost you more than a less secure safe. With infinite money, you could use the best safe available to lock all your valuables, but since you don't have infinite money, you must determine how valuable the thing you want to protect is, and you must judge how much you are willing to pay to protect it. This illustrates that security is often a cost-benefit analysis where someone needs to make a decision regarding how much security is worth.

A corollary of this principle is you should focus your energy on securing the weakest links. Security is like a chain: a system 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; attackers aren't going to bother trying to pick the lock when they can just rip out the screen and step through.

1.4 Detect if you can't prevent

If prevention is stopping an attack from taking place, detection is simply learning that the attack has taken place, and response would be doing something about the attack. The idea is that if you cannot prevent the attack from happening, you should at least be able to know that the attack has happened. Once you know that the attack has happened, you should find a way to respond, since detection without response is pointless.

For example, the Federal Information Processing Standard (FIPS) are publicly announced standards developed for use in computer systems by various government contractors. 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.

A *false positive* is when you detect something when there is nothing there. A *false negative* is when you don't detect something when there is something there. False positives and false negatives are usually the real cost of detection since false positives require some type of response (and too many false positives can cause systems to be removed), and false negatives are a failure of detection and can be catastrophic. All reasonable detectors will have a non-zero false positive and false negative rate.

When dealing with response, you should always assume that bad things will happen, and therefore prepare your systems for the worst case outcome. You should always plan security in a way that lets you get back to some form of a working state. For example, keeping offsite backups of computer systems is a great idea; should your computer and house catch on fire, for example, it should be no big deal since all your data is backed-up in some other location.

1.5 Defense in depth

The key idea of defense in depth is that multiple types of defenses should be layered together so an attacker would have to breach all the defenses to successfully attack a system.

Take, for example, a castle defending its king. The castle has high walls. Behind those walls might be a moat, and then another layer of walls. Layering multiple simple defensive strategies together can make security stronger. However, defense in depth is not foolproof—no amount of walls will stop siege cannons from attacking the castle. Also, beware of diminishing returns—if you've already built 100 walls, the 101st wall may not add enough additional protection to justify the cost of building it (security is economics).

Another example of defense in depth is through a composition of detectors. Say you had two detectors, D_1 and D_2 , which have false positive rates of FP_1 and FP_2 respectively, and false negative rates of FN_1 and FN_2 , respectively. One way to use the two detectors would be to have them in parallel, meaning that either detector going off would trigger a response. This would increase the false positive rate and decrease the false negative rate. On the other hand, we could also have the detectors in series, meaning that both detectors have to alert in order to trigger a response. In this case, the false positive rate would decrease while the false negative rate would increase.

1.6 Least privilege

Consider a research building home to a team of scientists as well as other people hired to maintain the building (janitors, IT staff, kitchen staff, etc.) Some rooms with sensitive research data might be only accessible to trusted scientists. These rooms should not be accessible to the “other hired people” e.g. janitors. For best security practices, any one party should only have as much privilege as it needs to play its intended role.

In technical terms, 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 becomes subverted.²

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.

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 an 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.

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 have taken many steps to warn people away from running with Administrator privileges, so things have gotten better in this respect.

1.7 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.

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

Another example of this principle in action is in a movie theater, where 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.

The idea here is that if you need to have privilege, require multiple parties to work together to exercise that privilege since it is more likely for a single party to be malicious than for all of the parties to be malicious and collude with one another.

Ensure complete mediation

²There are many real-world analogies to this principle. For example, we don't require exam proctors to carry sidearms when a notepad or a camera will suffice.

When enforcing access control policies, make sure that you check *every* access to *every* object. This kind of thinking is helpful to detect where vulnerabilities could be. As such, you have to ensure that all access is monitored and protected. One way to accomplish this is through a *reference monitor*, which is a single point through which all access must occur.

1.8 Shannon's Maxim

Shannon's Maxim states that the attacker knows the system that they are attacking.

"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 issue with this, however, is that it is extremely brittle and it often proves to be difficult to keep the design of a system secret from a sufficiently motivated attacker. 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.

As such, you should never rely on obscurity as part of your security. Always assume that the attacker knows every detail about the system that you are working with (including its algorithms, hardware, defenses, etc.)

1.9 Use fail-safe defaults

Choose default settings that "fail safe", balancing security with usability when a system goes down. When we get to firewalls, you will learn about default-deny policies, which start by denying all access, then allowing only those which have been explicitly permitted. Ensure that if the security mechanisms fail or crash, they will default to secure behavior, not to insecure behavior.

For example, firewalls must explicitly decide to forward a given packet or else the packet is lost (dropped). If a firewall suffers a failure, no packets will be forwarded. Thus, a firewall 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 firewall to crash (or induce a crash) and then the fort is wide open.

1.10 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

³One 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.

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 can be stuck with supporting the worst insecurities of all previous versions of the software.

Finally, let's examine three principles that are widely accepted in the cryptographic community (although not often articulated) that can play a useful role in considering computer system security as well.

1.11 Conservative design

Systems should be evaluated according to the worst security failure that is at all plausible, under assumptions favorable to the attacker. If there is any plausible circumstance under which the system can be rendered insecure, then it is prudent to consider seeking a more secure system. Clearly, however, we must balance this against *Security is economics*: that is, we must decide the degree to which our threat model indicates we indeed should spend resources addressing the given scenario.

1.12 Kerckhoffs' principle

Cryptosystems should remain secure even when the attacker knows all internal details of the system.⁴ The **key** should be the only thing that must be kept secret, and the system should be designed to make it easy to change keys that are leaked (or suspected to be leaked). If your secrets are leaked, it is usually a lot easier to change the key than to replace every instance of the running software. (This principle is closely related to *Don't rely on security through obscurity*.)

1.13 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.

1.14 The Trusted Computing Base (TCB)

Now that you understand some of the important principles for building secure systems, we will try to see what you can do at design time to implement these principles and improve security. The question we want to answer is 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, also known as the TCB.

⁴We'll discuss the notions of cryptography and cryptosystems more in subsequent lectures (Chapter 5).

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. Generally, the TCB is made to be as small as possible since a smaller, simpler TCB is easier to write and audit.

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, or if it was programmed to behave maliciously, then it will be able to violate my security goal by allowing access to unauthorized users. 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.

1.15 TCB Design Principles

Several principles guide us when designing a TCB:

- *Unbypassable (or completeness)*: There must be no way to breach system security by bypassing the TCB.
- *Tamper-resistant (or security)*: 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 (or correctness)*: 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 excellent. 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!⁵ The lesson is to shed code: design your system so that as much code as possible can be *moved outside* the TCB.

⁵Windows XP consisted of about 40 million lines of code—all of which were in the TCB. Yikes!

1.16 Benefits of TCBs

The notion of a TCB is a very powerful and pragmatic one as it 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 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.

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.

1.17 TOCTTOU Vulnerabilities

A common failure of ensuring complete mediation involves race conditions. The time of check to time of use (TOCTTOU) vulnerability usually arises when enforcing access control policies such as when using a reference monitor. Consider the following code:

```
procedure withdraw(amount w) {  
    // contact central server to get balance  
    1. let b := balance  
    2. if b < w, abort  
  
    // contact central server to set the balance  
    3. set balance := b - w  
    4. dispence $w to the user  
}
```

This code takes as input the amount you want to withdraw, w . It then looks up your bank balance in the database; if you do not have enough money in your account to withdraw the specified amount, then it aborts the transaction. If you do have enough money, it decrements your balance by the amount that you want to withdraw and then dispenses the cash to you.

Suppose that multiple calls to withdraw can take place concurrently (i.e. two separate

ATMs). Also suppose that the attacker can somehow pause the execution of procedure on one ATM.

So suppose that your current account balance is \$100 and you want to withdraw \$100. At the first ATM, suppose you pause it *after* step 2. Then, you go over to the second ATM and proceed to withdraw \$100 successfully (meaning that your account balance should now be \$0). You then go back to the first ATM and unpause the procedure; since the account balance check was completed before you withdrew the money from the second ATM, the first ATM still thinks you have \$100 in your account, and it allows you to withdraw another \$100! So despite your bank account having only \$100 to begin with, you ended up with \$200.

This is known as a *Time-Of-Check To Time-Of-Use* (TOCTTOU) vulnerability, because between the check and the use of whatever state was checked, the state somehow changed. In the above example, between the time that the balance was checked and the time that balance was set, the balance was somehow changed.