What is Security?

6.1600 Course Staff: Henry Corrigan-Gibbs, Yael Kalai, Ben Kettle (TA), Nickolai Zeldovich

Fall 2022

1 Overview

The goal of this course is to give you an overview of the most important "Big Ideas" on securing computer systems. Throughout the course, we will touch on ideas from the fields of computer security, cryptography, and (to some extent) computer systems.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

2 What is Security?

Security is a very broad property, but generally the goal of computer security is to ensure that a particular computer system is "behaves correctly" even in the face of an "adversary" (or "attacker") whose goal is to foil the system.

We will use the terms "adversary" and "attacker" interchangeably throughout this course.

To achieve this goal, we will need some kind of systematic plan. That is, we will have to carefully define what it means for our system to "behave correctly" and we will have to specify the class of "adversaries" against which we want to defend.

For the purposes of this course, we will typically structure our plan in terms of three components: a *goal*, a *threat model*, and an *implementation*.

- **Goal:** The security goal defines what we want our system to achieve. For example, a informal goal might be that "only Alice can read the file F," or "if anyone tampers with the file F, Alice will detect it." As you will learn throughout the course, figuring out exactly what your security goal should be is often quite subtle and challenging.

- **Threat model:** The threat model (or "adversary model") specifies the class of adversaries against which we want to defend. Or, put another way, the threat model specifies the adversary's power: what can the adversary do in trying to break our security goal? By defining a threat model, we can make it tractable to achieve a security goal. For example, in analyzing the security of a password-based login system, our threat model might specify that the adversary can guess passwords, but cannot steal the server.

- **Implementation:** The implementation is how we achieve the goal. For example, we might set permissions on a file F, use Linux to enforce those permissions, and require two-factor authentication to access the system.

Together, the goal and the threat model create our *definition* of security. As such, they cannot be "wrong"—the goal might turn out to not be exactly what we needed, the threat model might not have captured all of the attacks that our system might actually face in the real world, and the system designer might be surprised when they discover this, but together the goal and threat model define the security properties that we set out to achieve. The implementation, on the other hand, can definitely be wrong—if the implementation does not guarantee the goal under the threat model, due to bugs or oversights or supply chain vulnerabilities or anything else, the implementation has a mistake.

When you read about security failures in the news, it is worth trying to understand whether the failure arose from a problem with the goal, the threat model, or the implementation.

2.1 *Security is Hard*

Building secure systems is challenging. There are at least two broad reasons for this.

Secure systems must defend against worst-case behavior. First, a secure system must defend against *all possible attacks* within the scope of the threat model. In contrast, when we are just concerned about functionality or correctness, we are often satisfied with a system that performs well for the cases that users care about. In other words, security is concerned with behavior in *worst-case situations*, while correctness is often about behavior in *expected situations*.

Some developers do worry about correctness of their system for all possible inputs and corner cases; this is is indeed the mindset that is often necessary for security.

For example, suppose that we have a student-information system that should achieve the security goal: "**only** TAs can access grades." It is easy to test if a random sample of TAs can access grades (correctness), but it is much harder to test whether there is some sequence of interactions that allow *anyone else* to access the grades (security).

An implementation can never defend against all possible threats. When we specify a threat model, we delimit the set of adversaries against which our implementation must defend. But real-world adversaries can behave in ways that are outside of our threat model and thereby violate our security goals.

For example, someone besides a TA might be able to access the grades by:

- finding a bug in the server software,
- breaking into a TA's office,
- compromise the TA's laptop,

- stealing the password to an administrator's account,
- tricking a TA into disclosing grades,
- breaking the server's cryptography,
- getting a job at the registrar and making herself a TA.

And the list never ends. Because our threat model cannot capture all possible threats, **security is never perfect**. There will essentially always be *some* attacker than can break your system.

This is why we need a threat model: the threat model defines what kinds of attacks we worry about and which we decide are out of scope.

2.2 *Designing security goals and a threat model*

Specifying security goals and a threat model is all about comparing the cost of defending against an attack with the cost of that attack if it were to happen. It is almost always impractical to exactly calculate these costs, but this framework is useful conceptually. Cheap defenses that block major holes are likely to be worth implementing, but defending against an esoteric RF side channel that could leak unimportant information is likely not.

Building a threat model always requires iterating—you will not get it right on the first try. There is likely to be some type of attack that you didn't consider at first that ends up being important.

2.3 *Designing an implementation*

We will focus largely in this class on techniques that have a big payoff—methods of developing software and tools to use that eliminate entire classes of attacks (or make them much harder).

3 *Examples*

We give a handful of examples of security failures arising from poor choices of security goals or threat model.

3.1 *Bad Goals*

Business-class airfare. An airline tried to add value to their business-class tickets by allowing ticket-holders to change the ticket (i.e., the departure date, origin, and destination) at any time with no fee. One customer realized that they could board the flight *then* change their ticket. The customer could then take an unlimited number of business-class flights for the price of one.

In this case, the airline's goal did not meet their real needs—perhaps they needed to add an additional goal of the form "every time someone takes a flight, we get paid."

Sarah Palin's Email. Sarah Palin had a Yahoo email account, and Yahoo used security questions for password reset—their goal may have been something like "no one can reset a user's password unless they know all of the answers to the user's security questions." (The security questions are typically things like "What is your mother's maiden name?") As it turned out, it was possible to find the answer to all of Palin's account-recovery security questions on the Internet.[1] Yahoo's implementation may have been perfect, but their goal did not provide any meaningful security for certain users.

[1] Kim Zetter. *Palin e-mail hacker says it was easy*. `https://www.wired.com/2008/09/palin-e-mail-ha/`. Sept. 2008.

Instruction Set Architecture (ISA) Specification. When defining ISAs for processors, designers did not think that timing was important, and processors could take a variable number of cycles to execute a particular instruction. This had big benefits for performance and for compatibility, but as we'll talk about later in the semester, researchers have recently exploited this timing variability to perform sophisticated attacks on wide ranges of processors.[2] The processor implementation matched the specification, but the specification itself allowed for this attack.

[2] Mark D. Hill et al. "On the Spectre and Meltdown Processor Security Vulnerabilities". In: *IEEE Micro* 39.2 (2019), pp. 9–19.

Complicated access-control policies. A school in Fairfax, Virginia used an online course-management software with a somewhat complex access-control structure: each teacher is in charge of some class, each class has many students, and each student has many files. Teachers cannot access student's files, and there is also a superintendent that has access to all the files. Teachers are able to change their students' passwords, and are able to add students to their class. It turned out that a teacher could add the superintendent as a student, change the superintendent's password, and then access all files via the superintendent's account. While each of the access-control policies individually sounds reasonable, together they lead to a bad outcome.

Mat Honan's Gmail Account. A journalist for wired named Mat Honan had his Gmail account compromised via a clever attack.[3] Honan had a Gmail account. Gmail's reset-password feature avoided using security questions, and instead used a backup email account. The attacker triggered the reset-password feature, which sent a reset link to Honan's Apple email account.

[3] Mat Honan. *How Apple and Amazon Security Flaws Led to My Epic Hacking*. `https://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/`. Aug. 2012.

The attacker then attempted to gain access to Honan's Apple email account. The attacker triggered the reset-password feature on the

Apple account. Apple's reset-password feature, in turn, required Honan's address and the last four digits of the credit card number. The attacker was able to find Honan's address publicly, but could not easily find his credit-card number.

The attacker found this credit-card information through Honan's Amazon account. Amazon, which knew his credit card number, required a full credit-card in order to reset an account. However, Amazon allowed buying something for a certain account without logging in, so long as you provide a new credit card number. It also allowed *saving* this new credit card number to the user's Amazon account. So, the attacker made a purchase on Amazon with the attacker's credit card. Then, the attacker saved their own credit card number to Honan's Amazon account. Next, the attacker triggered Amazon's reset-password feature, and—using the credit-card number saved to the account—were able to reset Honan's Amazon password and access his Amazon account. The attacker was then able to see the last four digits of Honan's real credit card within his Amazon account, use that to reset his Apple mail account, and then use that to reset Honan's Gmail account.

Complex chains of systems like this can be very hard to reason about, but these interactions ultimately are security-critical.

3.2 Bad Threat Models

Assuming specific strategy: CAPTCHA. CAPTCHAs were designed to be expensive to solve via automation, but easy for a human to read. Indeed it might be expensive to build an optical-character recognition system for CAPTCHAs in general, but attackers who want to bypass CAPTCHAs do not do this. Instead, they set up computer centers in countries where the cost of labor is cheap. Attackers then pay people working in these centers to solve CAPTCHAs.[4] The result is that it costs some fraction of a cent to solve a CAPTCHA. The cost of solving a CAPTCHA is still non-zero, but the cost is much lower than the system designers may have intended.

[4] Marti Motoyama et al. "Re: CAPTCHAs—Understanding CAPTCHA-Solving Services in an Economic Context". In: *Proceedings of the 19th USENIX Security Symposium*. Washington, DC, Aug. 2010.

Computational Power: DES. There used to be an encryption standard called DES that had 2^{56} possible keys. At the time that it was designed, the U.S. government standards agencies asserted that it was secure against even powerful attackers, but today a modern computer can try all 2^{56} keys at only modest cost. (Academic researchers even at the time of DES's design understood that 56-bit keys were not large enough to prevent exhaustive cryptanalysis.[5])

For example, `https://crack.sh/` uses an array of FPGAs to provide a service that exhaustively checks all possible keys.

[5] Whitfield Diffie and Martin E. Hellman. "Exhaustive Cryptanalysis of the NBS Data Encryption Standard". In: *Computer* 6.10 (1977), pp. 74–84.

Because of the weakness of DES against modern computers, everyone using the standard had to upgrade their block ciphers. For example, MIT had to switch from using DES for authentication to newer block ciphers with longer keys, such as AES.

Dependencies: Two-factor authentication (2FA) via SMS. Many 2FA systems use a text message for authentication, but an attacker then just needs to convince the clerk at the AT&T store to give them a new SIM card for your phone number.

These attacks are often called "SIM swapping" or "SIM hijacking".

Software Versions: Xcode. iPhone apps are normally created and compiled on a developer's machine, sent to Apple's App Store, and sent to iPhones from there. iPhone apps are created using a tool called Xcode that is normally downloaded from Apple servers. However, Xcode is a big piece of software and for developers behind China's firewall, it was very slow to download. Someone within China set up a much faster mirror of Xcode, and lots of developers in China used the version of Xcode from this mirror. However, this mirror was not serving exactly Apple's version of Xcode—instead, it was serving a slightly modified version of Xcode that would inject some malicious code into every app that was compiled with it. This took a long time to detect.

The Wikipedia article on XcodeGhost, `https://en.wikipedia.org/wiki/XcodeGhost`, provides more details about this attack.

3.3 Problematic Implementations

Bugs, misconfigurations, and other mistakes are the most common cause of security issues. A rule of thumb to keep in mind is that every 1000 lines of code will have around 1 bug. This is a very rough estimate, but the basic idea is that more code will have more bugs. An effective strategy to reduce security vulnerabilities is to reduce the amount of code in your system.

Missing Checks: iCloud. Apple's iCloud performs many functions—email, calendar, storage, and Find my iPhone. Each of these had their own way of logging in, but across all of them a common goal was to limit the attacker's ability to guess a user's password. To do this, they added rate limiting to all the login interfaces, allowing something like only 10 login attempts per hour—but they forgot the Find my iPhone login interface.[6] Because this authentication code was duplicated all over the place, there were many places to remember to add this rate limiting, but the attacker only needed one weak login interface to brute-force a password. In general, avoiding this repition will make it much easier to build a secure system.

[6] J. Trew. *'Find My iPhone' exploit may be to blame for celebrity photo hacks (update).* `https://www.engadget.com/2014-09-01-find-my-iphone-exploit.html`. Sept. 2014.

Insecure Defaults. When you set up a new service, they almost always come with some defaults to make the setup simpler. Wi-Fi routers come with default passwords, AWS S3 buckets come with default permissions, and so on. These defaults can be convenient, but they are very important to security because many people will forget or neglect to change the default. Because of this, the default becomes the way that the system operates. In order to build a secure system, it is important that the default is secure.

4 *What are the general principles for secure system design?*

4.1 *Goals and Threat Models*

- Create simple, general goals.
- Avoid assumptions (such as "no one else will be able to get a user's SIM card") through better designs.
- Learn and iterate.
- Practice Defense in Depth: don't rely on one single defense for all your security—it is useful to use backup defenses to guard against bugs that will inevitably come up in one defense.

4.2 *Implementation*

- A simpler system will lead to fewer problems.
- Factor out the security-critical part into a small separate system or piece of code (for example, hardware security keys).
- Reuse well-designed code, such as well-tested crypto libraries.
- Understand and test the corner cases.

References

Diffie, Whitfield and Martin E. Hellman. "Exhaustive Cryptanalysis of the NBS Data Encryption Standard". In: *Computer* 6.10 (1977), pp. 74–84.

Hill, Mark D. et al. "On the Spectre and Meltdown Processor Security Vulnerabilities". In: *IEEE Micro* 39.2 (2019), pp. 9–19.

Honan, Mat. *How Apple and Amazon Security Flaws Led to My Epic Hacking*. `https://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/`. Aug. 2012.

Motoyama, Marti et al. "Re: CAPTCHAs—Understanding CAPTCHA-Solving Services in an Economic Context". In: *Proceedings of the 19th USENIX Security Symposium*. Washington, DC, Aug. 2010.

Trew, J. *'Find My iPhone' exploit may be to blame for celebrity photo hacks (update)*. https://www.engadget.com/2014-09-01-find-my-iphone-exploit.html. Sept. 2014.

Zetter, Kim. *Palin e-mail hacker says it was easy*. https://www.wired.com/2008/09/palin-e-mail-ha/. Sept. 2008.