6.1600 Foundations of Computer Security

6.1600 Staff

September 14, 2022

Abstract

This is an introduction to the class

Contents

Chapter 1

What is Security?

1.1 Overview

Big Idea: Big ideas for securing computers.

- Lectures: ask questions!
- Labs (coding) + psets (theory)
- Midterm + final

1.2 What is Security

Security is a very broad property, but generally the goal is that our system is "ok" even in the face of an "adversary" whose goal is to foil the system. To get there, we will need some kind of systematic plan. If we try to do it ad-hoc, the adversary will find a way around. We'll structure our plan in terms of a goal, a threat model, and an implementation.

- Goal: what we want to achieve, such as "only Alice can read the file F"
- Threat Model: delineates our assumptions about the world—what the adversary can and cannot do. By defining the threat model, we can make the problem tractable. For example, "the aversary can guess passwords, but cannot steal the server".
- Implementation: how we achieve the goal. For example, we might set permissions on F, use linux to enforce those permissions, and require 2FA.

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, but together they define what we are achieving. The implementation, on the other hand, can definitely be wrong—if it does not guarantee the goal given the threat model, due to bugs or oversights or supply chain vulnerabilities or anything else, the implementation has a mistake.

1.2.1 Security is Hard

Security inherently if defined in terms of negative goals, such as "**only** TAs can access grades". It is easy to test if a TA can access grades, but it is much harder to test if there is some sequence of interactions that allow *anyone else* to access the grades. For example, someone besides a TA might be able to access the grades through:

- Bug in server software
- Break into a TA's office
- Access an admin account

- Trick the TA
- Break the server's cryptography
- Get a job at the registrar and make yourself a TA

And the list never ends. Because of this, **security is never perfect**. There will almost 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.

1.2.2 Building a threat model

Building 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.

1.2.3 Techniques

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

1.3 Examples

1.3.1 Bad Goals

Business-class airfare

An airline tried to add value to their business-class tickets by allowing them to be changed at any time with no fee. One customer realized that they could board the flight *then* change their ticket.

In this case, the airline's goal did not meet their real needs—perhaps they needed to add something like "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 user's security answers". As it turned out, all of Palin's answers could be looked up on her wikipedia page. Yahoo's implementation may have been perfect, but their goal did not provide any meaningful security.

Instruction Set Architecture Spec

When defining ISAs for processors, designers did not think that timing was important, and processors could be implemented to have instructions to take any number of cycles. This had big benefits for performance and for compatability, but as we'll talk about later in the semester, this variability has been exploited recently to perform sophisticated attacks on wide ranges of processors. The processor implmenetation matched the spec, but the spec itself allowed for this attack.

Fairfax VA School

This school had a system similar to Canvas with a somewhat complex 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.

Matt Honan's Gmail Account

A journalist for wired named Matt Honan had a Gmail account. Gmail's reset password feature avoided security questions, and instead used a backup email account. Honan used an Apple email for this.

Apple's reset password feature then required the user's address and the last four digits of the credit card number. The attacker was able to find his address publically, but could not easily find his credit card digits.

Amazon, which knew his credit card number, required a full CC# 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 credit card number to Amazon's account. After the attacker saved their own credit card number to Honan's Amazon account, they 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.

This complex chain can be very hard to reason about, but these interactions ultimately define security.

1.3.2 Problematic Assumptions

Assuming specific strategy: CAPTCHA

CAPTCHAs were designed to be expensive to automate, but easy for a human to read. Indeed it would be expensive to build an OCR system for these, but attackers did not do this. Instead, they set up farms in countries where the cost of labor is cheap and paid people to solve CAPTCHAs. The result is that it costs some fraction of a cent to solve a CAPTCHA. This is still useful, but not nearly as useful as the original intent.

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, this was considered secure, but today it is easily crackable with a modern computer. As a result, MIT had to switch from DES to Kerberos for authentication.

Dependencies: 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.

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.

1.3.3 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. 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.

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.

1.4 Improving

1.4.1 Goals and Threat Models

- Creating simpler, more general goals
- Avoiding assumptions (such as "no one else will be able to get a user's SIM card") by redesigning the system
- Learn and iterate
- 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.

1.4.2 Implementation

- A simpler system will lead to fewer problems
- Factor out security-critical part (for example, hardware security keys).
- Reuse well-designed code, such as well-tested crypto libraries
- Understand the corner cases

Chapter 2

Authenticating People

2.1 Authentication

In this class, we will talk a lot about requests going to a computer system. And a lot of security comes down to looking at that request and deciding how to handle it. For this, it is crucial to know *who* issued the request. Then, we can decide whether the class should be allowed.

1. Authenticate: find the principal
2. Authorize: decide if the principal is authorized to make the request.

2.2 Passwords

Passwords are the most widespread way of authenticating someone. To make a request, a user will include their username and password with the request and pass it along to the computer system.

Some passwords:

- `password`
- `PaSsW0rd1!`
- `purple-student-hat`

Which of these are "good" and which are "bad"? Many sites would let you have `PaSsW0rd1!` but would not allow `purple-student-hat`. However, what really matters is the way that the adversary will guess passwords. And most likely, they will guess the most common passwords first.

Ideally, we would want all passwords to be equally as likely. But that is not the case. People have to remember their passwords, and it turns out that many people are likely to choose the same password

So what we really want is for our password to have high entropy. How do we get there?

- Require longer password? If someone tries to use `abc123` as a password but it's not long enough, they might use `abc123456`—but this doesn't really add any real entropy. There are predictable ways to lengthen passwords.
- No predictable english words?
- Generate password for the user? Yes.
- Force password changes? Makes it harder to remember their password, may actually decrease entropy. Forcing a password change may be more effective if the passowrd is definitely leaked.

But passwords are still likely to have low entropy. What can we do about this?

2.2.1 Dealing with Low Entropy

A "good" password might have 20 bits of entropy—if an adversary is able to make 2^{20} passwords, they can expect to guess the password. And with current technology, guessing 2^20 times is easy. So, we must limit guesses.

Strawman solution: allow 10 wrong guesses for each account, after which the user is logged in completely.

- This likely does well to prevent any one account from being compromised.
- However, it does not protect the account base overall—an adversary can guess the 10 most common passwords for each account and expect to get into many accounts that use those common passwords.
- This also has a big functionality downside—users are likely to be locked out of their accounts (DoS).

So we also need to limit guessing across accounts. This normally takes the form of limiting entries per IP, or using something like a CAPTCHA, and so on.

2.2.2 Storing Passwords

The most obvious way to store passwords on a server would be to just store each username along with their password.

user	password
alice	`abc123`
bob	`1234`

Risk: adversary steals the password table (by breaking in to the system, stealing the hard drive, etc). Now, every user's account is compromised. However, since many users use the same password across many sites, those other sites are now compromised too.

So, it would be great if we could check whether a user's inputted password is correct without storing the actual password. To do this, we can use a *hash function.* For our current purposes, a hash function is a deterministic way of scrambling the input such that it cannot easily be reversed. That is, the same input will always produce the same output, but given an output there is no easy way to determine the corresponding input.

user	H(pw)
alice	h_a
bob	h_b

Now, if the database gets compromised, only the *hashes* of those passwords are made public. Assuming we choose a *one-way* hash function, meaning that there is no easy way to compute pw given only $H(pw)$, the only way to find pw and log into the user's account then is to figure out the mapping from each possible password to H(pw) and find the one that matches.

To make this more difficult, systems can make use of an expensive H function. These are often called Key Derivation Functions to separate them from hashes that are meant to be fast. Examples are bcrypt and scrypt. In effect, stealing the database removes any rate limiting that is enforced by the server. But we can add cryptographic rate limiting by using a KDF to generate the hash.

However, if all systems use the same hash function (hash functions are hard to make, so they are likely to), then many adversaries could get together and compute a *rainbow table* that maps $H(pw)$ to pw.

2.2.3 Avoiding pre-computation: salting

This precomputation attack only works when everyone is using the same hash function—the problem is that you can precompute some values and get a lot of use out of them—for many users across many systems. If we could have a hash function for each user, pre-computing would be ineffective as the generated table would be only applicable to a single user.

Generating a brand new hash function is difficult, but we can use a *salt* to break this precomputation. Now the table looks like this:

user	salt	H(salt|pw)
alice	r_a	h_a
bob	r_a	h_b

What we have done is generated a number for each user. Then, instead of hashing just the password, we hash the password along with the salt and store that. So now, if an adversary wanted to brute-force the password for a given account, they would have to perform all the work for only that account—this greatly increases the cost of compromising an account.

2.3 Authentication over the Network

We have so far been talking about manually entering a password. But what we normally do is log in to some server on the network—Facebook, Gmail, MIT, etc. And in this scenario, we can get much more creative.

2.3.1 Password Manager

A user can install a password manager on their computer that will generate random passwords for them. Since the user doesn't need to remember these passwords, they can have very high entropy—they are essentially *keys*. Once the user logs into their computer, they can then access their randomly generated passwords and use them to log in to their websites.

server	pw/key
`amazon.com`	`3xyt42...`

But we are still sending this password over the network. If an adversary can watch our network, they can see our password. Things like TLS protect against this, but what if we could authenticate without ever sending the password over the network?

2.3.2 Challenge-Response

Assume that our computer has some key (password with high entropy) k, and the server also knows k.

1. The server chooses a random r (we will call this a *nonce*)
2. The computer computes $F(k, r)$, where F cannot be computed without knowing k, F here is a Message Authentication Code, which we will talk more specifically about. However, it can be approximated as $MAC(k, r) \approx H(k||r)$.
3. The computer sends this value to the server, where it can compute the same value to check whether the computer knows k without the computer actually transmitting k.

Attack: man-in-the-middle—an adversary could sit between the server and Alice, get the nonce from the server, ask alice to log in, and then intercept alice's login request and change the request contents to perform some other function. To help with this, we can include the request itself in the MAC. Now, if the man in the middle changes the request, the MAC will become invalid.

2.3.3 Two-Factor Authentication

Passwords are unlikely to ever be perfect—whether by leaving it on a sticky not on your screen, an adversary stealing the database, or whatever else, a password can be leaked. A common solution is to combine a password with another method of authentication—importantly, one with a different failure mode. Common "classes" are:

- Something you know: password, PIN, etc
- Something you have: USB key, phone, etc
- Something you are: biometrics (fingerprint, face ID)...

Time-based One Time Passwords (TOTP)

In this scenario, the server requests a code along with the password. The user has a device (like a phone) that shares some key k with the server, and they both agree on a protocol by which to generate this code—something like $MAC(k, \text{time / 30 seconds})$. The phone can generate the code, display it to the user, and the server can then verify the code by recomputing it.

Attack: ask the user to give you the code—pretend to be tech support/customer service, etc. This is essentially a phishing attack. The code is then good for 30 seconds, so the attacker can then just enter the code into the website on their end. Similar attacks would include setting up a fake website that looks like the real one, etc.

Avoiding phishing: U2F (simplified)

We can add a bit of complexity to the protocol to avoid this attack. If we include the name of the server that the user is trying to log in to in the request that we sent to the device, the code won't be able to be used on any other site—now, the code might look something like MAC(k, r||name). If the attacker sets up `amason.com` and gets the user to visit it, the U2F device will only generate a code that is good for that site and not the real `amazon.com`.

2.3.4 Biometrics

Biometrics are things like fingerprints, your face, etc. They are very convenient, since you won't forget them, but they are not great for authenticating over the network because they are not replaceable and because they are not particularly secret. An adversary knowing what your fingerprint looks like should not allow them to log in to your account. Biometrics are much more useful if you have a trusted input path that can guarantee that a real human who owns that biometric is on the other end.

Chapter 3

Collision Resistance and File Authentication

In the last chapter, we focused on authenticating people—finding some verification that a person (or a request on behalf of that person) is likely who they claim to be. In this chapter, we will focus on authenticating files, code, and other data, and we will make use of a new tool called a *collision-resistant hash function*, or CRHF, to do so.

3.1 Authenticating a file?

When we say that we want to authenticate a file, we mean that we want to verify that the file's contents are authentic—that is, that they are exactly as they were when we or someone we trust last viewed them.

3.2 CRHF Intuition

Definition 1 (Hash Function) *A hash function H maps a bitstring of any length onto a fixed size space of outputs.* $H : \{0,1\}^* \to \{0,1\}^\lambda$.

In order for a hash function to be *collision-resistant*, we want it to be the case that for any input, the generated output should be "unique". Of course, it cannot really be unique—we are mapping infinitely many inputs onto finitely many outputs—but we want it to be *computationally infeasible* to find a case where they are not unique (a *collision*.

Security Goal: It is "computationally infeasible" to find two distinct inputs to a collision resistant hash function that hash to the same value.

Importantly, CRHFs let you authenticate a long message (f) by authenticating a short string $H(f)$.

3.2.1 Applications

Secure Mirroring

Last time, we saw the Xcode example of mirroring—some large file was difficult to download from far away, so someone set up a mirror in another location and claimed to host the same copy of Xcode, making it easier for people to download. However, we saw that this mirror was able to change the file they were hosting however they like—there was no verification that the downloaded file was an authentic Xcode binary.

To help avoid this attack, we can add some authentication on the file that the mirror hosts. The real vendor can host the ouput of $H(\text{Xcode})$ on their server, and people from across the world can download this *digest*. Then, no matter where they download the large file from, they can recompute the digest $H(\text{downloaded file})$. If H is a CRHF, then if the output matches the vendor digest, the files are almost certainly identical.

Subresource Integrity

If a program fetches a file from some content delivery network, it can store the hash of that file locally and use it to verify that the contents of the file did not change since the application was developed.

Outsourced File Storage

If you want to store your files on a cloud provider, you want to be sure that the cloud provider does not maliciously modify the files without you noticing. To make sure of this, you can store H(files) locally (which takes very little storage space). Then, when you redownload your files locally, you can recompute the hash to verify that they were not tampered with.

3.3 Defining a CRHF (formally)

Note: This is likely the first formal cryptographic definition for many of you, so please ask questions.

An adversary's goal in breaking a Collision Resistant Hash Function is to find a collision—a pair of values $m_0 \neq m_1$ such that $H(m_0) = H(m_1)$.

Definition 2 (Collision Resistance) *A function $H : \{0,1\}^* \to \{0,1\}^\lambda)$ is collision-resistant if for all "efficient" adversaries A, there is some "negligible" function $\mu(\lambda)$ such that*

$$\Pr[H(m_0) = H(m_1), m_0 \neq m_1 : (m_0, m_1) \leftarrow A()] \leq \mu\lambda$$

In words, this means that the probability of finding a collision is so small that no efficient adversary could hope to do it.

There are two ways of thinking about the terms we use in this definition—values in practice and values in theory.

- In Theory
  - λ is the *security parameter*
  - An "efficient" algorithm is a randomized algorithm that runs in time polynomial in λ.
  - "Negligible" is a function that grows slower than the inverse of every polynomial—a function that is $O(\frac{1}{\lambda^c})\, \forall c \in \mathbb{N}$.
- In Practice
  - $\lambda = 128, 256$, or 512

- "efficient" = runs in time $\leq 2^{128}$.
- "negligible" = probability $\leq 2^{-128}$.

3.3.1 Where do these numbers come from?

Table 3.1: Big numbers in terms of hashes

2^{30}	operations/second on a laptop
2^{58}	ops/sec on Fugaku supercomputer
2^{68}	hashes/second on the Bitcoin network (as of Fall 2022)
2^{92}	hashes/yr on the Bitcoin network
2^{114}	hashes required to use enough energy to boil the ocean
2^{140}	hashes required to use one year of the sun's energy

Table 3.2: Exponents as probabilities

2^{-1}	fair coin lands heads
2^{-28}	probability of winning Mega Millions
2^{-128}	will never happen

The takeaway is that if our hash function has a 2^{-128} chance of a collision. We can be extremely sure that a collision will never occur.

3.4 Constructing a CRHF

Pretty much every CRHF that exists today follows the same two-step approach:

1. Build a small CRHF $H_{small} : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$. This step is "more art than science", and involves doing things that seem collision resistant and testing the result extensively to see if it seems breakable. The current standard for fast collision-resistant hashing is SHA256, which was designed by the NSA in 2001. Hash Functions can also be build from "nice" assumptions about hardness of problems such as factoring of large numbers, but hash functions based on these assumptions tend to be very slow so are almost never used in practice.
2. Use H_{small} to construct $H : \{0,1\}^* \to \{0,1\}^{\lambda}$. This can be done very cleanly, and one approach is the "Merkle-Damgard" approach below.

3.4.1 Merkle-Damgard

Merkle-Damgard is based on splitting the message into λ-sized blocks $[m_1, \ldots, m_n]$ and successively hashing them together.

First, start with a bitstring 0^λ. and compute $h_1 = H_{\text{small}}(0^\lambda||m_1)$. Next, compute $h^2 = H_{small}(h_1||m_2)$, h_3, and so on through h_n. Finally, compute $h_{\text{final}} = H_{small}(h_n, n)$ and output h_{final}.

We won't prove this here, but we can use the fact that H_{small} is collision-resistant to prove that H must also be collision-resistant. The basic idea of the proof is to show that given a collision in H, we can easily compute a collision in H_{small}.

3.4.2 The Birthday Paradox

An important thing to note when dealing wiht hash functions is the birthday paradox, which says that given a hash function with λ-bit output, you can alays find a collision in time $O(\sqrt{2^\lambda}) = O(2^{\frac{\lambda}{2}})$. This means that a collision can be found in $2^{\frac{\lambda}{2}}$—if you want it to take $2^1 28$ to find a collision, you need 256-bit output from your hash function.

3.4.3 Domain Separation

Consider the case where we have a one-input CRHF (such as SHA2) $H : \{0,1\}^* \to \{0,1\}^{256}$ and want to construct a two-input CRHF $H_2(x, y)$.

Bad Idea: An obvious solution may be to just concatenate the two values, so that $H_2(x, y) = H(x||y)$. However, this allows two different pairs of messages to hash to the same value—H_2("key", "value") = H_2("key", "value"). Both Amazon and Flickr had a bug arising from this—they concatenated all parameters before hashing, and had parameters such that two different intents had the same concatenation.

3.4.4 Length-Extension

Recall the concept of Message Authentication Codes (MAC) from the last lecture—a code that can be sent along with a message to verify that the message was not changed.

Last time, we used $MAC(k, m) = H(k||m)$ as a very simple construction of a MAC. However, this construction has an easy attack—given MAC(k, m), it is easy to compute MAC$(k, m||m')$ without knowing k if using a hash function built with the Merkle-Damgard construction. To do this, you can hash the output of MAC(k, m) with two more blocks—a new message m'' and another length block. Now, we have computed MAC$(k, m||m')$ where m' is the original length block plus some custom message without knowing k.

This lesson here is that we were using a hash function that was only guaranteed to be collision-resistant, and assumed it had other properties (such as that it is guaranteed to be difficult to compute the hash of an extension of the original message).

3.5 Applications

3.5.1 Merkle Trees

In many cases, we would like to authenticate many files quickly. If we have N files to authenticate, one option would be to store the hash of all N files on the server. However, in this case the client needs to download every file in order to authenticate a single file. Another option would be to store the hashes for each file on the server, but then authenticating many files becomes difficult. A better option is to build a tree out of the files.

I think that the diagram from lecture is pretty important here. But the basic idea is that we build a tree by hashing together pairs of files, then hash each pair of hashes and so on until we eventually end up with a single root hash. Then, in order to verify any single file we need to download $\log N$ hashes, and to verify all files we only need to download a single root hash.