Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 11, Passwords and Authorization

§11.1 Passwords (hashing, cracking, password managers, PAKEs and OPAQUE, hardening); §11.2 Authentication devices (SMS, TOTP, FIDO/WebAuthn and passkeys) (book pages 209–229)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lecture 25, Mon Dec 7: passwords and authentication

`https://practicalcrypto.org`

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-11-15.

Chapter 11

Passwords and Authorization

The need to verify the identity of users is one of the most basic problems in computer security. Authentication (or authorization) is the process of determining which human beings and remote programs are allowed to interact with different parts of a system. While machines can often solve these problems using cryptographic protocols, the machines themselves are typically controlled by humans. Thus to ensure that a given system is actually secure, we must authenticate the users who control it.

When it comes to authenticating human users, there are typically three different types of mechanism, or *authentication factors*, that we employ. Computer security folks typically summarize these factors as follows:

1. **Something you know** refers to knowledge factors such as passwords, PINs, secret gestures and other authorization secrets that a person can remember. These are often the weakest authentication factor, although that isn't necessarily a rule..

2. **Something you have** refers to physical items that you possess. An obvious real-world example is a physical key that fits into a lock. In computer security this usually refers to items such as smartcards, specialized authentication devices such as USB dongles, and smartphones that receive SMS codes or (much better) run specialized authenticator apps.

3. **Something you are** refers to items that are part of your physical person, including biometrics such as your face, fingerprints or iris pattern. These elements are convenient because you physically possess them, and because they're often difficult to remove. However, these same facts make them inconvenient, should sensitive data be lost in a breach. You've probably changed your password, but have you ever tried changing your fingerprints?

In the field of computer security it's considered self-evident that an authentication system based on two or more of these factors is always better than a system that just uses one, such

DRAFT

as password-only authentication. This is why many banking websites require you to enter an authenticator code as well as your password, and increasingly this practice is filtering down to many other types of system. Systems that require more than one factor are said to use *multi-factor authentication* (MFA).[1]

So what does this have to do with cryptography? In one sense, the answer to this question is fundamental: there is no point in cryptography if we can't trust our endpoints. But cryptography also shows up in many specific aspects of the authentication process. It is used in everything from password storage to one-time authenticator code generation, and it appears in many of the systems used for verifying biometrics. Human authentication can also be directly tied to cryptographic protocols, for example, to enable encryption that is secured by passwords or other authentication factors.

In the next several sections we'll discuss several examples of each of these ideas.

11.1 Passwords

Passwords are almost certainly the oldest authentication system. The use of the password a means to authenticate oneself to others dates back to antiquity, and perhaps to the dawn of human language. The Book of Judges describes a password, or *shibboleth*. Members of the Roman army used "watchwords" to identify themselves to sentries. And of course, Ali Baba would never have entered his treasure cave without the right password.

The first recorded use of a computer password was by Fernando Corbató in 1960, as a way for users to log into MIT's time sharing system [127]. Today we are all forced to remember multiple passwords, some to access our phones, others to take money out of the bank, and dozens more for each of our many accounts. We have almost certainly forgotten more passwords than we currently use. No matter how hard security professionals try to come up with something better, even the most secure systems are probably dependent on human-memorized passwords at some level or another.

The problem with passwords. To abuse a Winston Churchill quote: passwords are the worst form of authentication, except for possibly all the others. In their defense, passwords have two critical features that many authentication systems lack. First, they can be memorized by human beings, or at least written down conveniently without the aid of a computer. Second, passwords can be easily changed. If you forget a password or share it with too many people, it's often relatively easy to replace the password with a new one. This is not something you can easily do with your face or fingerprints!

The downsides of passwords are also well-known. The sorts of passwords that humans come up with on their own are often extremely easy to guess. In study after study, hu-

[1] Sometimes the term two-factor authenticaiton (2FA) is used instead.

DRAFT

mans have been found to gravitate towards either weak (guessable) passwords, or else they *re-use* the same password across many services [233, 251, 318]. When system designers force users to select more complex passwords, those users will often cheat in the laziest way possible (*e.g.,* adding numbers and symbols to otherwise guessable passwords) or they will simply forget their password, which causes security to devolve to that of the password recovery mechanism they'll use instead. System designers often aggravate these problems in the name of "security": many organizations force routine password changes due to an obsolete understanding of security practices, even though this practice is now discouraged by standards organizations like NIST [143].

Passwords in cryptography. Cryptography allows us to extend trust from one secure device to other parts of the network. As discussed above, passwords are fundamental to cryptography *even if no cryptographic protocol uses them*, simply because they secure the systems where we anchor our trust. But passwords are also used routinely in cryptographic protocols, and cryptography is used to protect passwords. Some examples include password hashing and password-authenticated key agreement (PAKE) protocols, which we'll discuss below.

DRAFT

11.1.1 Password hashing

Back in the 1970s, Robert Morris developed a technique for protecting Unix password files from insider theft. Instead of storing plaintext passwords in system password files, Morris proposed to process each stored password using a *one-way function* such as a hash function [233].[2] The resulting hash $h \leftarrow H(\texttt{password})$ was stored in the password file instead. When a user returned to log into the system, they would enter their password and it would be hashed a second time: this result could be compared against the stored hash value. An attacker who obtained only the hash values, at least in principle, could not easily invert them to recover the initial passwords.

Of course, Morris and his colleague Ken Thompson quickly discovered that simple password hashing was extremely vulnerable to *dictionary attacks* [233]. Here an attacker would assemble a "dictionary" of candidate passwords and simply hash each candidate forward to see if it matched a target digest h from the password file. Even worse, since the only input to these early password hashing functions was the password itself, all of the hashing could be done in advance: attackers could hash enormous password dictionaries at their leisure, and then simply compare the pre-computed results against any later-stolen password file. To block these attacks, first Morris and Thompson, and then later researchers, proposed several countermeasures:

[2] The initial function Morris proposed used the password as a key to encrypt a fixed plaintext with the M-209 rotor cipher. This is certainly not secure. Later versions used DES, which is not much better. The principle is still employed today, just with much better cryptographic functions.

1. **Salt.** One way to block pre-computation attacks is to hash each specific user's password with a *completely different hash function*, so that the attacker will not be able to pre-compute a dictionary until after they compromise the password file and learn which function was used. Since developing millions of different hash functions is expensive and risky, a more practical approach is to use a *keyed* hash function, where the random key is called *salt*. (In practice, many constructions simply concatenate the salt and password and hash them together as $H(\mathsf{salt}\|\mathsf{password})$.) Salt is typically chosen when a user sets up their password, and is normally stored in the password file/database along with the hashed password. Naturally, salt must be long and random enough that the attacker cannot predict it before they steal the password database, and it should be generated uniquely for each password that is stored. Note that salt is absolutely essential in modern password hashing: a failure to use proper salt should be viewed as an urgent security vulnerability.

2. **Time-intensive functions.** Morris and Thompson also proposed a second countermeasure to slow down dictionary attacks: they suggested that the hash function could be made more time-consuming to compute. Of course, a slower hash function means a proportionally longer time for a legitimate user to log in, but this might be a good tradeoff: user login only requires one hash, while a dictionary attack could require millions. One example of a deliberately time-consuming hash function is the PBKDF2 password hashing function, which can be configured to call some underlying hash function like SHA2 a user-configurable number of times (or *iterations*) in order to slow down dictionary attacks.

   The problem with computationally-intensive functions is that password cracking is embarrassingly parallel, and often friendly to specialized hardware: modern cracking rigs use GPUs, ASICs or FPGAs enable the simultaneous cracking of many passwords. A password-hashing "difficulty" sufficient to stop a parallelized hardware attack could actually prove too slow for puposes of normal user login, since that implementation is likely written in software. For these and other reasons, purely time-intensive functions should be considered insufficient for new deployed systems.[3]

3. **Memory-intensive functions.** The next advance in combating hardware cracking was the development of *memory-intensive* password hash functions. These functions take advantage of the fact that many algorithms are relatively efficient if one has enough memory, but are vastly slower otherwise. Since RAM is relatively expensive compared to ASIC/FPGA gates, this can increase the cost of parallel hardware cracking. Some examples of these functions include `scrypt` and `Argon2` [263, **?**]

   These functions typically work in three stages: (1) a first (and possibly time-consuming)

[3] Thanks to the relatively slow development of FIPS standards, PBKDF2 may still be required for FIPS compliance. This typically does not preclude using a better memory-intensive function: you may have to use both. Always talk to an expert before you take my advice on FIPS compliance.

DRAFT

stage builds a large table of computed values in a manner that does not facilitate the easy generation of individual values, (2) a second stage reads reads selected values from various locations in the table, such that the locations read are not easily predictable in advance, and (3) the results are folded together into a final hash. A computer with minimal RAM can still in principle compute these functions, but would need to repeatedly re-compute portions of the table at enormous cost. The goal of these functions is to force hardware attackers to purchase huge amounts of relatively expensive RAM in order to perform rapid dictionary attacks.

While this approach is recommended by many security experts, it is not perfect. First, the amount of RAM specified in many real deployment parameters is somewhat modest, usually because large RAM requirements can also limit the rate of normal user logins. For example, OWASP currently recommends using `argon2id` with 19 MiB of RAM [259], which is a reasonable amount but may not stop well-resourced attackers.[4] Even if memory-intensive hashing is not sufficient to stop all attacks, it is recommended best practice to use such a function.

4. **Pepper (aka "secret salt").** A final approach to securing systems is to use a true cryptographic secret *in addition* to the salt used for normal password hashing. This secret is sometimes called *pepper*. Pepper is often just a single key that is used for all accounts, and it is stored within a portion of the system that the attacker is less likely to compromise, such as a Hardware Security Module. The goal of pepper is to add defense in depth: even if the attacker steals a password database, they may not be able to extract the pepper. Without knowing that value, they cannot perform offline dictionary attacks.

   Of course, an attacker with persistent access to the system may be able to use the ordinary login system's access to the pepper in order to conduct an *online* dictionary attack: for example, repeatedly querying an HSM to obtain password hashes. However, the rate at which they can do this should be vastly slower than an offline guessing attack, and implementers may be able to further throttle or limit this attack. As a final note, pepper does not replace salt. It should be exclusively used to *augment* a well-designed password hashing system: if the pepper is stolen, passwords should still benefit from all of the other protections discussed above.

5. **Secure lockboxes.** Some devices and cloud storage systems combine the previous techniques to construct a "secure lockbox."[5] A secure lockbox uses specialized hardware or software to enforce *guessing limits* on user passwords. It contains an account-

[4] Note: please check current recommendations at the time you're reading this! Requirements can change quickly.

[5] The term was originally introduced by Apple in its Platform Security Guide [92], but I think it's broad enough to be used more generally. Both Apple iCloud and Google Android Backup, as well as phones using both Apple and Google operating systems contain instances of it.

DRAFT

specific cryptographic secret S that is combined with the entered passcode using a strong key derivation function, as well as a counter that is initialized to 0. Each time the user enters an incorrect passcode (a condition that can be checked by testing the resulting derived hash/key), the counter is incremented. When the user enters a correct password, the counter is set back to 0. When the counter reaches some pre-set maximum value max, the secret S is zeroized and the user's account is locked. This design depends on the correct operation of the hardware or software, since an attacker who can disable the counter can bypass most of its security. One must also be careful that outsiders cannot accidentally disable a user's account.

Password-based encryption. Most of the discussion above can be applied to the problem of deriving cryptographic keys from passwords. This is because "hash functions" such as scrypt, Argon 2 and PBKDF2 are also key derivation functions. That is, their output can be used directly as a cryptographic key. Whether it is advisable to use passwords depends on the application, as well as the other protections around any system that uses them.

Password-based cryptography is used in a variety of applications, including disk encryption, smartphone storage encryption, protection of SSH private keys, authenticating Wi-Fi WPA2 connections, and PAKE schemes (see §11.1.4 below.) The security of any given system depends on the strength of the passwords being used, as well as other protections such as the use of memory-hard KDFs and hardware-enforced guessing limits.

Case study: smartphone passcodes. Most smartphones default to relatively weak *passcodes*, which are typically six decimal digits in length, and chosen by the user. These passcodes double as both the code to unlock the device, as well as the basis for deriving the encryption keys that are used to encrypt data in the device's storage [20, 17]. This has led to conflict between smartphone manufacturers and law enforcement, the most famous case being a lawsuit between Apple and the FBI in 2016 [92]. Since most phone passcodes are obviously not strong, an obvious question is: *why do they cause so much difficulty for law enforcement?*

The reason that short passcodes provide strong security is that most smartphones contain an instance of the *secure lockbox* pattern described above. On Apple devices, for example, each device contains a specialized co-processor called the Secure Enclave Processor (SEP), that performs security-sensitive processes such as encryption and key management. When a user enters a passcode, it is sent to the SEP, where it is hashed together with a phone-specific secret key using PBKDF2 [20]. The silicon design of the SEP makes it fundamentally challenging to extract this key, either physically or via software attacks. The SEP also enforces a 10-attempt counter before erasing the key. As a backup to the counter logic, the number of PBKDF2 iterations is configured so that each hashing attempt takes approximately 80 milliseconds on this processor. Thus even if an attacker obtains total control over the rest of the phone and somehow disables the 10 attempt limit within the SEP, they should be limited to testing at most 45,000 ($\approx 2^{16}$) passwords per hour if they cannot extract the key and use alternative hardware. Table 11.1 shows cracking times for various sorts of random passwords at this rate.

DRAFT

Type of (random) password	Worst-case cracking time	Average case
4 decimal digits	13.3 minutes	6.65 minutes
6 decimal digits	22.2 hours	11.1 hours
6 uppercase letters	9.36 months	4.68 months
3 words from BIP39 dictionary [260]	21.79 years	11.9 years
10 decimal digits	25.36 years	12.68 years

Table 11.1: **Apple SEP cracking times.** Some example worst/average case cracking times for different types of *random* passwords on an iPhone SEP. Here we assume an 80ms hardware-enforced hashing time, and that the 10-attempt guessing limit has been disabled. Note that we assume each password is drawn randomly from the set of passwords of that type: user-chosen passwords will typically be much weaker.

Many Google Android devices perform a similar function using the TrustZone virtualization service: a specialized Trusted Application called "Gatekeeper" uses `scrypt` to hash keys into passwords, enforcing attempt limits in a similar manner [50, 17]. See §15.2 for more on the TrustZone architecture.

High-entropy passwords. The simplest and best defense against password cracking is for users to simply generate *high-entropy* passwords. This is extremely challenging for ordinary human beings to do in their heads, but it's relatively easy to do using mechanical or computer assistance. The challenge, of course, is that it takes some effort for people to remember these passwords.

Generating wordlist passwords. A common approach to generating high-entropy human memorable passwords is to begin with a wordlist consisting of several thousand words from a natural human language. Each password (or *passphrase*) is simply a sequence of N concatenated words, where each word is randomly sampled from the list (using a strong random number generator), with repeated words explicitly allowed: *e.g.,* `latin rural blur`. If a digital random number generator isn't available, proposals like Diceware can help you sample passwords offline using physical randomness [239].

```
1049    loan
1050    lobster
1051    local
1052    lock
1053    logic
1054    lonely
1055    long
1056    loop
1057    lottery
1058    loud
```

Figure 11.1: A small fragment of the BIP-39 English wordlist [260].

DRAFT

Many wordlists are available. As just one popular example, Bitcoin's BIP-39 standard [260] specifies an English wordlist of size 2^{11}. To generate a word from this list, simply pick an 11-bit random unsigned integer using a secure RNG, and select the word at that index. Repeat this N times to obtain a password with an entropy of $11 \cdot N$ bits. Concretely: a password with at least 128 bits of security requires 12 words, and 256 bits requires 24 words. In principle, larger wordlists can slightly reduce the length of passphrase needed for an equivalent security level. 1Password currently uses a list containing approximately 18,000 (about $2^{17.4}$) words, which gives 128-bit security for passwords comprising only eight words. Even a six-word password based on this list provides more than 100 bits of security, which is sufficient to prevent most practical cracking attacks. At least one study has shown that users can remember 6 or 12-word passphrases with high accuracy, provided that they memorize them using spaced repetition techniques [76].

11.1.2 Password cracking

As mentioned previously, the main threat to password hashing and password-based encryption is *offline password cracking.* This can be conducted using CPUs, but is more typically performed using specialized hardware such as GPUs, ASICs or FPGAs. A number of open source tools have been developed to assist in cracking passwords across many different applications. The most popular are Hashcat [157] and John the Ripper [258]. Both support CPU and GPU cracking, with varying levels of support for different hashing algorithms and applications.

The input to these tools is usually a set of target hashes along with any necessary salt values. In the case of password-based encryption schemes, sometimes it is necessary to provide a portion of ciphertext that contains known (testable) plaintext. In this case, the cracking algorithm first hashes to derive the appropriate key, attempts to decrypt the ciphertext, and verifies correct decryption. Since cracking is embarrassingly parallel, these tools will typically crack many passwords at the same time, either using multiple cores on a CPU or by employing the SIMD instructions of a GPU.

Dictionaries. A critical input to the cracking process is a good *dictionary* of candidate passwords. For unstructured passwords, such as strings of random numbers or letters, dictionaries can be generated programatically. For human-selected passwords, it is common to use wordlists containing likely user passwords. These dictionaries are often compiled from actual dictionaries, and combined with popular passwords found in past data breaches, such as those in Table 11.2. Often, dictionaries are "mutated" by combining words from multiple worldlists, and adding variable elements such as numbers, capitalization and punctuation, so that `kitten` becomes `Kitten123!` and so on. Most cracking tools also accept custom dictionaries, for cases where additional information is available about the target.

DRAFT

Password	Number of occurrences	Cracking time
`123456`	3,018,350	$<$ 1 second
`123456789`	1,625,135	$<$ 1 second
`12345678`	884,740	$<$ 1 second
`password`	692,151	$<$ 1 second
`qwerty123`	583,630	$<$ 1 second

Table 11.2: **Top five most common passwords.** From a list compiled by Nordpass from breach data in April 2025 [253], results are worldwide.

Pre-computation

There are two flavors of password cracking. For passwords with appropriate salt, cracking can only begin once the hashed password and salt have been obtained. This follows the paradigm discussed above, where the password cracker begins testing candidates, hashing them using the salt, and then testing against the target value. Since every target should use different salt, cracking attempts must be aimed at one target and no pre-computation is possible.

Although it is increasingly rare, some password-based derivation schemes use no (or insufficient) salt. In this case, the cracker's task becomes much more efficient. Here, the cracker can hash each dictionary entry just one time, and then compare it against many different target password hashes in one efficient lookup. In an extreme case, an attacker can *pre-compute* a database of passwords and their hashes even before any target hashes have been obtained: in this case, the process of cracking a target hash can be as simple as looking up a target hash in this pre-computed database. Some past examples of vulnerable schemes include Windows NTLM [**?**], as well as WPA2 password hashing (§9.2.5.)

DRAFT

Time-memory tradeoffs. While database lookups are clearly much faster than repeated testing, a problem with "just build a database of all candidate hashes" is that the resulting databases can be inconveniently large. For example, a dictionary containing 2^{40} values might be possible to build in a reasonable time using hardware, but at a cost of 16 bytes per entry, would require a database of size approximately 16 TB. This is difficult to give out as a party favor.

Back in 1980, Hellman proposed a compromise solution called a *time-memory tradeoff* [158].[6] In this approach, an attacker pre-computes a much smaller database that greatly speeds up later cracking attempts. Ideally, the time and database size can be fluidly adjusted, or "traded off" against one another to find an optimal solution. While Hellman's

[6]Hellman's original motivation was to find the key used to product a DES ciphertext from some fixed plaintext. This is similar to password hashing if the "key" is the password and the "hash" is the ciphertext.

original proposal is impractical, a more popular data structure for this application is the *rainbow table*, which was invented by Philippe Oechslin in 2003 [255].

Towards rainbow tables: simple hash chains. A full database of password hashes might consist of a series of values of the form $(\texttt{password}, H(\texttt{password}))$ for some huge number of candidate passwords. Our goal is to store only a tiny fraction of those values into a table, and then to use an automated approach to generate all of the intervening values when a new password must be cracked. This can be done by constructing a sequence of hash chains, each of a specific form.

A key requirement of the chaining approach is that we must have a machine-computable function $R(h)$ that takes in a password hash h, and generates a next candidate password. For completely unstructured random passwords, this is relatively easy to build. For more structured passwords, this function will often be more complicated: it might use the input bits to seed a "mutation function" that mixes together dictionary words, numbers and letters in a manner that emulates real human password choices.

To construct our table, the goal is to produce a series of M chains, each of which is computed as follows:

1. Each chain begins with some random hash h_0.
2. For $i = 1$ to N, the pre-computation phase next computes $\mathsf{passwd}_i \leftarrow R(h_{i-1})$ and $h_i \leftarrow H(\mathsf{passwd}_i)$.
3. The pair (h_0, passwd_N), representing the start and ending values for the chain, are written to the file.

Note that while millions of intermediate passwords may be computed as part of the process, only two values need to be written to the file for each chain. This process repeats for many, many different chains. By adjusting N (and implicitly, the number of chains M), this process can cover the entirety, or at least a huge part of the password space with only a small amount of data needing to be stored for each chain.

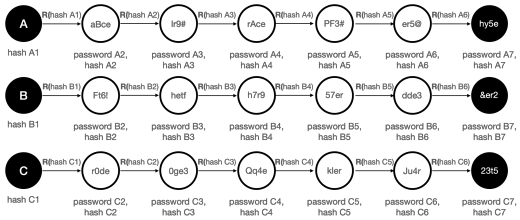

Figure 11.2: Three example hash chains. The values inside of the circles are passwords produced by the function $R()$. The dark circles represent the start and end of each chain, and contain the only values that are actually written to the resulting table. The hollow circles are only derived as part of the computation process.

DRAFT

Let us imagine that the attacker has a target hash h'_0 they wish to crack. We can imagine that this corresponds to a password inside of one of our chains, *e.g.,* the nth password in chain m. Our problem is that we do not know m and n! The goal of the search process is to begin with the hash that we know, and then repeatedly compute the function $R()$, hashing and repeating this process until we reach a value that matches the *ending* value of one of our chains. At this point we have (hopefully) learned which chain m the original hash h'_0 was in. Once we know the identity of the correct chain, we can simply re-compute the full chain from its initial value, until we find a password with a hash that matches h'_0.

So what's the problem? The major problem with this approach is caused by collisions. Ideally, each password in every chain should be completely distinct, which means that the cracking process above will always identify the correct chain if h' is present in one. In practice, if the value $R(h)$ in any chain collides with the value $R(\bar{h})$ in any other chain (or if the hashes themselves collide), the two chains will *merge*. This means that the searching party starting at h' might end up identifying the wrong chain m, and will not identify the password. An example is shown in Figure 11.3.

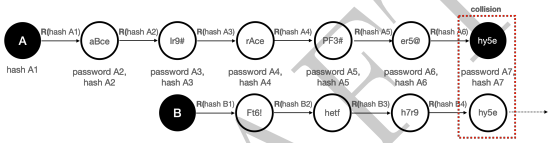

Figure 11.3: An example of two colliding chains. A password collision causes a value in the middle of chain B to be identical to the value `hy5e`, which is also the ending point of chain A. If a cracker begins with a target in chain B, it will eventually encounter this ending value, which will lead it to incorrectly believe that the password is contained within chain A.

DRAFT

Rainbow tables.

11.1.3 Password Managers

The problem of deriving and remembering passwords is difficult for human beings. As with most challenging tasks, we've found it easier to outsource most of the work to computers. One result is the *password manager*. Password managers are applications that generate, store, and (in some cases) synchronize passwords across devices. Most password managers protect a large *vault* full of passwords using a single master password, and occasionally with other authentication mechanisms, such as biometrics. Used wisely, password managers are generally a great idea: they dramatically increase the strength of user passwords, while also protecting users from the danger of re-using passwords across different sites and services.

From a technical perspective, *local* password managers are fairly straightforward. For each new site/app/service that requires a password, the password manager can generate

a strong password using a secure random number generator. The username, site's domain name, and password are stored together in a database. Commonly, the database (or "vault") is then encrypted under a key derived from the master password: ideally, with an aggressively-configured password hashing function and appropriate salt. Some operating systems and browsers contain built-in password managers, and/or provide an API to integrate applications with third-party password managers. In compatible web browsers and apps, the password manager can auto-fill passwords into the appropriate login form when needed.

Cloud-based synchronization. The real security challenge in most password managers is the problem of making the user's password database available on multiple devices. Deployed password managers typically use one of three possible solutions to this problem. Each has its own unique set of tradeoffs:

1. *Simple cloud-based backup.* Many services advertise "cloud-based" password managers. Typically these systems will upload your password-encrypted vault file to the provider. These services will often advertise that the product is end-to-end encrypted, but users should be cautious, since the key is based on your master password. Some providers layer additional checks such as MFA and PAKE schemes (§11.1.4) as a requirement for downloading these vault files. The problem is that these additional mechanisms can be bypassed if the provider is breached, which means that the true "end-to-end" security of these systems ultimately reduces to that of the password-based encryption. For example, in 2022 the popular service LastPass was breached and many encrypted vault files were stolen. Even though the majority of the keys were derived using PBKDF2 set to 600,000 iterations [307], researchers suspect that this breach is linked to a rash of cryptocurrency thefts that occurred over the following years [192].

2. *HSM-based synchronization.* To strengthen cloud-based synchronization, some providers use Hardware Security Modules (HSMs) to generate and store the keys used to encrypt password vaults. These secure devices are designed to be much more resilient to breaches than software-only systems. In particular, they can implement online *guessing limits* that strictly limit the number of password guesses that an attacker can make before the user's account is locked. The disadvantage of this approach is that the security of these systems is heavily based on that of the HSM used. This approach is used by Apple's iCloud Key Vault, which synchronizes passwords across Apple devices on the same account [20]. See §11.1.5 for much more on this approach.

3. *Peer-to-peer synchronization.* A final approach is to establish strong shared keys between individual devices, and then use these keys directly to encrypt data uploaded to a server. These keys can be established by running an authenticated key agreement between the devices, or by scanning a cryptographic key into each device using a QR code. The benefit of this approach is that, since the keys are not based on passwords,

DRAFT

they should be resilient to dictionary attacks. The downside is that you must always keep one functioning device available, or else carefully store a backup code that allows you to recover from the loss of all devices.

11.1.4 Password Authenticated Key Agreement (PAKE)

Even if you use all of the password hashing techniques above, you are still faced with a problem: the user must send the cleartext password to the server to log in. If an attacker is able to gain persistent access to this server, or if the server accidentally logs user passwords,[7] an attacker will obtain any passwords that users send in.

One way to solve this problem is to use a protocol called a Password Authenticated Key Agreement (PAKE) scheme. In the main flavor of PAKE we will describe, a client has a password and the server has an authenticator derived from the *hash* of a password. The goal of the protocol is for both parties to agree on a shared session key. If the client's password matches the server's hash, the key they agree on will be the same. Otherwise it won't be, or the protocol will fail. Most critically, the server (and anyone observing the communication) should learn only one bit of information: whether or not the key exchange succeeded. They should learn no additional information about the real password or any incorrect password guess values the client attempts to use.

PAKE is useful if the end-goal is to establish shared cryptographic keys for communication. But even if you're not interested in encrypting things, it can still be a useful tool simply to determine whether the client knows the right password. In the latter case, the client and server can run the PAKE protocol and then conduct a simple challenge/response protocol to determine whether they agreed on the same key. The key itself can then be discarded or ignored.

Symmetric vs. asymmetric. The version of PAKE we describe above is called *asymmetric* PAKE. This simply means that the server stores the *hash* of a password, and the client knows the plaintext password. In *symmetric PAKE* the client and server must each have the cleartext password. We mostly won't discuss symmetric PAKE here because it's bad practice to keep plaintext passwords on servers. However, one example where it is used is the Dragonfly PAKE used in WPA3, see §9.2.5.

Limitations of PAKE schemes. A common misunderstanding is that PAKE schemes can somehow prevent attackers who compromise the server's password database from running dictionary attacks. Unfortunately this is not the case. In the very worst case, a compromised server can "run a PAKE protocol with itself" by acting as both client and server, and testing passwords one at a time. This is unfortunately just the nature of using passwords: if I have

[7] This is not theoretical. In May 2018, Twitter reported that due to a software flaw, it had accidentally logged some user passwords in internal logs.

DRAFT

the information that allows me to check your password, a dictionary attack is fundamentally hard to avoid.

Obsolete protocols: SRP. One popular and venerable PAKE scheme is called the Secure Remote Password (SRP) protocol [332]. I mention SRP not because you should use it: in fact, you shouldn't. The protocol is something of an antique and, in my opinion, *should not be used for any new deployments.* It is, however, still used in a few places across the Internet.

SRP has many problems. For one thing, the protocol lacks a security proof. For another, it uses obsolete finite-field cryptography and does not support elliptic curves. Most critically, the server must send the user's salt value to *every single remote client who wishes to attempt a login.* This means that any attacker can query a server for the salt used to secure any user's account, and then launch a pre-computation attack to build a dictionary of hashed passwords for each user. Such a dictionary can be useful if the attacker later compromises the password database.

OPAQUE

Fortunately there are secure and modern protocols that do not have the flaws of SRP. One of those protocols is OPAQUE [179], which is currently undergoing an IETF/CFRG standardization process [78]. OPAQUE is an asymmetric PAKE that has several advantages. First, it does not enable pre-computation attacks by handing over the salt. Second, it can be implemented using efficient elliptic curve groups. Third, it can be used with any password hashing algorithm you choose. Most importantly, it has a careful security analysis and was developed by professional cryptographers and engineers. You may not always need a PAKE, but if you do, OPAQUE is the one to implement.[8]

DRAFT

Registration. When a client wishes to register a new password with a server that uses OPAQUE, the server first generates a per-account salt S. The client and server interact via a secure channel, such as TLS, and conduct the following steps:

1. The client executes a privacy-preserving protocol called an *oblivious pseudorandom function* (OPRF). This protocol securely combines the salt S with the client's password, such that neither party learns the other's input to the protocol, and the server does not learn the function's output.[9] The output of this protocol is a cryptographic mixture $K' \leftarrow \mathsf{PRF}(S, \mathsf{password})$ that is only given to the client.

2. Once the client obtains the key K' from the OPRF protocol, it can further apply a memory-hard hash function such as `scrypt` to obtain a new key $K \leftarrow H(K')$. This

[8] And you should do so by following the draft standard [78].

[9] This property ensures that clients never learn the account's salt. It also ensures that the client's password input is totally hidden from the server.

ensures that stored passwords receive the protection of both salt and strong hashing. Even better, since the hashing is done on the client's side, the server does not need to expend its own resources to compute it.

3. To enroll into the OPAQUE system, the client next generates a long-term authentication keypair $pk_{\mathsf{client}}, sk_{\mathsf{client}}$ for a secure key agreement protocol. It securely encrypts the secret key sk_{client} under K to form a ciphertext, for example:

$$C \leftarrow \mathsf{AEAD.Encrypt}(K, sk_{\mathsf{client}})$$

The client sends the values $(pk_{\mathsf{client}}, C)$ to the server. The server now stores the tuple $(\mathsf{user_id}, pk_{\mathsf{client}}, C, S)$ into its password database. The server may run the authentication protocol with the client (below) to ensure that the client has completed the protocol successfully.

Authentication. When the client later wishes to authenticate to the server, it first obtains the server's public authentication key pk_{server}. It can now perform the following steps:

1. The client identifies its account to the server, and executes the OPRF using the client's password guess $\mathsf{password}'$ and the server's salt S. The result is the value $K' \leftarrow \mathsf{PRF}(S, \mathsf{password}')$.
2. The client hashes K' to obtain $K \leftarrow H(K')$.
3. The server sends C to the client, which decrypts it using K to obtain sk_{client}.
4. The client and server now perform a *mutually authenticated* key agreement protocol (§7.3.7). Here the client uses the server's public key pk_{server} as its input, and the server uses the client's public key pk_{client}. This protocol will compute a shared key MS if and only if the two parties possess the corresponding secret keys.
5. The client and server confirm that they each received the same shared key MS.

Notice that the client can still run the protocol many times to test candidate passwords. However, each execution of the protocol should give an attacker at most one guessing attempt: this means a protocol can slow down online guessing attacks by *e.g.,* adding delays or setting a maximum attempt counter. The client should also remember to zeroize any values such as sk_{client} when it has completed the protocol.

The OPRF. The OPAQUE protocol is modular, meaning that the components above can be realized using different ingredients. However, the draft IETF standard proposes a concrete instantiation based on a specific elliptic curve OPRF protocol described in RFC 9497 [97] and the *Triple DH* key agreement protocol [212]. By combining messages together, the result is a three-move protocol, the exact details of which are specified in [78].

DRAFT

The main interesting part of this protocol is the oblivious PRF, so we will briefly describe it here. Let $\mathbb{G}$ be a prime-order group, which we will describe using additive notation. The goal of the PRF is to take the server's salt value $s \in [0, \mathsf{order}(\mathbb{G}))$ and to compute the value $s \cdot H_{\mathbb{G}}(\mathsf{password})$, where $H_{\mathbb{G}} : \{0,1\}^* \rightarrow \mathbb{G} \setminus I$ is a hash function that outputs group elements, excluding the identity element. To do this obliviously, the protocol operates as follows:

1. The client hashes its password to obtain $Q \leftarrow H_{\mathbb{G}}(\mathsf{password})$. It then selects a random "blinding" scalar $r \in [0, \mathsf{order}(\mathbb{G}))$, and sends $Q' \leftarrow rQ$ to the server.
2. The server verifies that $Q' \in \mathbb{G}$ and returns $R \leftarrow s \cdot Q'$.
3. The client verifies that $R \in \mathbb{G}$ and computes $K' \leftarrow r^{-1} \cdot R$ to remove the blinding factor. The result simplifies to $K' = s \cdot H_{\mathbb{G}}(\mathsf{password})$.

Because the group is of prime order, each value Q derived in step (1) is a generator of the full group: this means that from the server's perspective, the blinded value Q' it receives from the client is a completely random element of $\mathbb{G}$, and hence offers it no information about $\mathsf{password}$. Similarly, under reasonable assumptions, the server's response R should not allow the client to obtain the salt s. There are some caveats in implementing this protocol that are not given here, such as additional checks and the use of appropriate groups, so please see [78] for the exact details if you plan to implement this!

11.1.5 Password hardening

Password-hardening protocols are closely related to PAKE schemes. In this case, the client possesses a weak secret such as a password, and the server possess a strong cryptographic secret. The goal of the protocol is for the client to securely combine its secret with the server's, typically without revealing the password or the resulting key to the server. Many password hardening protocols also add various protections. Three common ones are:

1. **Limiting guessing attempts.** These protocols can be configured to limit the number of incorrect guesses that a client can make before the account is locked and/or erased.

2. **HSM-based password hardening.** Many systems offer "end-to-end encrypted" backup by deploying password hardening servers inside of HSMs, with the HSM configured so that the provider cannot access the secret data or reprogram the server. Combined with limited guesses, this allows end-users to protect data with relatively weak human-memorable secrets such as passwords and PINs.

3. **Distributed password hardening.** As an alternative to using HSMs (or as an extra security layer on top), some password hardening protocols *distribute* the work of the server across multiple distinct machines. Here the goal is to split (or secret-share) the server's secret so that the compromise of a single machine does not result in a loss of security. This can be done in a threshold manner, such that a client can contact any T of N servers, for $T \leq N$.

DRAFT

Apple uses a somewhat *ad hoc* password hardening protocol built from various ingredients, including the Secure Remote Password protocol and a custom HSM, in order to power its various end-to-end backup services, such as iCloud Advanced Data Protection and iCloud Keychain [20]. Signal, WhatsApp, Google Android and other services operate similar systems. Password hardening protocols can be built from schemes such as OPAQUE: for example, the key generated from the OPRF interaction can be used as a strong cryptographic key.[10] For a good example of a threshold password hardening scheme, see the open source Juicebox protocol [308].

11.2 Authentication devices

Many MFA systems make use of the "something you know" factor of authentication by using physical devices such as code-generators, authentication keyfobs, and smartphones that run various forms of authentication software. These systems generally possess a cryptographic key (either a public or secret key) that is shared with the authentication server, and they perform operations to prove possession of this value. The result provides varying degrees of security, depending on precisely which method is used.

DRAFT

11.2.1 SMS 2FA

The most common form of 2FA today is also the least secure: this method sends a short authentication code from the service to a device using the Short Message Service (SMS) provided by most cellular networks.[11] From a cryptographic perspective there isn't anything interesting to say about SMS 2FA. The service simply generates a short random number, typically a six-digit decimal code, and then transmits that to the device via an SMS delivery service such as Twilio. The device owner then enters the received code into the service's interface, *e.g.,* a website.

At a cryptographic level, SMS and other text messages are protected in transit from the tower to the device, using cryptographic key material within the handset's SIM or eSIM. Unfortunately they are not cryptographically protected in any other part of the network, which means it is possible to intercept these codes at various points between the service provider and the user, for example by abusing the cellular signaling system SS7. While such attacks are possible, SMS-2FA users in the real world are usually much more vulnerable to simpler attacks such as *SIM swapping*. Here an attacker calls your phone company and convinces customer service department to transfer (or port) your phone number to a new device, at which point all SMS authentication codes will go to them. These attacks have become hilariously common in the era of cryptocurrency, and most reputable cryptocurrency

[10] In fact, the OPAQUE standard [78] derives a second key called the *export key* that can be used for application-specific purposes.

[11] Some networks use RCS or MMS, instead. We will treat these all identically.

exchanges have deprecated SMS 2FA. In general SMS 2FA may be somewhat better than using no second factor at all, but it is usually not considered adequate for high-security applications. When SMS is used as a single factor to enable password resets, then it is arguably worse than passwords.

11.2.2 TOTP and "Authenticator apps"

A more secure approach to authentication is based on a system called Time-based One Time Passwords (TOTP). This system uses a hardware device or smartphone app to generate a short, but periodically-changing "one time password" that can be entered into a website to log in. The idea of time-based one-time passwords was pioneered by RSA Data Security, which manufactured physical devices for authenticating to networks. A TOTP algorithm was standardized by the IETF in 2011 [235], and is now used by many open source authentication apps.

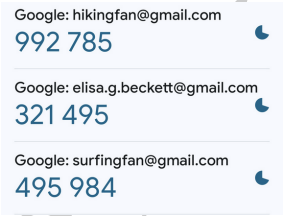

Figure 11.4: TOTP codes displayed in the Google Authenticator app. Image source: Google Play Store.

DRAFT

TOTP systems are relatively simple. A server and an authenticator device each share a per-account key K that is used to seed a pseudorandom function (PRF). This key can be stored into the device at the factory, but more commonly it is provisioned into a smartphone app by scanning a QR code. The server and authenticator must also possess synchronized clocks. Both parties will use this secret to derive a short, random code that changes every X seconds, where X is a convenient number, usually 30 seconds. The current code is displayed on the authenticator as in Figure 11.4, and can be easily entered into the service or website, which re-computes the current code and verifies that the result is correct.

TOTP from RFC 6238 [235]. We will assume that both server and client possess the shared key K as well as the current time in seconds S. They also agree on a shared "timestep" X that defines how many seconds each code will be displayed for, as well as the length D (in decimal digits) of the codes that they wish to generate (typically $D = 6$, see below.) The IETF standard allows PRF to be implemented using HMAC with ~~SHA1~~, SHA256 or SHA512

as the hash function.[12] To compute the current code, both server and client conduct the following algorithm:

1. First, compute the current time epoch $E \leftarrow \lfloor S/X \rfloor$.[13]
2. Next, compute $P \leftarrow \mathsf{PRF}(K, E)$.
3. Convert the bits of P into a D-digit numeric code as follows:
   (a) Truncate P to 31 bits.[14]
   (b) Compute $\mathsf{code} = P \ mod \ 10^D$.

I want to take a minute to note that the specific algorithms used in the IETF standard are archaic and weird, mostly because they drag in a legacy standard called HOTP [234]. In particular, the HOTP chooses to truncate a perfectly nice 160/256/512-bit PRF output string into 31-bit segments, and then perform a modular reduction to produce a numeric code. While this was presumably done to avoid bignum arithmetic,[15] it instantly creates a small bias, since 2^{31} is not evenly divisible by 10. The designers successfully argue that for the common value $D = 6$, this bias is not large enough to matter. They don't talk about larger values of D, or even place a specific upper-limit on what D should be. My strong recommendation is that you should not use HOTP for codes longer than six digits, which is probably good enough. (See §13.1, "Most cryptography standardized before 2013 was created on a dare.")

Security and implementation notes. Provided that the key is long enough, the most obvious attack on a TOTP server is to simply guess the next code. For keys of length D digits, each guess should succeed with probability about 10^{-D}, or around one in 1 million attempts for standard 6-digit TOTP. Of course, this assumes that the server will accept only one possible code for each guess. Most server-side TOTP implementations assume that clocks will be slightly out of sync, and thus will accept codes from the previous and/or subsequent epoch. This can double or triple the attacker's guessing probability. A system that allows for one-epoch time slips and accepts five incorrect code-entry attempts would still block such an attack with probability above 99.99%. More productive attacks on TOTP typically include spear-phishing and social engineering attacks, in which users are tricked into revealing their code to an attacker; these attacks are assisted by the fact that TOTP devices cannot verify the identity of the website you are entering codes into.

[12] Obviously SHA1 is not recommended for new implementations!

[13] This represents dividing S by X and then computing the floor, *i.e.,* discarding any remainder.

[14] In RFC 6238 this is done using an extremely silly procedure drawn from an older scheme called HOTP [234]. I'm not going to give the details of this function because (1) it is stupid, and (2) I don't want anyone to implement it out of this book and blame me.

[15] With thanks to Antoine Pitrou for pointing this obvious fact out to me.

DRAFT

Historical note: the RSA SecurID hack. Since we're talking about one-time passwords, it is worth briefly noting that in 2011 a master database of RSA SecurID seeds was stolen from the RSA company. Since RSA devices were provisioned centrally, this database enabled the attacker to access the systems of many of RSA's customers. The attackers used the stolen seeds to access the systems of Lockheed Martin, a major U.S. defense contractor [146]. This highlights a weakness in TOTP: like passwords, TOTP secret keys are *symmetric*. It is very important to make sure that the TOTP seeds in your server's authentication database kept secret. One should also try not to use a single seed to derive TOTP seeds for many accounts, and administrators should obviously disable existing keys if the server database is ever breached.

11.2.3 FIDO, WebAuthn and Passkeys

While TOTP supports simple authentication devices (or "tokens"), the TOTP approach has some major limitations. In particular, TOTP fundamentally depends on symmetric cryptography, which means that servers must store secret keys. Moreover, the TOTP paradigm relies on a human user to bridge the gap between token device and any website where codes are entered. Since the TOTP token cannot directly verify the identity of the website that a user is entering codes into, TOTP codes can be stolen via phishing attacks.

An alternative approach is to use more sophisticated token devices that communicate with the client device *electronically*, for example, using USB or Bluetooth, or by implementing the token in software directly within the client device. This helps to address the two shortcomings of TOTP identified above: since authentication data does not need to be manually entered by humans, tokens can employ larger authentication strings that contain digital signatures. The benefit is that servers can now hold users' *public keys* rather than secret ones. Additionally, with the proper software installed on a client device, tokens can communicate with "user agents", such as web browsers, to verify the identity of any website the token interacts with. This prevents phishing sites from obtaining useful data that can be replayed against a target site.

DRAFT

From FIDO to WebAuthn/FIDO2. The first standards for token devices were produced by the Fast IDentity Online (FIDO) Alliance, who published protocols for communication between servers, clients and token devices [210, 203]. In recent years, these original standards have been deprecated and replaced in part with an open standard produced through the W3C, called Web Authentication (WebAuthn) [161]. The FIDO alliance still produces a standard called *Client to Authenticator Protocol* (CTAP) [79] that covers client-to-device communications. The combination of WebAuthn and CTAP is now referred to as FIDO2. In this section we will mainly focus on WebAuthn.

WebAuthn. The WebAuthn protocol allows a token device ("authenticator") to prove that it possesses a secret key. This is done using digital signatures over a collection of data,

including a *challenge* issued by the remote authentication server. On the client side, this authentication protocol will often flow through an intermediate application such as a web browser, which in turn passes messages to the authenticator via some local software.

Registration. The user wishes to register with a new service (or "application") using their username. Each authenticator device has a unique device name, and the service/app has its own identifier. To register with the WebAuthn server, the server generates a list of supported cryptographic algorithms and other policy information, along with a random challenge value. The authenticator constructs a new signing keypair $\mathsf{pk}_\mathsf{C}, \mathsf{sk}_\mathsf{C}$ and sends the public key and a "handle" (a identifier for the key pair) back to the server. It also attaches a signature over a variety of data, including the username, the domain of the website (if one is used), the remote app ID, and the server's challenge.

Authentication protocol. The authentication protocol is very similar to the registration protocol. The server again generates a challenge, and requests the device to produce a valid signature over a collection of data including the username, the domain of the website, the remote app ID, and the server's challenge.

DRAFT

Passkeys. A concrete implementation of the FIDO2/WebAuthn standards is *passkeys*. Passkeys are "multi-device" credentials that are typically stored within the client's password manager and optionally synchronized to other devices via cloud-based synchronization. When a user registers with a website, the system can offer to install a passkey on their client. This provides a high-security alternative to normal password-based authentication. A useful feature of passkeys is that they are *discoverable credentials*. This means that each passkey contains the username, so that the user does not need to first log in using their account username. Instead the server can direct the local client software to search for any stored passkeys, and then allow the user to select the one they wish to use.

Once stored on a system, passkeys may be further protected with local passwords and/or biometric authentication if it is available on the system. For example, in Figure 11.5, a user must use a Mac's Touch ID sensor to authorize the use of a passkey. Passkeys can also be stored onto removable token devices, or stored in separate devices such as phones.

Privacy concerns. An obvious risk of using discoverable credentials is that a malicious website or advertising script might attempt to enumerate a list of all passkeys stored on a user's system, using the local interface `navigator.credentials.get()`. To mitigate this, the WebAuthn standard directs browsers to ask the user for permission before allowing a remote site to enumerate available passkeys. In addition, credentials can only be made available if they match the top-level website making the request. This means that `advertising_company.com` cannot enumerate credentials associated with `news_website.com`, even if a script or iframe from the former has been included within the latter (without explicit authorization.) Applications on iOS and