Modern password security for system designers

What to consider when building a password-based authentication system

By Ian Maddox and Kyle Moschetto, Google Cloud Solutions Architects

This whitepaper describes and models modern password guidance and recommendations for the designers and engineers who create secure online applications. A related whitepaper, Password security for users, offers guidance for end users. This whitepaper covers the wide range of options to consider when building a password-based authentication system. It also establishes a set of user-focused recommendations for password policies and storage, including the balance of password strength and usability.

The technology world has been trying to improve on the password since the early days of computing. Shared-knowledge authentication is problematic because information can fall into the wrong hands or be forgotten. The problem is magnified by systems that don't support real-world secure use cases and by the frequent decision of users to take shortcuts.

According to a 2019 Yubico/Ponemon study, 69 percent of respondents admit to sharing passwords with their colleagues to access accounts. More than half of respondents (51 percent) reuse an average of five passwords across their business and personal accounts. Furthermore, two-factor authentication is not widely used, even though it adds protection beyond a username and password. Of the respondents, 67 percent don't use any form of two-factor authentication in their personal life, and 55 percent don't use it at work.

Password systems often allow, or even encourage, users to use insecure passwords. Systems that allow only single-factor credentials and that implement ineffective security policies add to the problem. Arbitrary and inconsistent rules nudge users to handle their passwords insecurely, and password recovery systems can leave the user and the application vulnerable to threat categories they might not have considered.

Overview

This document outlines:

- Trusted sources of thoughtful and researched information about password security.
- Recommendations for engineers who design password management systems.
- Common anti-patterns and urban legends around password security.
- Topics for additional research.

Terminology

character set
A defined list of characters that is recognized or supported by hardware or software.

entropy
Related to passwords, a measurement of how unpredictable a password is. Commonly represented as a number of bits.

hash
The result of one-way, mathematically irreversible, and deterministic encryption algorithm.

MFA, 2FA
Multi-factor authentication or *two-factor authentication*. Methods to add verification for protecting account data, traditionally in addition to a password.

OAuth
An open standard for token-based authentication and authorization on the internet.

OTP
One-time password (sometimes *one-time PIN*). A single-use code often used in 2FA verification.

password manager
Software that assists in creating, storing, and retrieving passwords.

rainbow table
A systematically generated table of precomputed hashes and their plaintext values. Used to look up hashes to get the corresponding pre-hashed value.

salt
Random data added to a value to be hashed. Usually used to produce an alternative hash of the original input in order to break rainbow tables.

virtual device MFA
A multi-factor authentication system that's available as software. Commonly installed on mobile phones.

Password considerations for systems designers

The responsibility to encourage strong passwords and secure those credentials belongs to system designers and security engineers. As the individuals responsible for implementing the authentication rules of your applications, you have the power to encourage (or thwart) strong password use.

Best practices for system designers

DO	DON'T
Allow the largest character set possible, such as UTF-8, including emoji.	Apply arbitrary constraints like banned characters or restrictions on certain character sets.
Have a long minimum length and allow very long passwords.	Have a minimum password length of less than 8 characters.
Show that any password requirements are met or provide a secure password generator.	Tell a user that their password is "strong."
Check new passwords against leaked databases, and other account information.	Allow the use of P@ssw0rd or other well known credentials.
Allow password pasting.	Truncate or trim password text field input.
Support MFA.	Only support SMS-based MFA.
Have robust password reset capabilities.	Use security questions unless you have a good reason.
Have neutral error messages that don't reveal information to a malicious actor.	Give away clues in your error messages such as "no such user" or "bad password".
Monitor for abnormal account behavior and communicate with customers.	Force regularly scheduled password rotation.
Salt and hash your passwords in storage.	Invent your own hashing algorithm.
Encrypt all data in transit and at rest.	Store any passwords in plain text or provide lost password recovery.
Offer offline account recovery codes.	Enable users to easily lock themselves out forever.

Password generation methods

If you are designing authentication systems for a cloud-native world, you should make it as easy as possible for your users to follow strong security standards while using your application. The most important thing is that you don't force any unnecessary standards or anti-patterns to increase security. It's fine to limit lower-end security (minimum password length, for example), but avoid any limits on the higher end of the security spectrum.

If you are designing your system today, remember that it will probably still be in use many years from now. The only constant you can rely on is change, and you want to architect your application to provide higher security options well into the future.

Allow the largest character set you can

Supporting 7-bit or 8-bit ASCII for passwords is not enough. Your objective should be to allow the full list of UTF-8 printable characters.

Technical note on Unicode

> RFC 8265, or Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords, is a proposed standard. This RFC states that Unicode passwords must be normalized using the Normalization Form C (NFC) standard where characters are decomposed and then recomposed by canonical equivalence. This is important to avoid the problem of the same password entered on different systems not matching.
>
> For example, the distinct Unicode strings `U+212B` (the angstrom sign Å) and `U+00C5` (the Swedish letter Å) are both expanded by NFD (or NFKD) into the sequence `U+0041 U+030A` (Latin letter A and combining ring above °), which is then reduced by NFC (or NFKC) to `U+00C5` (the Swedish letter Å).

If you have a reason to restrict the character set, make sure you fully understand this restriction and implement it evenly across all platforms you support. Legacy applications can be tricky and often contain password restrictions that are now anti-patterns. However, there are several methods available to hash and reduce passwords to something even a space-constrained legacy database can handle. If you are properly sanitizing and hashing password inputs, there are few legitimate reasons to limit character sets.

Valid character-set limitations often depend on context or interface restrictions. For example, HTML password fields cannot accept the carriage return (`U+000D`) and line feed (`U+000A`) characters. Some applications might automatically trim whitespace from inputs, which affects users who use leading or trailing spaces in their passwords.

Systems that rely on PIN pad or telephone authentication might need to restrict their character set to digits or the basic English alphabet or engineer their systems for varying keypad layouts, as in the following pictures.

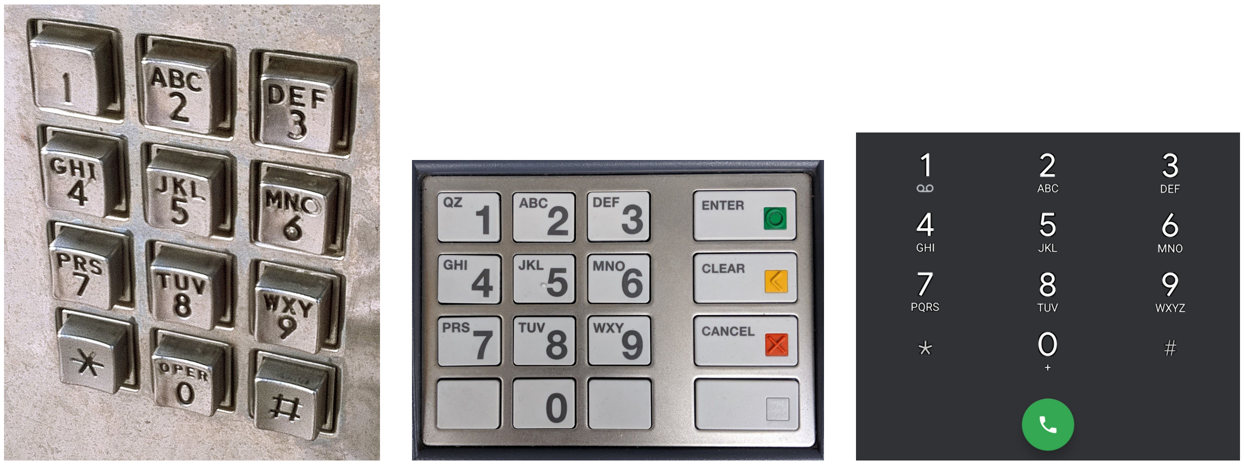


Figure 1. Three numeric keypads with different placements for Q and Z.

Do not unduly restrict the length of passwords

Allow a length that enables high-entropy practices. When choosing a minimum password length, ask yourself: What minimum length will reasonably require my users to have good password practices? A quick check of common services shows that every company seems to implement different password requirements. Understand and document the categories of risk you want to protect your users against. Are your users' potential adversaries social friends, digital thieves, or foreign states? To what extent do you need to protect your users and your service from these adversaries? Understand the methods used by these threats and design your requirements accordingly. In practical terms, this means choosing a minimum password length that reflects the value that attackers will place on cracking your users' passwords.

Make the maximum password length as large as your application can allow. What is the maximum password length your application can support before it impacts performance or storage? This is your ideal answer for maximum password length. Cryptographic hashing of passwords is a basic requirement for credential storage, and most hash algorithms yield a fixed-length output no matter the input size, eliminating most storage concerns. Web applications are commonly configured to limit HTTP POST requests to 2 MB or 10 MB. Such hard technical limitations should form the basis of your password length upper limit.

Don't truncate password input. This practice applies both in the interface and on the server side. If a user supplies a 32-character password and you quietly trim it to 8 characters, or the HTML input box only accepts the first 16 characters, you create a host of long-term problems for your application and users. Instead, inform the user of a password that's too long through UI feedback on any form where the password can be submitted, set, or changed. Password managers commonly generate long passwords, and some cannot determine if the input was truncated by the UI.

Relying solely on the input field's maximum length is problematic because it can be difficult to see whether the box is accepting new characters:

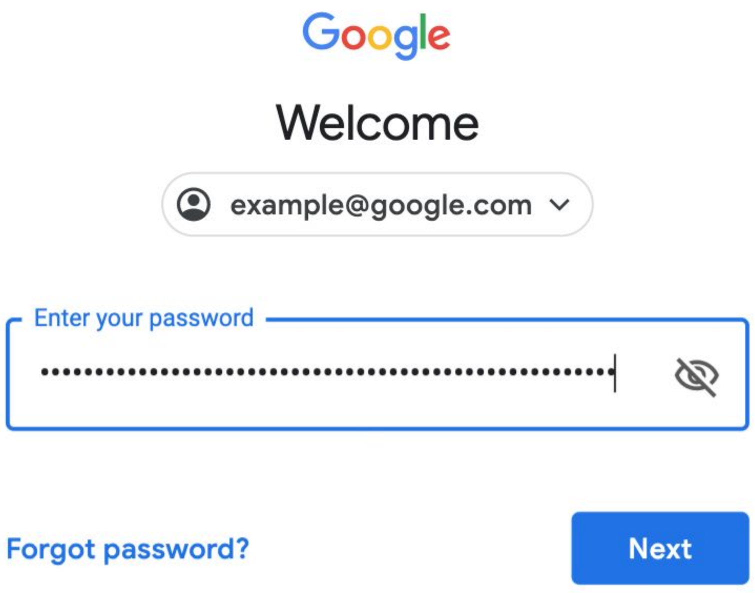


Figure 2. "Did it get those last few characters?"

A better approach is to change the interface or display a warning if the field maximum has been reached, even if that maximum is hundreds or thousands of characters.

Understand password psychology

In theory, system creators should enforce password entropy based on full use of the character set. The classic pattern of “1 uppercase, 1 lowercase, 1 number, and 1 special character” plus the minimum password length provides a reliable password cracking formula for a bad actor seeking to minimize the problem space of cracking a password from that system. In reality, enforcing these restrictions causes users to do predictable (read: nonrandom) things. These users tend to choose passwords that rely on similar character substitution—a well-known and readily defeated strategy. A typical user response to these requirements is to add a single digit or special character to the end of their favorite password, or to capitalize the first letter of dictionary words. Even though “P@ssw0rd” or “Password1!” meet the classic pattern requirements, they are not strong passwords.

Encourage good password practices

System administrators often assume that increasing the minimum password length will lead to offices covered with passwords written on sticky notes. You can avoid this problem by nudging users in the right direction. You can use a number of methods to gently encourage users to follow best practices:

- If you want each user to pick a complicated password, consider putting a password generator function in the frontend code of your website, or generate a password and give it to the user over a secure channel. The goal here is to make it as easy as possible for your users to make a strong password choice.

- If you want your users to use a password manager, provide links to some popular solutions on your account setup pages and password reset pages.

- If you want users to know what makes a good password, link to some articles discussing the topic, like the companion piece to this whitepaper.

- By computing entropy of new passwords and showing it to the user in real time, help users make good decisions. While this approach is not infallible, it can gently steer users away from bad decisions. This action should always be paired with checking the password hash against leaked passwords at creation time, as described in the next sections.

Ensure broad usability

Some applications disable the ability to paste or programmatically enter text into the password field. This is an anti-pattern for both security and usability. Such restrictions can drive users to use insecure practices or discourage them from using your application at all. Some accessibility services rely on such mechanisms to work. At a minimum, it will annoy those users with the strongest security practices. This is counter to the goals of any application security developer.

Check for predictable passwords

The best time to check a user's password strength is when the user is creating or changing their password. Compare new passwords to other information previously provided by the user. Two commonly reused elements in a password are user name and the email address. Others include birth year and organization name or motto. Depending on your application, there might be other pieces of data you should check. You should have a rule that disallows the appearance of any of these values in a password. Use algorithms such as soundex or Levenshtein distance to detect small tweaks that try to subvert such a rule.

Second, check the password against previously leaked passwords. In order to avoid leaking security information to a third party, your application will require its own copy of a leaked password database like the ones provided by haveibeenpwned.com. Use this database to check every new password. If you detect that a leaked password has been used, tell the user right away in the application.

Depending on your application architecture, you might be able to check your entire password database against a leaked password list. If you find matches, you can then notify users that their password has been compromised and recommend they change it. However, this might be cost prohibitive if each user has a unique salt or you've used an acceleration-resistant hashing algorithm, per OWASP best practices. For existing password scans, it is reasonable to security-check the provided password on successful login.

There is a big difference between telling a user that they have provided all of the minimum requirements for using a password, and telling the user that their password is “strong” or “weak.” Showing a user that they have met the requirements of your password policy is easy to do with confidence, and passes no judgement on the user’s security choices. Telling a user that their password is strong, however, gives them a false sense of security and is ultimately a risky approach. Unless you are intelligently looking at each password in relation to entropy and similar-character replacement, and regularly checking against known password compromises, you cannot definitively say that a password is strong. With only basic entropy calculation, you can confidently identify weak passwords and provide a best-case-scenario estimate of the password's strength, but you will also identify several types of weak passwords as high-entropy.

Provide a basic password generator

Consider providing a secure password generator on the signup and password-change interfaces. Users are much more likely to use a high-quality password if you don't make them work to get it. Use your own rules for length and entropy, and offer the user a securely generated random password or passphrase. You might be surprised by how many users take advantage of it.

If you create your own password generator, consider applying algorithms that prevent specific short words from appearing in the output. There are many three- and four-letter combinations that users might consider offensive. Their random creation might reflect poorly on your product or service and can generate unwanted media attention. The reduction in possible letter combinations will be statistically insignificant and should be worth the decrease in total randomness.

Support multi-factor authentication

Passwords or passphrases represent a single factor to authenticate a user. If having just a password will authenticate a user, then your application only requires one factor to gain access to a resource. Multi-factor authentication (MFA) is sometimes also referred to as two-factor authentication (TFA/2FA) or occasionally three-factor authentication (3FA). Regardless of the name, multi-factor authentication means using more than one of the following factors:

- **Something you know**. Things that you can remember, memorize, or store. Passwords, passphrases, and personal identification numbers (PINs) are the most common examples of something you know. These provide only a modest level of security, because they can be known by others using a variety of methods. For example, a user might tell someone, someone might watch the user type it, or someone might read it off the user’s sticky note.

- **Something you have**. Things that you control in physical space. Common examples include smartphone MFA apps, common access cards (CAC), security keys like the Google Titan key, or RFID fobs (see Figure 3). This factor provides slightly higher security than “something you know” because you have to be in physical possession of a device. This factor is difficult to copy and cannot be easily shared.

- **Something you are**. Unique qualities that define who you are. These are sometimes referred to as biometrics. Fingerprints, palmprints, voice patterns, iris patterns, and facial structures are all commonly used for something you are. This factor is extremely difficult to copy and cannot be shared, however there are many implementation-specific exploits you must research and consider carefully.

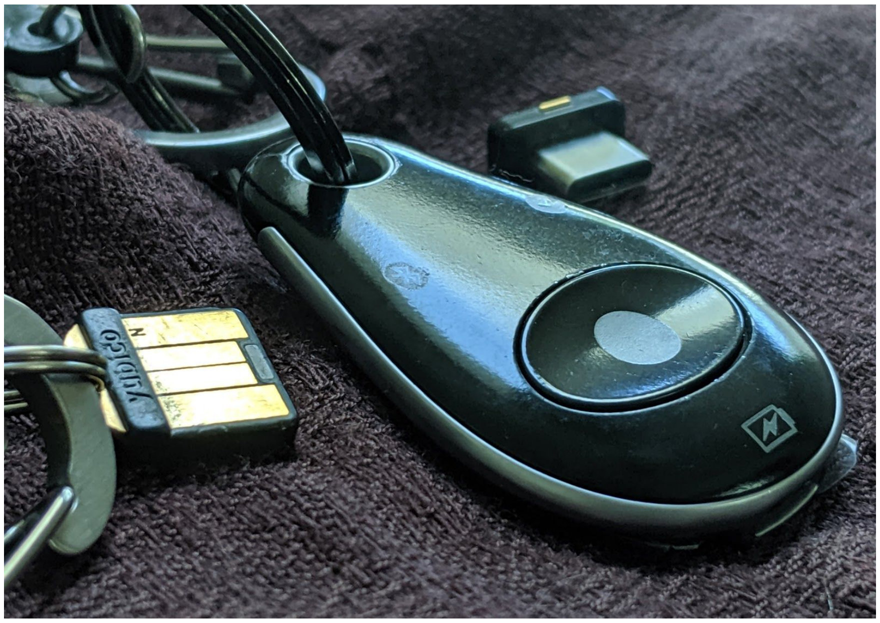


Figure 3: Several hardware MFA devices

Hardware and virtual device MFA

Try to make every effort to support hardware and virtual device MFA. The availability of smartphones allows for easy and rapid adoption of virtual device MFA. Smartphone apps like Google Authenticator allow for secure storage and generation of many MFA device keys. Some virtual device applications and devices even back up the authenticator keys to secure cloud storage. If your users have limited access to a smartphone or are working with offline systems, many hardware-based token options like the ones pictured in Figure 3 support this choice.

Risks of SMS with MFA

Avoid the use of SMS-based MFA. SMS is an insecure technology that is easy to compromise or spoof with no authentication mechanism or eavesdropping protection. Messages can be hijacked by a malicious app, or a malicious actor could intercept the message by spoofing the device or by using social engineering to transfer service to a device they control. For these reasons, it makes much more sense to use email or app push notifications instead of SMS for MFA.

While it might not make sense to require three levels of MFA, it's a good idea to offer the most secure second factor that you can for your users. This factor will usually be something a user has, like a smartphone MFA application or security key. Enabling third-party identity or single sign-on (SSO) providers and piggybacking on their MFA support is a simple means to boost your users' security. For more details, see Supporting single sign-on.

Supporting security questions

Security questions are a holdover from in-person and telephone banking in the last century. It used to be the case that discovering answers to the most common questions was difficult for third parties. Mothers' maiden name, the street you grew up on, your first pet's name, the city you were born in… all of these are commonly used security questions. Today, these questions can often be answered in minutes through public records searches and background checks. Security questions are an anti-pattern of security for these reasons:

- **Non-unique knowledge**. The answers to many questions are not a secret. The street you grew up on is probably known by at least a few family members and friends. Or people who know your parents' name might perform a public records check on their previously known addresses and cross-reference your approximate age. If you've ever posted about your pets on social media or talked about them in public, you shouldn't use pet names. A password is supposed to be a private key that only you know. Most security questions require answers that are probably known by (or can be discovered by) more people than just you.

- **Multiple password problems**. For users who understand how easy it can be to find the correct answers, the natural option is to introduce non-real answers that are more secure. For example, a user might tell the system that their first pet's name was 5zvw496n8p7b. In effect, they now need multiple passwords to authenticate one account. This might add security, but does not lessen the administrative overhead of managing multiple passwords or eliminate the threat of exposing the single-factor credential, even if it is effectively broken into two passwords.

- **The weakest-link problem**. Most authentication systems that use security questions gather multiple challenges to pick from as needed. Sophisticated systems keep asking the same question for every challenge until a correct answer is received. The rest offer a different or random question at each authentication attempt. This opens the system to a weakest-link attack where an unauthorized individual can collect all of the questions and find the answer to the easiest one.

Consider the use of security questions in the context of your application. Many of the reasons organizations rely on them can be addressed through other means:

- An OTP sent to the users' verified email address.
- An MFA device or application.
- An OTP sent to a mobile app, or a browser push notification sent to already-authenticated sessions.

You might have inflexible regulatory or policy reasons for using security questions instead of MFA. Perhaps your user base cannot be relied upon to use other technology. Whatever the reason, if you cannot avoid security questions, ensure that you aren't creating a weak link. If an account fails a security question test, keep using that same question for every auth attempt until it is answered correctly or the user resets their password.

Limit information disclosure

As a general rule, don’t disclose any clues about an account to unauthenticated users. If a login fails based on invalid user input, provide the same non-specific error (such as "Invalid login") for bad username and for a bad password. Never provide contextual information such as "invalid password" or "account not found". When you provide information to your users in a password reset, keep in mind the fundamental difference in the following two messages:

- “A password reset email will be sent to that address if it matches an account”
- “No account exists with this username”

The first message offers a neutral response to an unauthorized query. “*If* this is an account, and I’m not saying it is, then you should expect an email soon.” There’s no additional information, and no disclosure of anything sensitive. This is an ideal piece of text.

The second message divulges information. “I can confirm that isn’t an account” gives a malicious user a method to verify accounts. They just have to walk through email addresses until the reply message changes. This makes it easier for unauthorized users to gain information they shouldn’t have.

Password hints are another example of inappropriate information disclosure. They are literally free information about a password to *anyone* who wants it. Consider if your front door had a note stating, “the key is hidden near a plant.”

Password maintenance methodologies

After your users create passwords, your application has an ongoing responsibility to secure and protect that information.

Avoid password rotation

Mandatory password rotation is discouraged by every major authority (NIST, NCSC, OWASP, and others), yet continues to be normal behavior for many applications and for a few outdated compliance standards. The requirement to reset or change a password at regular intervals often runs headlong into human nature and leads to users writing passwords on sticky notes, storing them in insecure locations, or incrementing passwords (for example, adding 1 to the number at the end of their previous password).

In the best-case scenario, your user has a difficult password and uses a password manager. Password rotation is simply a few clicks for them as they update their password manager. These users gain little security advantage by password rotation.

In a worst-case scenario, your user has a traditional password that violates best practices but is easy for them to remember, such as DogCat91. One of two things will happen when that user is required to change their password:

1. DogCat91 will become DogCat92, then 93, 94, and so on.
2. The user will get tired of playing the “add one” game, and just start to write the new password down. “Surely people are smarter than that,” you say? The 2019 study from the Ponemon Institute showed that 52% of users surveyed wrote down their passwords or stored them in an insecure file.

Make password lockout consistent

Password lockout is another area where practices vary widely. Should an account lock out after 3 unauthorized attempts in a row? Or 5 attempts in a 10-minute period? Or 10 in a 24 hour period? Do you lock and unlock the account silently? Do you email the user? What if this account is their email account? Do you reenable the account in an hour? 15 minutes? A day? There are so many options to accomplish the same goal—don’t allow unauthorized access. Everything beyond that depends on your risk level, on your users' technical ability, and on your application.

Ask yourself what the highest security is that you can provide using only the tools you already have available. The answer is password reset! For example, if a user account has more than a few consecutive failed login attempts, say 5, over any timeline, invalidate the existing password and immediately send the user an email explaining the situation in simple language:

> “We noticed that someone tried unsuccessfully to log into your account 5 times. Your account and your data are still intact, but for your security, we’ve locked your account. We need you to reset your account password before you will be allowed to access your account again. Please click this link to do so.”

You can go further and also add the IP address of any failed login to a database and track activity and monitoring for anomalous behavior. (More on that later.) You might also use services and application plugins that share blocklists of IP addresses that have attempted to compromise sites.

Make password reset easy for users

Password reset is the process of invalidating the existing password and requiring the user to create a new one. This reset should be an easy process for the user. You can usually send a time-limited password reset URL to them in email. Other options include OTP delivered through in-app push notifications for smartphone apps, in-person verification for brick-and-mortar businesses, and postal mail for applications where password recovery is valuable but not time sensitive.

No matter what your situation is, think of password reset as a backdoor into the account that can be used by anyone at any time. You are reducing the security of your user accounts to that of whatever service controls password reset, so design your system accordingly. Consider a CAPTCHA (such as Google reCAPTCHA), MFA, or other advanced methods to increase the security of your password reset process.

CAPTCHA, for example, is used to distinguish humans from bots, which might or might not be used for brute-force attacks.

Beware of password recovery

Password recovery is where your user is able to get a copy of their existing password from your application. Password recovery is perilous for user and system security, so applications should instead rely on password reset whenever possible. If you are able to unilaterally produce a plain-text version of a user's password, you are storing passwords either with reversible encryption or in plain text. In most circumstances, this is an anti-pattern verging on unacceptable security. You do not want to be responsible for divulging a users' password to an unauthorized individual.

When you create software, it’s best to assume that your database or code, or both, will someday be compromised. Companies who lose control of plaint-text customer passwords might face legal liabilities and financial penalties from consumers and government agencies. Several large data breaches have resulted in multi-million dollar settlements.

Monitor anomalous behavior

As a designer, you should have an idea of what normal behavior looks like within your application. Designing software around this behavior lets you respond automatically to abnormal behavior. The key is knowing what to monitor and how to respond. Some methods are common, like those outlined in the following list, but every application is unique, and you should design application security with monitoring in mind.

The following examples represent a minimum base for monitoring user behavior:

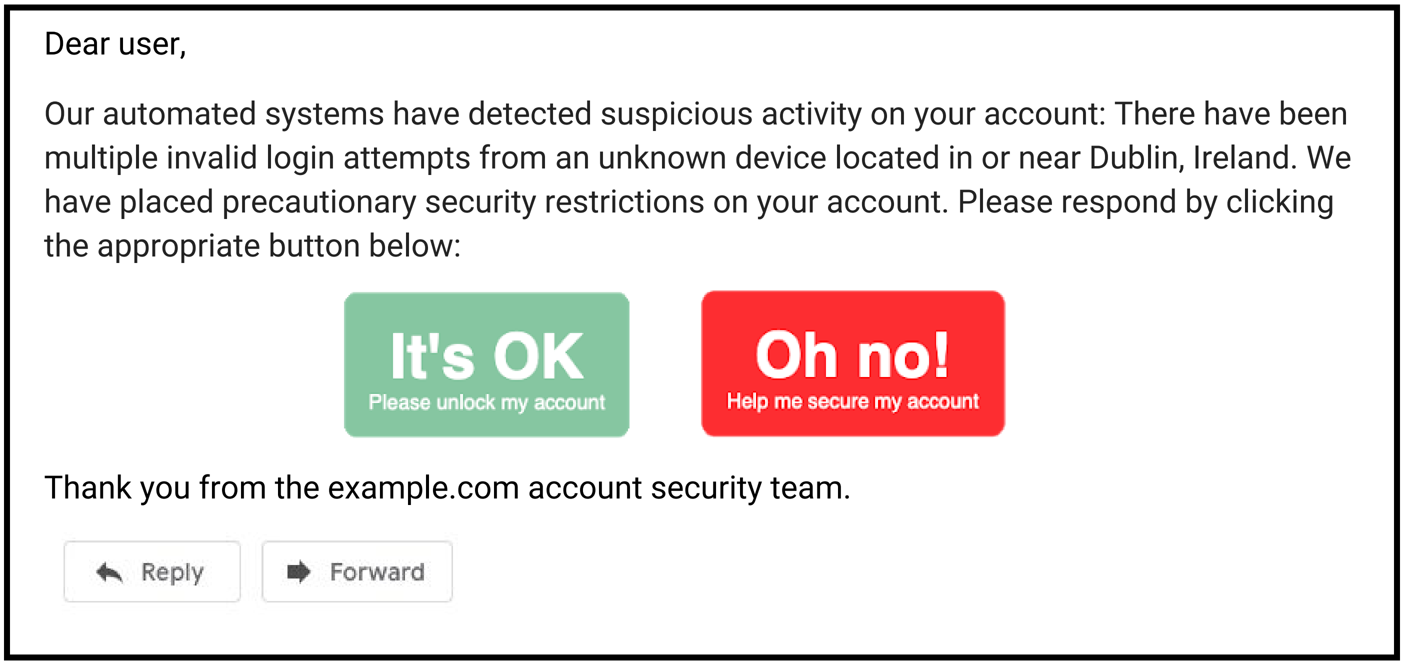
- **Volume**. Understand how often users access the application and log into the service. If a user normally logs in a couple of times a week, but has tried 6 times in the last 10 minutes, you should flag that as abnormal.
- **Geography**. Monitor the geographic origin of login requests. If a user has always logged in from Canada, but suddenly logs in from South America, that should be flagged as abnormal.
- **Method**. Understand customer login methods. If a user always logs in from the mobile app, but suddenly tries to connect through a RESTful API, that should be flagged as abnormal.
- **Failed attempts**. If a user's failed login rate spikes to abnormal levels, that should be flagged as abnormal.
- **Incomplete MFA login**. If a user attempts authentication but is thwarted by MFA checks, that should be flagged as abnormal.

Remember that abnormal doesn’t always mean malicious. Maybe your user is on vacation, or decided to start using your application’s API in a new way. You want to trust your users, but also verify through other communication channels.

If you notice abnormal behavior, contact the user. Banks have been doing this for years when they detect questionable transactions. Consider this example:

> "We recently noticed some abnormal behavior on your account. (*List the abnormality.*) If this is expected behavior, please click this big green It's OK button. Otherwise, click this big red Oh no! button to alert our security team."

If you don't receive a response in a reasonable time, consider forcing a password reset.



Dear user,

Our automated systems have detected suspicious activity on your account: There have been multiple invalid login attempts from an unknown device located in or near Dublin, Ireland. We have placed precautionary security restrictions on your account. Please respond by clicking the appropriate button below:

It's OK
Please unlock my account
Oh no!
Help me secure my account

Thank you from the example.com account security team.

Figure 4: An account security notification email

Another option is to restrict account privileges until the user responds. For example, when you detect abnormal behavior, send the same email to the customer, but immediately put the account in a read-only or limited-functionality state. Don't allow any edits, changes, or admin functions until the customer responds to the email. You should also place some notification in the application or through the API call that the account has been limited. After you know that the behavior is legitimate, restore normal account operation. If there is no response to the notification after a time, consider initiating password reset on the account and notifying the user.

Learn from common attacks

There are many lessons to be learned from common methods that malicious users use to either gain information about your system or generally act in bad ways:

- **Don't let the password lockout mechanism act as a denial-of-service vector**. A malicious user shouldn't be able to trigger an account lockout that causes the user to lose data or service.
- **Add a proof-of-life system such as reCAPTCHA**. To ease your users' experience, require the test only on increased-risk logins such as unrecognized browsers, repeated attempts, or requests that originate from suspicious locations.
- **Prevent internal system behavior from being detected and measured** by malicious users who pay attention to timing and speed of interactions. A simple script can try many possible usernames or email addresses and monitor the amount of time the system took to say "Invalid login". A nonexistent account might take less time to trigger an error because the initial lookup of the username would return empty. Use an algorithm that takes the same amount of time regardless of whether the account exists to thwart this type of attack.
- **Use software or a web application firewall to detect repeated login attempts** from the same location against multiple accounts. A bad actor might probe your system to find out the maximum number of password attempts that your system allows, and then try that number of common passwords against multiple usernames or application endpoints. Adding firewalls also helps mitigate other forms of attacks.

Secure password handling

Collecting, transmitting, and storing passwords requires the following techniques to help ensure security.

Salting and hashing

Your service should store a cryptographically strong hash of passwords that cannot be reversed. *Hashing* is the process of performing a one-way transformation of any string of bytes into a different and deterministic new string of characters. There are many examples of hashes that are appropriate for use when storing passwords, such as PBKDF2, Argon2, Scrypt, or Bcrypt.

Avoid the use of deprecated hashing technologies such as MD5 and SHA1, and under no circumstances should you use reversible encryption or try to invent your own hashing algorithm.

The hash should be salted with a value that's unique to that login credential. A *salt* is a cryptographically-strong random value that is combined with the value to be hashed. The salt is known to the system and is unique for each password stored. Consistently mixing the salt with the password each time the password is hashed creates a unique hash value even if others use the same password. Using a salt value prevents the use of precomputed rainbow tables that can be applied to every hashed password stored in your database.

Validate passwords

When a user attempts to register or log in to your site or service, the application should first perform a one-way hash on the password. This provides a fixed-length and reduced character-set string that can be easily compared and stored. The best practice is to use a unique salt for each user, so your application would need to retrieve the salt value before hashing.

After the password is hashed, you should discard the original password from system memory. Some application architectures might otherwise accumulate passwords in memory, which could increase the damage of an attack or leak sensitive information in the event of a memory dump.

To validate the hashed password of an existing user, fetch the hash value from the password store using the user ID or username. Don’t include the password hash in the query, because it could be logged as part of normal debug or audit processes.

Design to mitigate the worst-case scenario

You should design your system by assuming that it will eventually be compromised. Ask yourself, "If my database is exfiltrated today, how would my users' safety and security be in peril on my service, or on any other services they use? What can we do to mitigate the potential for damage in the event of a leak?"

Remember that even if you consider your application to be a low-risk target, many of your users will reuse the credentials they use for email, payment services, banking, and ecommerce.

Server-side reversible encryption

Storing encrypted passwords (instead of just hashes of the passwords) is an attractive choice, but in some ways, it is as problematic as plaintext. If the keys are improperly stored or are breached, the passwords might be compromised. Applications have few legitimate uses for a decrypted password aside from creating a hash. Account recovery should be handled by having the user create a new password instead of recovering the existing one. Login attempts from the user should lead to the hash of the password input being compared to the hash that’s on file; a decrypted string comparison is unnecessary in this case.

Preventing password reuse

If your security policies include restrictions on password reuse or similar passwords, these can be implemented without storing a plaintext or encrypted copy of the password. Each time a user creates a new password, store the hash of the old password and any anticipated variations of the new password in a blocklist for that user. These hashes can be compared to the hash of any new password that the user attempts to create in the future.

Client-side hashing

Some applications use software built into their web pages or mobile application to hash data before sending it to the server. If this sort of client-side hashing is a part of your app, be sure to use a multi-layer strategy. Have the client computer hash the password using a cryptographically secure algorithm and a unique salt provided by the server. When the password is received by the server, hash it again with a different salt that is unknown to the client. Be sure to store both salts securely. If you are using a modern and secure hashing algorithm, repeated hashing does not reduce entropy.

Encryption in transit and at rest

In the modern web, it is fully expected that all sensitive interactions with a server should be over HTTPS. Make sure your web server uses a secure suite of ciphers and is hardened to prevent cipher downgrade attacks. When exchanging data between servers or services, ensure those communications are encrypted in transit using similar TLS-controlled methods. As of this writing, TLS 1.1 and earlier (including SSL 3.0) are past their end-of-life and should not be used.

Whenever you store password hashes in a database, ensure that the data is encrypted at rest, especially in backups. Google Cloud, Amazon Web Services, and Azure all provide encryption-at-rest services for their cloud services. This way, a stolen database backup cannot be used to divulge all your users' password hashes.

Session management

User session length is another aspect of user security that has no universal standard. Sessions themselves have varying definitions depending on the technology being used. For the purposes of this whitepaper, a session is any means of maintaining authentication and application state across multiple requests or interactions with your system. The focus here is on how long the session lives and what triggers invalidation, not on the mechanics of how those sessions are maintained or invalidated.

The way sessions are typically handled depends on the type of application being used. For example, most websites eventually automatically log users out, while most mobile applications never expire sessions. User session length is an important part of application design and security. Many identity providers, including Google, put a great deal of effort into session management and continually validating user identity.

Some services are more sensitive than others, and each should take a pragmatic approach to session length. It’s best to avoid infinite sessions for web applications, but you should carefully choose the session timeouts for your application and what events might prompt a session reset. For example, certain activities like administrative functions, changes to email address, and password resets should always prompt for a new session. These sensitive operations should cause your application to verify user identity. You can prompt for an MFA verification or require a full login, depending on your application requirements.

When your service expires a user session or requires re-authentication, prompt the user in real time and provide a mechanism to preserve any state that they haven't saved since they were last authenticated. It's frustrating for a user to fill out a long form, submit it some time later, and find out all their input has been lost and they must log in again.

Finally, provide a means for users to expire all active sessions associated with their account. This gives users the ability to quickly secure their account if they leave a session running on a shared computer or are concerned about unauthorized access. You should also trigger this same kind of session reset any time you lock an account. It might also make sense to invalidate all a user's active sessions when that user changes their password.

Supporting single sign-on

It often makes sense to add the ability for your users to log in through another trusted service. This is referred to as single sign-on (SSO) or federated identity. If implemented properly, SSO is beneficial to users and to your application. When you implement SSO, the user's identity in your application should be a separate logical entity from the authentication schemes you support to access that account, whether the scheme is a password, biometrics, or several different SSO providers. In other words, one user should be able to have many authentication methods without creating duplicate accounts. A user who authenticates to your service using their username and password one week might choose SSO the next week without understanding that this could create a duplicate account.

Your backend must also account for the possibility that a user gets part or all the way through the signup process before they realize they're using a new third-party identity that's not linked to their existing account in your system. You can do this by asking the user to provide a common identifying detail, such as email address, phone, or username. If that data matches an existing user in your system, require them to also authenticate with a known identity provider and link the new ID to their existing account.

Alternatives to passwords

No guide to passwords would be complete without mentioning the alternatives to passwords. It seems every few months there is an article with a headline claiming passwords are dead. The content is often a nuanced view on web security, or it's promoting 2FA or a password manager.

This section covers the handful of alternatives to passwords.

Digital certificates

Systems administrators often use authentication by digital certificate. This kind of authentication is one of the oldest non-password mechanisms in use today, commonly employed for SSH authentication. It is a cryptographic key, typically stored in a text file that can optionally be encrypted with a password.

SQRL

The Secure Quick Reliable Login protocol is a recent addition to the security space. It is designed for end-user authentication to websites and applications. SQRL users run a small client application on their computer or in their browser. Instead of giving servers a password that they must keep secret, the client provides a public key that is unique to the application or domain the user wants to authenticate to. The server provides a unique value to the client, and the client must then use their private key to sign and return that secret. The server verifies the signature by using their public key and authenticates the user. Most importantly, a compromised site or service cannot expose its users' credentials in a way that impacts any other site or service.

Users can expect to see the option for SQRL login to appear in more places in the coming years. Developers and software architects will appreciate the deep level of technical detail written with an eye toward an evolving security landscape and the way real humans interact with security controls.

Biometrics

In an ideal world, biometrics are one of the best authentication mechanisms. Verifying identity by using the unique and inherent attributes of an individual is appealing. In reality, most consumer-grade biometric security systems are vulnerable to spoofing attacks, which makes them a poor choice for highly sensitive systems. Other problems arise when a user's biometric data is obtained without their knowledge. Use cases such as sharing access or turning over access of a deceased user are difficult to manage if there is no backup authentication system.

Biometric technology is improving each year, but its security is most effective on consumer applications when used as a 2FA, instead of as a primary factor. Enterprise implementations can be significantly more secure, but before implementing such a system, you should still take care to understand the failure modes and vulnerabilities.

Device-based authentication

This increasingly popular authentication mechanism uses an active session on one device to authenticate a user on another. It is not the same as 2FA because the device is the primary means of authentication. For example, a user presents their identity to a website, which then sends a push notification to their smartphone for verification. In addition to being useful as traditional login MFA, this approach can be useful as a primary authentication mechanism on kiosks, consoles, and shared systems where entering a user's password is difficult or ill-advised. It can also be used to revalidate expired sessions or to add security to higher-risk activities such as account management.

What’s next

- Read the 12 best practices for user account, authorization, and password management.
- Review Google research on good account hygiene's effectiveness against hijacking.
- Read more about modern password security for users.
- Try out other Google Cloud Platform features for yourself. Have a look at our tutorials.