Authentication Cheat Sheet

Introduction

Authentication (**AuthN**) is the process of verifying that an individual, entity, or website is who or what it claims to be by determining the validity of one or more authenticators (like passwords, fingerprints, or security tokens) that are used to back up this claim.

Digital Identity is the unique representation of a subject engaged in an online transaction. A digital identity is always unique in the context of a digital service but does not necessarily need to be traceable back to a specific real-life subject.

Identity Proofing establishes that a subject is actually who they claim to be. This concept is related to KYC concepts and it aims to bind a digital identity with a real person.

Session Management is a process by which a server maintains the state of an entity interacting with it. This is required for a server to remember how to react to subsequent requests throughout a transaction. Sessions are maintained on the server by a session identifier which can be passed back and forth between the client and server when transmitting and receiving requests. Sessions should be unique per user and computationally very difficult to predict. The Session Management Cheat Sheet contains further guidance on the best practices in this area.

Authentication General Guidelines

User IDs

The primary function of a User ID is to uniquely identify a user within a system. Ideally, User IDs should be randomly generated to prevent the creation of predictable or sequential IDs, which could pose a security risk, especially in systems where User IDs might be exposed or inferred from external sources.

Usernames

Usernames are easy-to-remember identifiers chosen by the user and used for identifying themselves when logging into a system or service. The terms User ID and username might be used interchangeably if the username chosen by the user also serves as their unique identifier within the system.

Users should be permitted to use their email address as a username, provided the email is verified during sign-up. Additionally, they should have the option to choose a username other than an email address. For information on validating email addresses, please visit the input validation cheat sheet email discussion.

Authentication Solution and Sensitive Accounts

- Do **NOT** allow login with sensitive accounts (i.e. accounts that can be used internally within the solution such as to a backend / middleware / database) to any front-end user interface
- Do **NOT** use the same authentication solution (e.g. IDP / AD) used internally for unsecured access (e.g., public access / DMZ)

Implement Proper Password Strength Controls

A key concern when using passwords for authentication is password strength. A "strong" password policy makes it difficult or even improbable for one to guess the password through either manual or automated means. The following characteristics define a strong password:

- Password Length
  - **Minimum** length for passwords should be enforced by the application.
    - If MFA is enabled passwords **shorter than 8 characters** are considered to be weak (NIST SP800-63B).
    - If MFA is not enabled passwords **shorter than 15 characters** are considered to be weak (NIST SP800-63B).
  - **Maximum** password length should be **at least 64 characters** to allow passphrases (NIST SP800-63B). Note that certain implementations of hashing algorithms may cause long password denial of service.
- Do not silently truncate passwords. The Password Storage Cheat Sheet provides further guidance on how to handle passwords that are longer than the maximum length.
- Allow usage of **all** characters including unicode and whitespace. There should be no password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters.
- Ensure credential rotation when a password leak occurs, at the time of compromise identification or when authenticator technology changes. Avoid requiring periodic password changes; instead, encourage users to pick strong passwords and enable Multifactor Authentication Cheat Sheet (MFA). According to NIST guidelines, verifiers should not mandate arbitrary password changes (e.g., periodically).

- Include a password strength meter to help users create a more complex password
  - zxcvbn-ts library can be used for this purpose.
  - Other language implementations of zxcvbn listed here; however check the age and maturity of each example before use.
- Block common and previously breached passwords
  - Pwned Passwords is a service where passwords can be checked against previously breached passwords. Details on the API are here.
  - Alternatively, you can download the Pwned Passwords database using this mechanism to host it yourself.
  - Other top password lists are available but there is no guarantee as to how updated they are:
    - Various password lists hosted by SecLists from Daniel Miessler.
    - Static copy of the top 100,000 passwords from "Have I Been Pwned" hosted by NCSC in text and JSON format.

For more detailed information check

- ASVS v5.0 Password Security Requirements
- Passwords Evolved: Authentication Guidance for the Modern Era

Implement Secure Password Recovery Mechanism

It is common for an application to have a mechanism that provides a means for a user to gain access to their account in the event they forget their password. Please see Forgot Password Cheat Sheet for details on this feature.

Store Passwords in a Secure Fashion

It is critical for an application to store a password using the right cryptographic technique. Please see Password Storage Cheat Sheet for details on this feature.

Compare Password Hashes Using Safe Functions

Where possible, the user-supplied password should be compared to the stored password hash using a secure password comparison function provided by the language or framework, such as the password_verify() function in PHP. Where this is not possible, ensure that the comparison function:

- Has a maximum input length, to protect against denial of service attacks with very long inputs.
- Explicitly sets the type of both variables, to protect against type confusion attacks such as Magic Hashes in PHP.
- Returns in constant time, to protect against timing attacks.

Change Password Feature

When developing a change password feature, ensure to have:

- The user is authenticated with an active session.
- Current password verification. This is to ensure that it's the legitimate user who is changing the password. Consider this abuse case: a user logs in on a public computer and forgets to log out. Another person could then use that active session. If we don't verify the current password, this other person may be able to change the password.

Transmit Passwords Only Over TLS or Other Strong Transport

See: Transport Layer Security Cheat Sheet

The login page and all subsequent authenticated pages must be exclusively accessed over TLS or other strong transport. Failure to utilize TLS or other strong transport for the login page allows an attacker to modify the login form action, causing the user's credentials to be posted to an arbitrary location. Failure to utilize TLS or other strong transport for authenticated pages after login enables an attacker to view the unencrypted session ID and compromise the user's authenticated session.

Require Re-authentication for Sensitive Features

In order to mitigate CSRF and session hijacking, it's important to require the current credentials for an account before updating sensitive account information such as the user's password or email address -- or before sensitive transactions, such as shipping a purchase to a new address. Without this countermeasure, an attacker may be able to execute sensitive transactions through a CSRF or XSS attack without needing to know the user's current credentials. Additionally, an attacker may get temporary physical access to a user's browser or steal their session ID to take over the user's session.

Reauthentication After Risk Events

Overview: Reauthentication is critical when an account has experienced high-risk activity such as account recovery, password resets, or suspicious behavior patterns. This section outlines when

and how to trigger reauthentication to protect users and prevent unauthorized access. For further details, see the Require Re-authentication for Sensitive Features section.

When to Trigger Reauthentication

- **Suspicious Account Activity** When unusual login patterns, IP address changes, or device enrollments occur
- **Account Recovery** After users reset their passwords or change sensitive account details
- **Critical Actions** For high-risk actions like changing payment details or adding new trusted devices

Reauthentication Mechanisms

- **Adaptive Authentication** Use risk-based authentication models that adapt to the user's behavior and context
- **Multi-Factor Authentication (MFA)** Require an additional layer of verification for sensitive actions or events
- **Challenge-Based Verification** Prompt users to confirm their identity with a challenge question or secondary method

Implementation Recommendations

- **Minimize User Friction** Ensure that reauthentication does not disrupt the user experience unnecessarily
- **Context-Aware Decisions** Make reauthentication decisions based on context (e.g., geolocation, device type, prior patterns)
- **Secure Session Management** Invalidate sessions after reauthentication and rotate tokens—see the OWASP Session Management Cheat Sheet

References

- OWASP Session Management Cheat Sheet
- OWASP ASVS – 2.2.2: Reauthentication requirements
- NIST 800-63B: Digital Identity Guidelines – Authentication Assurance Levels

Consider Strong Transaction Authentication

Some applications should use a second factor to check whether a user may perform sensitive operations. For more information, see the Transaction Authorization Cheat Sheet.

TLS Client Authentication

TLS Client Authentication, also known as two-way TLS authentication, consists of both browser and server sending their respective TLS certificates during the TLS handshake process. Just as you can validate the authenticity of a server by using the certificate and asking a verifiably-valid Certificate Authority (CA) if the certificate is valid, the server can authenticate the user by receiving a certificate from the client and validating against a third-party CA or its own CA. To do this, the server must provide the user with a certificate generated specifically for him, assigning values to the subject so that these can be used to determine what user the certificate should validate. The user installs the certificate on a browser and now uses it for the website.

This approach is appropriate when:

- It is acceptable (or even preferred) that the user has access to the website only from a single computer/browser.
- The user is not easily scared by the process of installing TLS certificates on their browser, or there will be someone, probably from IT support, who will do this for the user.
- The website requires an extra step of security.
- It is also a good thing to use when the website is for an intranet of a company or organization.

It is generally not a good idea to use this method for widely and publicly available websites that will have an average user. For example, it wouldn't be a good idea to implement this for a website like Facebook. While this technique can prevent the user from having to type a password (thus protecting against an average keylogger from stealing it), it is still considered a good idea to consider using both a password and TLS client authentication combined.

Additionally, if the client is behind an enterprise proxy that performs SSL/TLS decryption, this will break certificate authentication unless the site is allowed on the proxy.

For more information, see: Client-authenticated TLS handshake

Authentication and Error Messages

Incorrectly implemented error messages in the case of authentication functionality can be used for the purposes of user ID and password enumeration. An application should respond (both HTTP and HTML) in a generic manner.

Authentication Responses

Using any of the authentication mechanisms (login, password reset, or password recovery), an application must respond with a generic error message regardless of whether:

- The user ID or password was incorrect.
- The account does not exist.
- The account is locked or disabled.

The account registration feature should also be taken into consideration, and the same approach of a generic error message can be applied regarding the case in which the user exists.

The objective is to prevent the creation of a discrepancy factor, allowing an attacker to mount a user enumeration action against the application.

It is interesting to note that the business logic itself can bring a discrepancy factor related to the processing time taken. Indeed, depending on the implementation, the processing time can be significantly different according to the case (success vs failure) allowing an attacker to mount a time-based attack (delta of some seconds for example).

Example using pseudo-code for a login feature:

- First implementation using the "quick exit" approach

```
IF USER_EXISTS(username) THEN
    password_hash=HASH(password)
    IS_VALID=LOOKUP_CREDENTIALS_IN_STORE(username, password_hash)
    IF NOT IS_VALID THEN
        RETURN Error("Invalid Username or Password!")
    ENDIF
ELSE
   RETURN Error("Invalid Username or Password!")
ENDIF
```

It can be clearly seen that if the user doesn't exist, the application will directly throw an error. Otherwise, when the user exists and the password doesn't, it is apparent that there will be more processing before the application errors out. In return, the response time will be different for the same error, allowing the attacker to differentiate between a wrong username and a wrong password.

- Second implementation without relying on the "quick exit" approach:

```
password_hash=HASH(password)
IS_VALID=LOOKUP_CREDENTIALS_IN_STORE(username, password_hash)
IF NOT IS_VALID THEN
   RETURN Error("Invalid Username or Password!")
ENDIF
```

This code will go through the same process no matter what the user or the password is, allowing the application to return in approximately the same response time.

The problem with returning a generic error message for the user is a User Experience (UX) matter. A legitimate user might feel confused with the generic messages, thus making it hard for them to use the application, and might after several retries, leave the application because of its complexity. The decision to return a *generic error message* can be determined based on the criticality of the application and its data. For example, for critical applications, the team can decide that under the failure scenario, a user will always be redirected to the support page and a *generic error message* will be returned.

Regarding the user enumeration itself, protection against brute-force attacks is also effective because it prevents an attacker from applying the enumeration at scale. Usage of CAPTCHA can be applied to a feature for which a *generic error message* cannot be returned because the *user experience* must be preserved.

INCORRECT AND CORRECT RESPONSE EXAMPLES

Login

Incorrect response examples:

- "Login for User foo: invalid password."
- "Login failed, invalid user ID."
- "Login failed; account disabled."
- "Login failed; this user is not active."

Correct response example:

- "Login failed; Invalid user ID or password."

Password recovery

Incorrect response examples:

- "We just sent you a password reset link."
- "This email address doesn't exist in our database."

Correct response example:

- "If that email address is in our database, we will send you an email to reset your password."

Account creation

Incorrect response examples:

- "This user ID is already in use."
- "Welcome! You have signed up successfully."

Correct response example:

- "A link to activate your account has been emailed to the address provided."

ERROR CODES AND URLS

The application may return a different HTTP Error code depending on the authentication attempt response. It may respond with a 200 for a positive result and a 403 for a negative result. Even though a generic error page is shown to a user, the HTTP response code may differ which can leak information about whether the account is valid or not.

Error disclosure can also be used as a discrepancy factor, consult the error handling cheat sheet regarding the global handling of different errors in an application.

Protect Against Automated Attacks

There are a number of different types of automated attacks that attackers can use to try and compromise user accounts. The most common types are listed below:

Attack Type	Description
Brute Force	Testing multiple passwords from a dictionary or other source against a single account.
Credential Stuffing	Testing username/password pairs obtained from the breach of another site.
Password Spraying	Testing a single weak password against a large number of different accounts.

Different protection mechanisms can be implemented to protect against these attacks. In many cases, these defenses do not provide complete protection, but when a number of them are implemented in a defense-in-depth approach, a reasonable level of protection can be achieved.

The following sections will focus primarily on preventing brute-force attacks, although these controls can also be effective against other types of attacks. For further guidance on defending against credential stuffing and password spraying, see the Credential Stuffing Cheat Sheet.

Multi-Factor Authentication

Multi-factor authentication (MFA) is by far the best defense against the majority of password-related attacks, including brute-force attacks, with analysis by Microsoft suggesting that it would have stopped 99.9% of account compromises. As such, it should be implemented wherever possible; however, depending on the audience of the application, it may not be practical or feasible to enforce the use of MFA.

The Multifactor Authentication Cheat Sheet contains further guidance on implementing MFA.

Login Throttling

Login Throttling is a protocol used to prevent an attacker from making too many attempts at guessing a password through normal interactive means, it includes the following controls:

- Maximum number of attempts.

ACCOUNT LOCKOUT

The most common protection against these attacks is to implement account lockout, which prevents any more login attempts for a period after a certain number of failed logins.

The counter of failed logins should be associated with the account itself, rather than the source IP address, in order to prevent an attacker from making login attempts from a large number of different IP addresses. There are a number of different factors that should be considered when implementing an account lockout policy in order to find a balance between security and usability:

- The number of failed attempts before the account is locked out (lockout threshold).
- The time period that these attempts must occur within (observation window).
- How long the account is locked out for (lockout duration).

Rather than implementing a fixed lockout duration (e.g., ten minutes), some applications use an exponential lockout, where the lockout duration starts as a very short period (e.g., one second), but doubles after each failed login attempt.

- Amount of time to delay after each account lockout (max 2-3, after that permanent account lockout).

When designing an account lockout system, care must be taken to prevent it from being used to cause a denial of service by locking out other users' accounts. One way this could be performed is to allow the use of the forgotten password functionality to log in, even if the account is locked out.

CAPTCHA

The use of an effective CAPTCHA can help to prevent automated login attempts against accounts. However, many CAPTCHA implementations have weaknesses that allow them to be solved using automated techniques or can be outsourced to services that can solve them. As such, the use of CAPTCHA should be viewed as a defense-in-depth control to make brute-force attacks more time-consuming and expensive, rather than as a preventative.

It may be more user-friendly to only require a CAPTCHA be solved after a small number of failed login attempts, rather than requiring it from the very first login.

Security Questions and Memorable Words

The addition of a security question or memorable word can also help protect against automated attacks, especially when the user is asked to enter a number of randomly chosen characters from the word. It should be noted that this does **not** constitute multi-factor authentication, as both factors are the same (something you know). Furthermore, security questions are often weak and have predictable answers, so they must be carefully chosen. The Choosing and Using Security Questions cheat sheet contains further guidance on this.

Logging and Monitoring

Enable logging and monitoring of authentication functions to detect attacks/failures on a real-time basis

- Ensure that all failures are logged and reviewed
- Ensure that all password failures are logged and reviewed
- Ensure that all account lockouts are logged and reviewed

Use of authentication protocols that require no password

While authentication through a combination of username, password, and multi-factor authentication is considered generally secure, there are use cases where it isn't considered the best option or even safe. Examples of this are third-party applications that desire to connect to the web application, either from a mobile device, another website, desktop, or other situations. When this happens, it is NOT considered safe to allow the third-party application to store the user/password combo, since then it extends the attack surface into their hands, where it isn't in your control. For this and other use cases, there are several authentication protocols that can protect you from exposing your users' data to attackers.

OAuth 2.0 and 2.1

OAuth is an **authorization** framework for delegated access to APIs. See also: OAuth 2.0 Cheat Sheet.

> **Note:** OAuth 2.1 is an IETF Working Group draft that consolidates OAuth 2.0 and widely adopted best practices and is intended to replace RFC 6749/6750; guidance in this cheat sheet applies to both OAuth 2.0 and OAuth 2.1. References: draft-ietf-oauth-v2-1-13, oauth.net/2.1

OpenID Connect (OIDC)

OpenID Connect 1.0 (OIDC) is an identity layer **on top of OAuth**. It defines how a client (**relying party**) verifies the **end user's** identity using an **ID Token** (a signed JWT) and how to obtain user claims in an interoperable way. Use **OIDC for authentication/SSO**; use **OAuth for authorization** to APIs.

OIDC implementation guidance

- **Validate ID Tokens** on the relying party: issuer (`iss`), audience (`aud`), signature (per provider JWKs), expiration (`exp`).
- Prefer **well-maintained libraries/SDKs** and provider discovery/JWKS endpoints.
- Use the **UserInfo** endpoint when additional claims beyond the ID Token are required.

> **Avoid confusion: OpenID 2.0 ("OpenID")** was a separate, legacy authentication protocol that has been **superseded by OpenID Connect** and is considered obsolete. New systems should not implement OpenID 2.0. References: OpenID Foundation — obsolete OpenID 2.0 libraries, OpenID 2.0 → OIDC migration

SAML

Security Assertion Markup Language (SAML) is often considered to compete with OpenId. The most recommended version is 2.0 since it is very feature-complete and provides strong security. Like OpenId, SAML uses identity providers, but unlike OpenId, it is XML-based and provides more flexibility. SAML is based on browser redirects which send XML data. Furthermore, SAML isn't only initiated by a service provider; it can also be initiated from the identity provider. This allows the user to navigate through different portals while still being authenticated without having to do anything, making the process transparent.

While OpenId has taken most of the consumer market, SAML is often the choice for enterprise applications because there are few OpenId identity providers which are considered enterprise-class (meaning that the way they validate the user identity doesn't have high standards required for enterprise identity). It is more common to see SAML being used inside of intranet websites, sometimes even using a server from the intranet as the identity provider.

In the past few years, applications like SAP ERP and SharePoint (SharePoint by using Active Directory Federation Services 2.0) have decided to use SAML 2.0 authentication as an often preferred method for single sign-on implementations whenever enterprise federation is required for web services and web applications.

See also: SAML Security Cheat Sheet

FIDO

The Fast Identity Online (FIDO) Alliance has created two protocols to facilitate online authentication: the Universal Authentication Framework (UAF) protocol and the Universal Second Factor (U2F) protocol. While UAF focuses on passwordless authentication, U2F allows the addition of a second factor to existing password-based authentication. Both protocols are based on a public key cryptography challenge-response model.

UAF takes advantage of existing security technologies present on devices for authentication including fingerprint sensors, cameras (face biometrics), microphones (voice biometrics), Trusted Execution Environments (TEEs), Secure Elements (SEs), and others. The protocol is designed to plug these device capabilities into a common authentication framework. UAF works with both native applications and web applications.

U2F augments password-based authentication using a hardware token (typically USB) that stores cryptographic authentication keys and uses them for signing. The user can use the same token as a second factor for multiple applications. U2F works with web applications. It provides **protection against phishing** by using the URL of the website to look up the stored authentication key.

Password Managers

Password managers are programs, browser plugins, or web services that automate the management of a large quantity of different credentials. Most password managers have functionality to allow users to easily use them on websites, either: (a) by pasting the passwords into the login form -- or -- (b) by simulating the user typing them in.

Web applications should not make the job of password managers more difficult than necessary by observing the following recommendations:

- Use standard HTML forms for username and password input with appropriate `type` attributes.
- Avoid plugin-based login pages (such as Flash or Silverlight).

- Implement a reasonable maximum password length, at least 64 characters, as discussed in the Implement Proper Password Strength Controls section.
- Allow any printable characters to be used in passwords.
- Allow users to paste into the username, password, and MFA fields.
- Allow users to navigate between the username and password field with a single press of the `Tab` key.

Changing A User's Registered Email Address

User email addresses often change. The following process is recommended to handle such situations in a system:

Note: The process is less stringent with Multifactor Authentication, as proof-of-identity is stronger than relying solely on a password.

Recommended Process If the User HAS Multifactor Authentication Enabled

1. Confirm the validity of the user's authentication cookie/token. If not valid, display a login screen.
2. Describe the process for changing the registered email address to the user.
3. Ask the user to submit a proposed new email address, ensuring it complies with system rules.
4. Request the use of Multifactor Authentication for identity verification.
5. Store the proposed new email address as a pending change.
6. Create and store **two** time-limited nonces for (a) system administrators' notification, and (b) user confirmation.
7. Send two email messages with links that include those nonces:
    - A **notification-only email message** to the current address, alerting the user to the impending change and providing a link to report unexpected activity.
    - A **confirmation-required email message** to the proposed new address, instructing the user to confirm the change and providing a link for unexpected situations.
8. Handle responses from the links accordingly.

Recommended Process If the User DOES NOT HAVE Multifactor Authentication Enabled

1. Confirm the validity of the user's authentication cookie/token. If not valid, display a login screen.
2. Describe the process for changing the registered email address to the user.
3. Ask the user to submit a proposed new email address, ensuring it complies with system rules.
4. Request the user's current password for identity verification.
5. Store the proposed new email address as a pending change.
6. Create and store three time-limited nonces for system administrators' notification, user confirmation, and an additional step for password reliance.
7. Send two email messages with links to those nonces:
    - A **confirmation-required email message** to the current address, instructing the user to confirm the change and providing a link for an unexpected situation.
    - A **separate confirmation-required email message** to the proposed new address, instructing the user to confirm the change and providing a link for unexpected situations.
8. Handle responses from the links accordingly.

Notes on the Above Processes

- It's worth noting that Google adopts a different approach with accounts secured only by a password -- where the current email address receives a notification-only email. This method carries risks and requires user vigilance.
- Regular social engineering training is crucial. System administrators and help desk staff should be trained to follow the prescribed process and recognize and respond to social engineering attacks. Refer to CISA's "Avoiding Social Engineering and Phishing Attacks" for guidance.

Adaptive or Risk Based Authentication

A feature of more advanced applications is the ability to require different authentication stages depending on various environmental and contextual attributes (including but not limited to, the sensitivity of the data for which access is being requested, time of day, user location, IP address, or device fingerprint).

For example, an application may require MFA for the first login from a particular device but not for subsequent logins from that device. Alternatively, a single sign-on solution may authenticate the

user and allow them to remain logged in for a day but require a reauthentication if they try to access their profile page.

Another option is the opposite approach where an application allows low risk access with just something that identifies the device (e.g., a specific mobile device fingerprint, a persistent cookie and browser fingerprint, etc. from the previous IP address) and then gradually requires stronger authentication for more sensitive operations. An example might be to allow someone to trigger something to see their current bank balance, but not the account number or anything else. If they need to see transactions, then the application puts them through some base level authentication and if they want to do any money movement, then MFA is required.

Questions that should be considered when implementing a mechanism like this include:

- Are the policies being put in place in line with any corporate policies and especially any regulatory policy?
- Which user- or device-attributes (IP, geolocation, device fingerprint, time-of-day, behavioral biometrics, etc.) will we monitor at session start?
- Which of those signals need to be refreshed during an active session, and at what cadence?
- How will we ensure each signal's accuracy and handle missing or low-confidence data?
- What scoring model (weights, thresholds, ML, rule-based, hybrid) will convert raw signals into a risk tier?
- Where will the model run (edge, API gateway, central service), and what is our latency budget?
- What action maps to each risk tier (allow, CAPTCHA, step-up MFA, block, revoke session)?
- What user-facing messages and error codes will accompany each action?
- At which exact code or platform layers will we invoke the risk engine (login controller, middleware, API gateway, service mesh)?
- How do we propagate decisions consistently across web, mobile, and API clients?
- How do we mutate, extend, or revoke tokens/cookies when a mid-session risk check escalates?
- How do we synchronize state across multiple concurrent devices or browser tabs?
- What monitoring and alerting will be in place for potentially suspicious activity, including how the user is notified.