Final Security Good Practices

Mobile Platform	Hybrid Application
Application domain type	m-Health
Authentication	Yes
Authentication schemes	Factors-based authentication ; ID-based authentication
Has DB	Yes
Type of database	SQL (Relational Database)
Which DB	SQLite
Type of information handled	Critical Data
User Registration	Yes
Type of Registration	Will be an administrator that will register the users
Programming Languages	HTML5
Input Forms	Yes
Upload Files	Yes
The system has logs	Yes
The system has regular updates	Yes
The system has third-party	Yes
System Cloud Environments	Public Cloud
Hardware Specification	Yes
HW Authentication	Basic Authentication (user/pass)
HW Wireless Tech	3G ; 4G/LTE ; 5G ; Wi-Fi ; GPS ; NFC ; Bluetooth
Data Center Phisical Access	Yes

Authentication

Authentication is the process of verifying that an individual, entity or website is whom it claims to be. Authentication in the context of web applications is commonly performed by submitting a username or ID and one or more items of private information that only a given user should know.

Authentication General Guidelines

User IDs

- Make sure your usernames/user IDs are case-insensitive (e.g. User 'smith' and user 'Smith' should be the same user);
- Usernames should also be unique;
- For high-security applications, usernames could be assigned and secret instead of user-defined public data.

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 back-end / middle-ware / DB) 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

- Password Length
  - Minimum password length (10 characters) should be enforced;
  - Maximum password length should not be too short because it will prevent users from creating passphrases;
  - The typical maximum length is 128 characters.
- Do not silently truncate passwords.
- Allow usage of **all** characters including unicode and whitespace.
- Ensure credential rotation when a password leak occurs, or at the time of compromise identification.
- Include password strength meter to help users create a more complex password and block common and previously breached passwords
  - zxcvbn-ts library can be used for this purpose;
  - Pwned Passwords is a service where passwords can be checked against previously breached passwords. You can host it yourself or use the API.

Implement Secure Password Recovery Mechanism

- Please check Forgot Password Cheat Sheet for details on this feature.

Store Passwords in a Secure Fashion

- Please see Password Storage Cheat Sheet for details on this feature.

Compare Password Hashes Using Safe Functions

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 variable, 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 change password feature, ensure to have:

- User is authenticated with active session.
- Current password verification.

Transmit Passwords Only Over TLSv1.3 or Other Strong Transport

Check: Transport Layer Protection Cheat Sheet

Require Re-authentication for Sensitive Features

Require the current credentials for an account before updating sensitive account information such as the user's password, user's email, or before sensitive transactions, such as shipping a purchase to a new address to avoid CSRF or XSS;

Consider Strong Transaction Authentication

The application must use a second authentication factor in financial transactions.

TLS Client Authentication

It is a good idea to do this when:

- It is acceptable (or even preferred) that the user only has access to the website from only a single computer/browser;
- The user is not easily scared by the process of installing TLS certificates on his browser, or there will be someone, probably from IT support, that 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 (e.g. Facebook)

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.

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."

Protect Against Automated Attacks

Countermeasures against automated attacks on authentication stand out:

- Multi-Factor Authentication (MFA);
- Account Lockout;
- CAPTCHA;
- Security Questions and Memorable Words.

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

OAuth * Implement OAuth 1.0a or OAuth 2.0 since the very first version (OAuth1.0) has been found to be vulnerable to session. fixation.

OpenId;

3. SAML * Implement version 2.0 since it is very feature-complete and provides strong security.
4. FIDO.

Password Managers

Web applications should not make password managers' job 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, such as 64 characters, as discussed in the Password Storage Cheat Sheet.
- Allow any printable characters to be used in passwords.
- Allow users to paste into the username and password fields.
- Allow users to navigate between the username and password field with a single press of the `Tab` key.

References

1. [Authentication Cheat Sheet] (https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html).

Multi-Factor Authentication

Introduction

Multi-Factor authentication (MFA), or Two-Factor Authentication (2FA) is when a user is required to present more than one type of evidence in order to authenticate on a system. There are four different types of evidence (or factors) that can be used, listed in the table below:

Factor	Examples

Something You Know	Passwords, PINs and security questions.
Something You Have	Hardware or software tokens, certificates, email, SMS and phone calls.
Something You Are	Fingerprints, facial recognition, iris scans and handprint scans.
Location	Source IP ranges and geolocation

It should be emphasised that while requiring multiple examples of a single factor (such as needing both a password and a PIN) **does not constitute MFA**, although it may provide some security benefits over a simple password.

Additionally, while the following sections discuss the disadvantage and weaknesses of various different types of MFA, in many cases these are only relevant against targeted attacks. **Any MFA is better than no MFA**.

Advantages

The most common way that user accounts get compromised on applications is through weak, re-used or stolen passwords. Despite any technical security controls implemented on the application, users are liable to choose weak passwords, or to use the same password on different applications. As developers or system administrators, it should be assumed that users' passwords will be compromised at some point, and the system should be designed in order to defend against this.

Multi-factor authentication (MFA) is by far the best defense against the majority of password-related attacks, including brute-force, credential stuffing and password spraying, with analysis by Microsoft suggesting that it would have stopped 99.9% of account compromises.

Disadvantages

The biggest disadvantage of MFA is the increase in management complexity for both administrators and end users. Many less technical users may find it difficult to configure and use MFA. Additionally, there are a number of other common issues encountered:

- Types of MFA that require users to have specific hardware can introduce significant costs and administrative overheads.
- Users may become locked out of their accounts if they lose or are unable to use their other factors.
- MFA introduces additional complexity into the application.
- Many MFA solutions add external dependencies to systems, which can introduce security vulnerabilities or single points of failure.
- Processes implemented to allow users to bypass or reset MFA may be exploitable by attackers.
- Requiring MFA may prevent some users from accessing the application.

Quick Recommendations

Exactly when and how MFA is implemented in an application will vary on a number of different factors, including the threat model of the application, the technical level of the users, and the level of administrative control over the users. These need to be considered on a per-application basis.

However, the following recommendations are generally appropriate for most applications, and provide an initial starting point to consider.

- Provide the option for users to enable MFA on their accounts using TOTP.
- Require MFA for administrative or other high privileged users.
- Consider allowing corporate IP ranges so that MFA is not required from them.
- Allow the user to remember the use of MFA in their browser, so they are not prompted every time they login.
- Implement a secure process to allow users to reset their MFA.

Implementing MFA

When to Require MFA

The most important place to require MFA on an application is when the user logs in. However, depending on the functionality available, it may also be appropriate to require MFA for performing sensitive actions, such as:

- Changing passwords or security questions.
- Changing the email address associated with the account.
- Disabling MFA.
- Elevating a user session to an administrative session.

If the application provides multiple ways for a user to authenticate these should all require MFA, or have other protections implemented. A common area that is missed is if the application provides a separate API that can be used to login, or has an associated mobile application.

Improving Usability

Having to frequently login with MFA creates an additional burden for users, and may cause them to disable MFA on the application. A number of mechanisms can be used to try and reduce the level of annoyance that MFA causes. However, these types of measures do decrease the security provided by MFA, so need to be risk assessed to find a reasonable balance of security and usability for the application.

- Remembering the user's browser so they don't need to use MFA every time.

    - This can either be permanent, or for a period of a few days.
    - This needs to be done with more than just a cookie, which could be stolen by an attacker.
        - For example, a cookie matched to the previous IP address the cookie was issued for.
- Allow corporate IP ranges (or, more strictly, using location as a second factor).
    - This doesn't protect against malicious insiders, or a user's workstation being compromised.
- Only requiring MFA for sensitive actions, not for the initial login.
    - This will depend heavily on the functionality in the application.

Failed Login Attempts

When a user enters their password, but fails to authenticate using a second factor, this could mean one of two things:

- The user has lost their second factor, or doesn't have it available (for example, they don't have their mobile phone, or have no signal).
- The user's password has been compromised.

There are a number of steps that should be taken when this occurs:

- Prompt the user to try another form of MFA
    - For example, an SMS code rather than using their hardware OTP token.
- Allow the user to attempt to reset their MFA.
- Notify the user of the failed login attempt, and encourage them to change their password if they don't recognize it.
    - The notification should include the time, browser and geographic location of the login attempt.
    - This should be displayed next time they login, and optionally emailed to them as well.

Resetting MFA

One of the biggest challenges with implementing MFA is handling users who forget or lose their second factors. There are many ways this could happen, such as:

- Re-installing a workstation without backing up digital certificates.
- Wiping or losing a phone without backing up OTP codes.
- Changing mobile numbers.

In order to prevent users from being locked out of the application, there needs to be a mechanism for them to regain access to their account if they can't use their existing MFA; however it is also crucial that this doesn't provide an attacker with a way to bypass MFA and hijack their account.

There is no definitive "best way" to do this, and what is appropriate will vary hugely based on the security of the application, and also the level of control over the users. Solutions that work for a corporate application where all the staff know each other are unlikely to be feasible for a publicly available application with thousands of users all over the world. Every recovery method has its own advantages and disadvantages, and these need to be evaluated in the context of the application.

Some suggestions of possible methods include:

- Providing the user with a number of single-use recovery codes when they first setup MFA.
- Requiring the user to setup multiple types of MFA (such as a digital certificate, OTP core and phone number for SMS), so that they are unlikely to lose access to all of them at once.
- Posting a one-use recovery code (or new hardware token) to the user.
- Requiring the user contact the support team and having a rigorous process in place to verify their identity.
- Requiring another trusted user to vouch for them.

Something You Know

The most common type of authentication is based on something the users knows - typically a password. The biggest advantage of this factor is that it has very low requirements for both the developers and the end user, as it does not require any special hardware, or integration with other services.

Passwords and PINs

Passwords and PINs are the most common form of authentication due to the simplicity of implementing them. The Authentication Cheat Sheet has guidance on how to implement a strong password policy, and the Password Storage Cheat Sheet has guidance on how to securely store passwords.

Most multi-factor authentication systems make use of a password, as well as at least one other factor.

It should be noted that PINs, "secret words" and other similar type of information are all effectively the same as passwords. Using two different types of passwords **does not constitute MFA**.

Pros

- Simple and well understood.
- Native support in every authentication framework.

- Easy to implement.

Cons

- Users are prone to choosing weak passwords.
- Passwords are commonly re-used between systems.
- Susceptible to phishing.

Security Questions

Security questions require the user to choose (or create) a number of questions that only they will know the answer to. These are effectively the same as passwords, although they are generally considered weaker. The Choosing and Using Security Questions Cheat Sheet contains further guidance on how to implement these securely.

Pros

- Simple and well understood.

Cons

- Questions often have easily guessable answers.
- Answers to questions can often be obtained from social media or other sources.
- Questions must be carefully chosen so that users will remember answers years later.
- Susceptible to phishing.

Something You Have

The second factor is something that the user possesses. This could be a physical item (such as a hardware token), a digital item (such as a certificate or private key), or based on the ownership of a mobile phone, phone number, or email address (such as SMS or a software token installed on the phone, or an email with a single-use verification code).

If properly implemented then this can be significantly more difficult for a remote attacker to compromise; however it also creates an additional administrative burden on the user, as they must keep the authentication factor with them whenever they wish to use it.

The requirement to have a second factor can also limit certain types of users' ability to access a service. For example, if a user does not have access to a mobile phone, many types of MFA will not be available for them.

Hardware OTP Tokens

Physical hardware OTP tokens can be used which generate constantly changing numeric codes, which must be submitted when authentication on the application. Most well-known of these is the RSA SecureID, which generates a six digit number that changes every 60 seconds.

Pros

- As the tokens are separate physical devices, they are almost impossible for an attacker to compromise remotely.
- Tokens can be used without requiring the user to have a mobile phone or other device.

Cons

- Deploying physical tokens to users is expensive and complicated.
- If a user loses their token it could take a significant amount of time to purchase and ship them a new one.
- Some implementations require a backend server, which can introduce new vulnerabilities as well as a single point of failure.
- Stolen tokens can be used without a PIN or device unlock code.
- Susceptible to phishing (although short-lived).

Software TOTP Tokens

A cheaper and easier alternative to hardware tokens is using software to generate Time-based One Time Password (TOTP) codes. This would typically involve the user installing a TOTP application on their mobile phone, and then scanning a QR code provided by the web application which provides the initial seed. The authenticator app then generates a six digit number every 60 seconds, in much the same way as a hardware token.

Most websites use standardized TOTP tokens, allowing the user to install any authenticator app that supports TOTP. However, a small number of applications use their own variants of this (such as Symantec), which requires the users to install a specific app in order to use the service. This should be avoided in favour of a standards-based approach.

Pros

- The absence of physical tokens greatly reduces the cost and administrative overhead of implementing the system.
- When users lose access to their TOTP app, a new one can be configured without needing to ship a physical token to them.

- TOTP is widely used, and many users will already have at least one TOTP app installed.
- As long as the user has a screen lock on their phone, an attacker will be unable to use the code if they steal the phone.

Cons

- TOTP apps are usually installed on mobile devices, which are vulnerable to compromise.
- The TOTP app may be installed on the same mobile device (or workstation) that is used to authenticate.
- Users may store the backup seeds insecurely.
- Not all users have mobile devices to use with TOTP.
- If the user's mobile device is lost, stolen or out of battery, they will be unable to authenticate.
- Susceptible to phishing (although short-lived).

Hardware U2F Tokens

Hardware U2F tokens communicate with the users workstation over USB or NFC, and implement challenge-response based authentication, rather than requiring the user to manually enter the code. This would typically be done by the user pressing a button on the token, or tapping it against their NFC reader.

Pros

- Longer codes can be used, which may provide a higher level of security.
- Users can simply press a button rather than typing in a code.
- Resistant to phishing.

Cons

- As with hardware OTP tokens, the use of physical tokens introduces significant costs and administrative overheads.
- Stolen tokens can be used without a PIN or device unlock code.
- As the tokens are usually connected to the workstation via USB, users are more likely to forget them.

Certificates

Digital certificates are files that are stored on the user's device which are automatically provided alongside the user's password when authenticating. The most common type is X.509 certificates (discussed in the Transport Layer Protection Cheat Sheet), more commonly known as client certificates.

Certificates are supported by all major web browsers, and once installed require no further interaction from the user. The certificates should be linked to an individual's user account in order to prevent users from trying to authenticate against other accounts.

Pros

- There is no need to purchase and manage hardware tokens.
- Once installed, certificates are very simple for users.
- Certificates can be centrally managed and revoked.
- Resistant to phishing.

Cons

- Using digital certificates requires backend PKI system.
- Installing certificates can be difficult for users, particularly in a highly restricted environment.
- Enterprise proxy servers which perform SSL decryption will prevent the use of certificates.
- The certificates are stored on the user's workstation, and as such can be stolen if their system is compromised.

Smartcards

Smartcards are credit-card size cards with a chip containing a digital certificate for the user, which is unlocked with a PIN. They are commonly used for operating system authentication, but are rarely used in web applications.

Pros

- Stolen smartcards cannot be used without the PIN.
- Smartcards can be used across multiple applications and systems.
- Resistant to phishing.

Cons

- Managing and distributing smartcards has the same costs and overheads as hardware tokens.
- Smartcards are not natively supported by modern browsers, so require third party software.
- Although most business-class laptops have smartcard readers built in, home systems often do not.
- The use of smartcards requires functioning backend PKI systems.

SMS Messages and Phone Calls

SMS messages or phone calls can be used to provide users with a single-use code that they must submit as a second factor.

Pros

- Relatively simple to implement.
- Requires user to link their account to a mobile number.

Cons

- Requires the user to have a mobile device or landline.
- Require user to have signal to receive the call or message.
- Calls and SMS messages may cost money to send (need to protect against attackers requesting a large number of messages to exhaust funds.
- A number of attacks against SMS or mobile numbers have been demonstrated and exploited in the past.
- SMS messages may be received on the same device the user is authenticating from.
- Susceptible to phishing.

Email

Email verification requires that the user enters a code or clicks a link sent to their email address. There is some debate as to whether email constitutes a form of MFA, because if the user does not have MFA configured on their email account, it simply requires knowledge of the user's email password (which is often the same as their application password). However, it is included here for completeness.

Pros

- Very easy to implement.
- No requirements for separate hardware or a mobile device.

Cons

- Relies entirely on the security of the email account, which often lacks MFA.
- Email passwords are commonly the same as application passwords.
- Provides no protection if the user's email is compromised first.
- Email may be received by the same device the user is authenticating from.
- Susceptible to phishing.

Something You Are

The final factor in the traditional view of MFA is something you are - which is one of the physical attributes of the users (often called biometrics). Biometrics are rarely used in web applications due to the requirement for users to have specific hardware.

Biometrics

The are a number of common types of biometrics that are used, including:

- Fingerprint scans
- Facial recognition
- Iris scans
- Handprint scans

Pros

- Well-implemented biometrics are hard to spoof, and require a targeted attack.

Cons

- Require manual enrolment of the user's physical attributes.
- Custom (sometimes expensive) hardware is often required to read biometrics.
- Modern browsers do not have native support, so custom client-side software is required.
- Privacy concerns: Sensitive physical information must be stored about users.
- If compromised, biometric data can be difficult to change.

Location

The use of location as a fourth factor for MFA is not fully accepted; however, it is increasingly be used for authentication. It is sometimes argued that location is used when deciding whether or not to require MFA (as discussed above) however this is effectively the same as considering it to be a factor in its own right. Two prominent examples of this are the Conditional Access Policies available in Microsoft Azure, and the Network Unlock functionality in BitLocker.

When talking about location, access to the application that the user is authenticating against is not usually considered (as this would always be the case, and as such is relatively meaningless).

Source IP Ranges

The source IP address the user is connecting from can be used as a factor, typically in an allow-list based approach. This could either be based on a static list (such as corporate office ranges) or a dynamic list (such as previous IP addresses the user has authenticated from).

Pros

- Very easy for users.
- Requires minimal configuration and management from administrative staff.

Cons

- Doesn't provide any protection if the user's system is compromised.
- Doesn't provide any protection against rogue insiders.
- Trusted IP addresses must be carefully restricted (for example, if the open guest Wi-Fi uses the main corporate IP range).

Geolocation

Rather than using the exact IP address of the user, the geographic location that the IP address is registered to can be used. This is less precise, but may be more feasible to implement in environments where IP addresses are not static. A common usage would be to require additional authentication factors when an authentication attempt is made from outside of the user's normal country.

Pros

- Very easy for users

Cons

- Doesn't provide any protection if the user's system is compromised.
- Doesn't provide any protection against rogue insiders.
- Easy for an attacker to bypass by obtaining IP addresses in the trusted country or location.

Authorization

Authorization may be defined as "the process of verifying that a requested action or service is approved for a specific entity" (NIST). Authorization is distinct from authentication which is the process of verifying an entity's identity. When designing and developing a software solution, it is important to keep these distinctions in mind. A user who has been authenticated (perhaps by providing a username and password) is often not authorized to access every resource and perform every action that is technically possible through a system. For example, a web app may have both regular users and admins, with the admins being able to perform actions the average user is not privileged to do so, even though they have been authenticated. Additionally, authentication is not always required for accessing resources; an unauthenticated user may be authorized to access certain public resources, such as an image or login page, or even an entire web app.

Authorization Guidelines

Enforce Least Privileges

Best practices:

- During the design phase, ensure trust boundaries are defined. Enumerate the types of users that will be accessing the system, the resources exposed and the operations (such as read, write, update, etc) that might be performed on those resources. For every combination of user type and resource, determine what operations, if any, the user (based on role and/or other attributes) must be able to perform on that resource. For an ABAC system ensure all categories of attributes are considered. For example, a Sales Representative may need to access a customer database from the internal network during working hours, but not from home at midnight.
- Create tests that validate that the permissions mapped out in the design phase are being correctly enforced.
- After the app has been deployed, periodically review permissions in the system for "privilege creep"; that is, ensure the privileges of users in the current environment do not exceed those defined during the design phase (plus or minus any formally approved changes).
- Remember, it is easier to grant users additional permissions rather than to take away some they previously enjoyed. Careful planning and implementation of Least Privileges early in the SDLC can help reduce the risk of needing to revoke permissions that are later deemed overly broad.

Deny by Default

Best practices:

- Adopt a "deny-by-default" mentality both during initial development and whenever new functionality or resources are exposed by the app. One should be able to explicitly justify why a specific permission was granted to a particular user or group rather than assuming access to be the default position.
- Although some frameworks or libraries may themselves adopt a deny-by-default strategy, explicit configuration should be preferred over relying on framework or library defaults. The logic and defaults of third-party code may evolve over time, without the developer's full knowledge or understanding of the change's

implications for a particular project.

Validate the Permissions on Every Request

Validating permissions correctly on just the majority of requests is insufficient. Use specific technologies to validate the permissions, such as:

- Java/Jakarta EE Filters including implementations in Spring Security
- Middleware in the Django Framework
- .NET Core Filters
- Middleware in the Laravel PHP Framework

Thoroughly Review the Authorization Logic of Chosen Tools and Technologies, Implementing Custom Logic if Necessary

Best Practices

- Create, maintain, and follow processes for detecting and responding to vulnerable components.
- Incorporate tools such as Dependency Check into the SDLC and consider subscribing to data feeds from vendors, the NVD, or other relevant sources.

  Implement defense in depth. Do not depend on any single framework, library, technology, or control to be the sole thing enforcing proper access control.

  Take time to thoroughly understand any technology you build authorization logic upon. Analyze the technologies capabilities with an understanding that *the authorization logic provided by the component may be insufficient for your application's specific security requirements*. Relying on prebuilt logic may be convenient, but this does not mean it is sufficient. Understand that custom authorization logic may well be necessary to meet an app's security requirements.

- Do not let the capabilities of any library, platform, or framework guide your authorization requirements. Rather, authorization requirements should be decided first and then the third-party components may be analyzed in light of these requirements.
- Do not rely on default configurations.
- Test configuration. Do not just assume any configuration performed on a third-party component will work exactly as intended in your particular environment. Documentation can be misunderstood, vague, outdated, or simply inaccurate.

Prefer Attribute and Relationship Based Access Control over RBAC

In software engineering, two basic forms of access control are widely utilized: Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC). There is a third, more recent, model which has gained popularity: Relationship-Based Access Control (ReBAC). The decision between the models has significant implications for the entire SDLC and should be made as early as possible.

Use ABAC and ReBAC access control for application development.

Ensure Lookup IDs are Not Accessible Even When Guessed or Cannot Be Tampered With

Applications often expose the internal object identifiers (such as an account number or Primary Key in a database) that are used to locate and reference an object. This ID may exposed as a query parameter, path variable, "hidden" form field or elsewhere. Recommended mitigations for this weakness include the following:

- Avoid exposing identifiers to the user when possible. For example it should be possible to retrieve some objects, such as account details, based solely on currently authenticated user's identity and attributes (e.g. through information contained in a securely implemented JSON Web Token (JWT) or server-side session).
- Implement user/session specific indirect references using a tool such as OWASP ESAPI (see OWASP 2013 Top 10 - A4 Insecure Direct Object References)
- Perform access control checks on *every* request for the *specific* object or functionality being accessed. Just because a user has access to an object of a particular type does not mean they should have access to every object of that particular type.

Enforce Authorization Checks on Static Resources

Best Practices:

- Ensure that static resources are incorporated into access control policies.
- Ensure any cloud based services used to store static resources are secured using the configuration options and tools provided by the vendor.
- When possible, protect static resources using the same access control logic and mechanisms that are used to secure other application resources and functionality.

Verify that Authorization Checks are Performed in the Right Location

Best Practices:

- Developers must never rely on client-side access control checks;
- Access control checks must be performed server-side, at the gateway, or using serverless function (see OWASP ASVS 4.0.3, V1.4.1 and V4.1.1);

Exit Safely when Authorization Checks Fail

Best Practices:

- Ensure all exception and failed access control checks are handled no matter how unlikely they seem (OWASP Top Ten Proactive Controls C10: Handle all errors and exceptions). This does not mean that an application should always try to "correct" for a failed check; oftentimes a simple message or HTTP status code is all that is required.
- Centralize the logic for handling failed access control checks.
- Verify the handling of exception and authorization failures. Ensure that such failures, no matter how unlikely, do not put the software into an unstable state that could lead to authorization bypass.

Implement Appropriate Logging

Best Practices:

- Log using consistent, well-defined formats that can be readily parsed for analysis.
- Carefully determine the amount of information to log.
- Ensure clocks and timezones are synchronized across systems. Accuracy is crucial in piecing together the sequence of an attack during and after incident response.
- Consider incorporating application logs into a centralized log server or SIEM.

References

ABAC

ABAC with Spring Security

NIST Special Publication 800-162 Guide to Attribute Based Access Control (ABAC) Definition and Considerations

NIST SP 800-178 A Comparison of Attribute Based Access Control (ABAC) Standards for Data Service Applications

NIST SP 800-205 Attribute Considerations for Access Control Systems

XACML-V3.0 for standard that highlights these benefits)

General

OWASP Application Security Verification Standard 4.0 (especially see V4: Access Control Verification Requirements)

OWASP Web Security Testing Guide - 4.5 Authorization Testing

- [Authorization Cheat Sheet] (https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html)

Least Privilege

- Least Privilege

RBAC

- Role-Based Access Controls.

ReBAC

- Relationship-Based Access Control (ReBAC)
- Google Zanzibar

Cross Site Scripting Prevention

Introduction

Cross-Site Scripting (XSS) is a misnomer. The name originated from early versions of the attack where stealing data cross-site was the primary focus. Since then, it has extended to include injection of basically any content, but we still refer to this as XSS. XSS is serious and can lead to account impersonation, observing user behaviour, loading external content, stealing sensitive data, and more.

Using the right combination of defensive techniques is necessary to prevent XSS.

Framework Security

Developers need to be aware of problems that can occur when using frameworks insecurely such as:

- *escape hatches* that frameworks use to directly manipulate the DOM;
- React's `dangerouslySetInnerHTML` without sanitising the HTML;
- React cannot handle `javascript:` or `data:` URLs without specialized validation;
- Angular's `bypassSecurityTrustAs*` functions;
- Template injection;
- Out of date framework plugins or components;
- and more.

XSS Defense Philosophy

- Frameworks make it easy to ensure variables are correctly validated and escaped or sanitised;
- Use of Output Encoding and HTML Sanitization to address security gaps that exist in popular frameworks like React and Angular.

Output Encoding

- Start with using your framework's default output encoding protection when you wish to display data as the user typed it in;
- If you're not using a framework or need to cover gaps in the framework then you should use an output encoding library;
- Each variable used in the user interface should be passed through an output encoding function.

There are many different output encoding methods because browsers parse HTML, JS, URLs, and CSS differently. Using the wrong encoding method may introduce weaknesses or harm the functionality of your application.

Best practices to Output Encoding: * Output Encoding for "HTML Contexts"; * Output Encoding for "HTML Attribute Contexts"; * Output Encoding for "JavaScript Contexts"; * Output Encoding for "CSS Contexts"; * Output Encoding for "URL Contexts" * Common Mistake.

Dangerous Contexts

Callback functions

- Where URLs are handled in code such as this CSS { background-url : "javascript:alert(xss)"; }
- All JavaScript event handlers (`onclick()`, `onerror()`, `onmouseover()`).
- Unsafe JS functions like `eval()`, `setInterval()`, `setTimeout()`

HTML Sanitization

Best practices:

- If you sanitize content and then modify it afterwards, you can easily void your security efforts.
- If you sanitize content and then send it to a library for use, check that it doesn't mutate that string somehow. Otherwise, again, your security efforts are void.
- You must regularly patch DOMPurify or other HTML Sanitization libraries that you use. Browsers change functionality and bypasses are being discovered regularly.

Safe Sinks

- Try to refactor your code to remove references to unsafe sinks like innerHTML, and instead use textContent or value;
- Check Safe HTML Attributes.

Other Controls

OWASP recommends in all circumstances to implement against XSS:

- Framework Security Protections;
- Output Encoding;
- HTML Sanitization.

Consider adopting the following controls in addition to the above.

- Cookie Attributes - These change how JavaScript and browsers can interact with cookies. Cookie attributes try to limit the impact of an XSS attack but don't prevent the execution of malicious content or address the root cause of the vulnerability.
- Content Security Policy - An allowlist that prevents content being loaded. It's easy to make mistakes with the implementation so it should not be your primary defense mechanism. Use a CSP as an additional layer of defense and have a look at the cheatsheet here.
- Web Application Firewalls - These look for known attack strings and block them. WAF's are unreliable and new bypass techniques are being discovered regularly. WAFs also don't address the root cause of an XSS vulnerability. In addition, WAFs also miss a class of XSS vulnerabilities that operate exclusively client-side. WAFs are not recommended for preventing XSS, especially DOM-Based XSS.

XSS Prevention Rules Summary

The following snippets of HTML demonstrate how to safely render untrusted data in a variety of different contexts.

Data Type	Context	Code Sample	Defense
String	HTML Body	`<span>UNTRUSTED DATA </span>`	HTML Entity Encoding (rule #1).
String	Safe HTML Attributes	`<input type="text" name="fname" value="UNTRUSTED DATA ">`	Aggressive HTML Entity Encoding (rule #2), Only place untrusted data into a list of safe attributes (listed below), Strictly validate unsafe attributes such as background, ID and name.
String	GET Parameter	`<a href="/site/search?value=UNTRUSTED DATA ">clickme</a>`	URL Encoding (rule #5).
String	Untrusted URL in a SRC or HREF attribute	`<a href="UNTRUSTED URL ">clickme</a> <iframe src="UNTRUSTED URL " />`	Canonicalize input, URL Validation, Safe URL verification, Allow-list http and HTTPS URLs only (Avoid the JavaScript Protocol to Open a new Window), Attribute encoder.
String	CSS Value	`HTML <div style="width: UNTRUSTED DATA ;">Selection</div>`	Strict structural validation (rule #4), CSS Hex encoding, Good design of CSS Features.
String	JavaScript Variable	`<script>var currentValue='UNTRUSTED DATA ';</script> <script>someFunction('UNTRUSTED DATA ');</script>`	Ensure JavaScript variables are quoted, JavaScript Hex Encoding, JavaScript Unicode Encoding, Avoid backslash encoding (`\"` or `\'` or `\\`).
HTML	HTML Body	`<div>UNTRUSTED HTML</div>`	HTML Validation (JSoup, AntiSamy, HTML Sanitizer...).
String	DOM XSS	`<script>document.write("UNTRUSTED INPUT: " + document.location.hash );<script/>`	DOM based XSS Prevention Cheat Sheet

Output Encoding Rules Summary

The purpose of output encoding (as it relates to Cross Site Scripting) is to convert untrusted input into a safe form where the input is displayed as **data** to the user without executing as **code** in the browser. The following charts details a list of critical output encoding methods needed to stop Cross Site Scripting.

Encoding Type	Encoding Mechanism
HTML Entity Encoding	Convert `&` to `&`, Convert < to `<`, Convert > to `>`, Convert " to `&quot;`, Convert ' to `&#x27;`, Convert `/` to `&#x2F;`
HTML Attribute Encoding	Except for alphanumeric characters, encode all characters with the HTML Entity `&#xHH;` format, including spaces. (**HH** = Hex Value)
URL Encoding	Standard percent encoding, see here. URL encoding should only be used to encode parameter values, not the entire URL or path fragments of a URL.
JavaScript Encoding	Except for alphanumeric characters, encode all characters with the `\uXXXX` unicode encoding format (**X** = Integer).
CSS Hex Encoding	CSS encoding supports `\XX` and `\XXXXXX`. Using a two character encode can cause problems if the next character continues the encode sequence. There are two solutions: (a) Add a space after the CSS encode (will be ignored by the CSS parser) (b) use the full amount of CSS encoding possible by zero padding the value.

Related Articles

XSS Attack Cheat Sheet:

The following article describes how to exploit different kinds of XSS Vulnerabilities that this article was created to help you avoid:

- OWASP: XSS Filter Evasion Cheat Sheet.

Description of XSS Vulnerabilities:

- OWASP article on XSS Vulnerabilities.

Discussion on the Types of XSS Vulnerabilities:

- Types of Cross-Site Scripting.

How to Review Code for Cross-site scripting Vulnerabilities:

- OWASP Code Review Guide article on Reviewing Code for Cross-site scripting Vulnerabilities.

How to Test for Cross-site scripting Vulnerabilities:

- OWASP Testing Guide article on testing for Cross-Site Scripting vulnerabilities.
- XSS Experimental Minimal Encoding Rules

Cross-Site Request Forgery Prevention

Introduction

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site when the user is authenticated. A CSRF attack works because browser requests automatically include all cookies including session cookies. Therefore, if the user is authenticated to the site, the site cannot distinguish between legitimate authorized requests and forged authenticated requests. This attack is thwarted when proper Authorization is used, which implies that a challenge-response mechanism is required that verifies the identity and authority of the requester.

In short, the following principles should be followed to defend against CSRF:

Check if your framework has built-in CSRF protection and use it

- **If framework does not have built-in CSRF protection add CSRF tokens to all state changing requests (requests that cause actions on the site) and validate them on backend**
- **For stateful software use the synchronizer token pattern**
- **For stateless software use double submit cookies**

Implement at least one mitigation from Defense in Depth Mitigations section

- **Consider SameSite Cookie Attribute for session cookies** but be careful to NOT set a cookie specifically for a domain as that would introduce a security vulnerability that all subdomains of that domain share the cookie. This is particularly an issue when a subdomain has a CNAME to domains not in your control.
- **Consider implementing user interaction based protection for highly sensitive operations**
- **Consider the use of custom request headers**
- **Consider verifying the origin with standard headers**

Remember that any Cross-Site Scripting (XSS) can be used to defeat all CSRF mitigation techniques!

- **See the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws.**

Do not use GET requests for state changing operations.

- **If for any reason you do it, protect those resources against CSRF**

References

CSRF

- OWASP Cross-Site Request Forgery (CSRF)
- PortSwigger Web Security Academy
- Mozilla Web Security Cheat Sheet
- Common CSRF Prevention Misconceptions
- Robust Defenses for Cross-Site Request Forgery
- For Java: OWASP CSRF Guard or Spring Security
- For PHP and Apache: CSRFProtector Project
- For AngularJS: Cross-Site Request Forgery (XSRF) Protection
- Cross-Site Request Forgery Prevention Cheat Sheet

Cryptographic Storage

Introduction

These guidelines provide a simple model to follow when implementing solutions to protect data at rest.

Passwords should not be stored using reversible encryption - secure password hashing algorithms should be used instead.

Architectural Design

The first step in designing any application is to consider the overall architecture of the system, as this will have a huge impact on the technical implementation.

- Considering the threat model of the application (i.e, who you trying to protect that data against);
- Use of dedicated secret or key management systems
- Making the management of secrets significantly easier.

Where to Perform Encryption

Encryption can be performed on a number of levels in the application stack, such as:

- At the application level;
- At the database level (e.g, SQL Server TDE);
- At the filesystem level (e.g, BitLocker or LUKS);
- At the hardware level (e.g, encrypted RAID cards or SSDs).

Minimise the Storage of Sensitive Information

- Wherever possible, the storage of sensitive information should be avoided.

Algorithms

- For symmetric encryption **AES** with a key that's at least **128 bits** (ideally **256 bits**) and a secure mode should be used as the preferred algorithm;
- For asymmetric encryption, use elliptical curve cryptography (ECC) with a secure curve such as **Curve25519** as a preferred algorithm;
- If ECC is not available and **RSA** must be used, then ensure that the key is at least **2048 bits**.

For an other many other symmetric and asymmetric algorithms, a number of factors should be taken into account, including:

- Key size.
- Known attacks and weaknesses of the algorithm;
- Maturity of the algorithm;
- Approval by third parties such as NIST's algorithmic validation program;
- Performance (both for encryption and decryption);
- Quality of the libraries available.
- Portability of the algorithm (i.e, how widely supported is it);

In some cases there may be regulatory requirements that limit the algorithms that can be used, such as FIPS 140-2 or PCI DSS.

Custom Algorithms

Don't do this.

Cipher Modes

GCM and **CCM**, cipher modes should be used as a first preference;

If GCM or CCM are not available, then CTR mode or CBC mode should be used in combination with separate authentication, such as using the Encrypt-then-MAC technique;

- If random access to the encrypted data is required then XTS mode should be used.

Random Padding

For RSA, it is essential to enable Random Padding also known as OAEP or Optimal Asymmetric Encryption Padding;

The Padding Schema of PKCS#1 is typically used in this case.

Secure Random Number Generation

The table below shows the recommended algorithms for each language, as well as insecure functions that should not be used.

Language	Unsafe Functions	Cryptographically Secure Functions
C	`random()`, `rand()`	getrandom(2)
Java	`java.util.Random()`	java.security.SecureRandom
PHP	`rand()`, `mt_rand()`, `array_rand()`, `uniqid()`	random_bytes(), random_int() in PHP 7 or openssl_random_pseudo_bytes() in PHP 5
.NET/C#	`Random()`	RNGCryptoServiceProvider
Objective-C	`arc4random()` (Uses RC4 Cipher)	SecRandomCopyBytes
Python	`random()`	secrets()
Ruby	`Random`	SecureRandom
Go	`rand` using `math/rand` package	crypto.rand package
Rust	`rand::prng::XorShiftRng`	rand::prng::chacha::ChaChaRng and the rest of the Rust library CSPRNGs.
Node.js	`Math.random()`	crypto.randomBytes, crypto.randomInt, crypto.randomUUID

Defence in Depth

- Applications should be designed to still be secure even if cryptographic controls fail;
- Application should also not rely on the security of encrypted URL parameters, and should enforce strong access control to prevent unauthorised access to information.

Key Management

Processes

Formal processes should be implemented (and tested) to cover all aspects of key management, including:

- Generating and storing new keys;
- Distributing keys to the required parties;
- Deploying keys to application servers;
- Rotating and decommissioning old keys.

Key Generation

- Keys should be randomly generated using a cryptographically secure function;
- Keys **should not** be based on common words or phrases, or on "random" characters generated by mashing the keyboard;
- Where multiple keys are used (such as data separate data-encrypting and key-encrypting keys), they should be fully independent from each other.

Key Lifetimes and Rotation

Encryption keys should be changed (or rotated) based on a number of different criteria:

- If the previous key is known (or suspected) to have been compromised.
  - This could also be caused by a someone who had access to the key leaving the organisation.
- After a specified period of time has elapsed (known as the cryptoperiod).
  - There are many factors that could affect what an appropriate cryptoperiod is, including the size of the key, the sensitivity of the data, and the threat model of the system. See section 5.3 of NIST SP 800-57 for further guidance.
- After the key has been used to encrypt a specific amount of data.
  - This would typically be 2^35 bytes (~34GB) for 64-bit keys and 2^68 bytes (~295 exabytes) for 128-bit block size.
- If there is a significant change to the security provided by the algorithm (such as a new attack being announced).

Once one of these criteria have been met, a new key should be generated and used for encrypting any new data. There are two main approaches for how existing data that was encrypted with the old key(s) should be handled:

1. Decrypting it and re-encrypting it with the new key.
2. Marking each item with the ID of the key that was used to encrypt it, and storing multiple keys to allow the old data to be decrypted.

The first option should generally be preferred, as it greatly simplifies both the application code and key management processes; however, it may not always be feasible. Note that old keys should generally be stored for a certain period after they have been retired, in case old backups of copies of the data need to be decrypted.

It is important that the code and processes required to rotate a key are in place **before** they are required, so that keys can be quickly rotated in the event of a compromise. Additionally, processes should also be implemented to allow the encryption algorithm or library to be changed, in case a new vulnerability is found in the algorithm or implementation.

Key Storage

Securely storing cryptographic keys is one of the hardest problems to solve, as the application always needs to have some level of access to the keys in order to decrypt the data. While it may not be possible to fully protect the keys from an attacker who has fully compromised the application, a number of steps can be taken to make it harder for them to obtain the keys.

Where available, the secure storage mechanisms provided by the operating system, framework or cloud service provider should be used. These include:

- A physical Hardware Security Module (HSM).
- A virtual HSM.
- Key vaults such as Amazon KMS or Azure Key Vault.
- Secure storage APIs provided by the ProtectedData class in the .NET framework.

There are many advantages to using these types of secure storage over simply putting keys in configuration files. The specifics of these will vary depending on the solution used, but they include:

- Central management of keys, especially in containerised environments.

- Easy key rotation and replacement.
- Secure key generation.
- Simplifying compliance with regulatory standards such as FIPS 140 or PCI DSS.
- Making it harder for an attacker to export or steal keys.

In some cases none of these will be available, such as in a shared hosting environment, meaning that it is not possible to obtain a high degree of protection for any encryption keys. However, the following basic rules can still be followed:

- Do not hard-code keys into the application source code.
- Do not check keys into version control systems.
- Protect the configuration files containing the keys with restrictive permissions.
- Avoid storing keys in environment variables, as these can be accidentally exposed through functions such as phpinfo() or through the `/proc/self/environ` file.

Separation of Keys and Data

- Where possible, encryption keys should be stored in a separate location from encrypted data.

Encrypting Stored Keys

Where possible, encryption keys should themselves be stored in an encrypted form. At least two separate keys are required for this:

- The Data Encryption Key (DEK) is used to encrypt the data.
- The Key Encryption Key (KEK) is used to encrypt the DEK.

For this to be effective, the KEK must be stored separately from the DEK. The encrypted DEK can be stored with the data, but will only be usable if an attacker is able to also obtain the KEK, which is stored on another system.

The KEK should also be at least as strong as the DEK. The envelope encryption guidance from Google contains further details on how to manage DEKs and KEKs.

In simpler application architectures (such as shared hosting environments) where the KEK and DEK cannot be stored separately, there is limited value to this approach, as an attacker is likely to be able to obtain both of the keys at the same time. However, it can provide an additional barrier to unskilled attackers.

A key derivation function (KDF) could be used to generate a KEK from user-supplied input (such a passphrase), which would then be used to encrypt a randomly generated DEK. This allows the KEK to be easily changed (when the user changes their passphrase), without needing to re-encrypt the data (as the DEK remains the same).

References

- Cryptographic Storage Cheat Sheet

Database Security

Introduction

This guides provides guidance on securely configuring and using the SQL and NoSQL databases. It is intended to be used by application developers when they are responsible for managing the databases, in the absence of a dedicated database administrator (DBA).

Connecting to the Database

The backend database used by the application should be isolated as much as possible, in order to prevent malicious or undesirable users from being able to connect to it. Exactly how this is achieved will depend on the system and network architecture. The following options could be used to protect it:

- Disabling network (TCP) access and requiring all access is over a local socket file or named pipe.
- Configuring the database to only bind on localhost.
- Restricting access to the network port to specific hosts with firewall rules.
- Placing the database server in a separate DMZ isolated from the application server.

Similar protection should be implemented to protect any web-based management tools used with the database, such as phpMyAdmin.

When an application is running on an untrusted system (such as a thick-client), it should always connect to the backend through an API that can enforce appropriate access control and restrictions. Direct connections should **never** be made from a thick client to the backend database.

Transport Layer Protection

The following steps should be taken to prevent unencrypted traffic:

- Configure the database to only allow encrypted connections.
- Install a trusted digital certificate on the server.
- Configure the client application to connect using TLSv1.2+ with modern ciphers (e.g, AES-GCM or ChaCha20).
- Configure the client application to verify that the digital certificate is correct.

Authentication

The database should be configured to always require authentication, including connections from the local server. Database accounts should be:

- Protected with strong and unique passwords.
- Used by a single application or service.
- Configured with the minimum permissions required as discussed in the permissions section below.

As with any system that has its own user accounts, the usual account management processes should be followed, including:

- Regular reviews of the accounts to ensure that they are still required.
- Regular reviews of permissions.
- Removing user accounts when an application is decommissioned.
- Changing the passwords when staff leave, or there is reason to believe that they may have been compromised.

For Microsoft SQL Server, consider the use of Windows or Integrated-Authentication, which uses existing Windows accounts rather than SQL Server accounts. This also removes the requirement to store credentials in the application, as it will connect using the credentials of the Windows user it is running under. The Windows Native Authentication Plugins provides similar functionality for MySQL.

Storing Database Credentials

Database credentials should never be stored in the application source code, especially if they are unencrypted. Instead, they should be stored in a configuration file that:

- Is outside of the webroot.
- Has appropriate permissions so that it can only be read by the required user(s).
- Is not checked into source code repositories.

Where possible, these credentials should also be encrypted or otherwise protected using built-in functionality, such as the `web.config` encryption available in ASP.NET.

Permissions

The permissions assigned to database user accounts should be based on the principle of least privilege (i.e, the accounts should only have the minimal permissions required for the application to function). This can be applied at a number of increasingly granular levels depending on the functionality available in the database. The following steps should be applicable to all environments:

- Do not use the built-in `root`, `sa` or SYS accounts.
- Do not grant the account administrative rights over the database instance.
- Only allow the account to connect from allowed hosts.
  - This would often be `localhost` or the address of the application server.
- Only grant the account access to the specific databases it needs.
  - Development, UAT and Production environments should all use separate databases and accounts.
- Only grant the required permissions on the databases.
  - Most applications would only need `SELECT`, `UPDATE` and `DELETE` permissions.
  - The account should not be the owner of the database as this can lead to privilege escalation vulnerabilities.
- Avoid using database links or linked servers.
  - Where they are required, use an account that has been granted access to only the minimum databases, tables, and system privileges required.

For more security-critical applications, it is possible to apply permissions at more granular levels, including:

- Table-level permissions.
- Column-level permissions.
- Row-level permissions
- Blocking access to the underlying tables, and requiring all access through restricted views.

Database Configuration and Hardening

The database application should also be properly configured and hardened. The following principles should apply to any database application and platform:

- Install any required security updates and patches.

- Configure the database services to run under a low privileged user account.
- Remove any default accounts and databases.
- Store transaction logs on a separate disk to the main database files.
- Configure a regular backup of the database.
  - Ensure that the backups are protected with appropriate permissions, and ideally encrypted.

Microsoft SQL Server

- Disable `xp_cmdshell`, `xp_dirtree` and other stored procedures that are not required.
- Disable Common Language Runtime (CLR) execution.
- Disable the SQL Browser service.
- Disable Mixed Mode Authentication unless it is required.
- Ensure that the sample Northwind and AdventureWorks databases have been removed.
- See Microsoft's articles on securing SQL Server.

MySQL and MariaDB

- Run the `mysql_secure_installation` script to remove the default databases and accounts.
- Disable the FILE privilege for all users to prevent them reading or writing files.
- See the Oracle MySQL and MariaDB hardening guides.

PostgreSQL

- See the PostgreSQL Server Setup and Operation documentation and the older Security documentation.

MongoDB

- See the MongoDB security checklist.

Redis

- See the Redis security guide.

References

- OWASP Database Security Cheat Sheet

Denial of Service

Introduction

This guides is focused on providing an overall, common overview with an informative, straight to the point guidance to propose angles on how to battle denial of service (DoS) attacks on different layers. It is by no means complete, however, it should serve as an indicator to inform the reader and to introduce a workable methodology to tackle this issue.

Application attacks

Application layer attacks focus on rendering applications unavailable by exhausting resources or by making it unusable in a functional way. These attacks do not have to consume the network bandwidth to be effective. Rather they place an operational strain on the application server in such a way that the server becomes unavailable, unusable or non-functional. All attacks exploiting weaknesses on OSI layer 7 protocol stack are generally categorised as application attacks. They are most challenging to identify/mitigate.

TODO: List all attacks per category. Because we cannot map remediations one on one with an attack vector, we will first need to list them before discussing the action points

Slow HTTP is a DoS attack type where HTTP requests are send very slow and fragmented, one at a time. Until the HTTP request was fully delivered, the server will keep resources stalled while waiting for the missing incoming data. At one moment, the server will reach the maximum concurrent connection pool, resulting in a DoS. From an attacker's perspective, slow HTTP attacks are cheap to perform because they require minimal resources.

Software Design Concepts

- **Cheap validation first**: Validation that is cheap in resources should be considered first. More (CPU, memory and bandwidth) expensive validation should be performed afterward. The reason is obvious, we want to reduce impact on these resources as soon as possible.
- **Graceful Degradation** is the ability of maintaining functionality when portions of a system or application break. DoS caused by application termination is a widespread problem. Implementing a fault tolerant design enables a system or application to continue its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails. Graceful degradation is a core concept to follow during application design phase, in order to limit impact of DoS.

- **Prevent single point of failure**
- **Avoid highly CPU consuming operations**
- **Keep Queues short**
- **Handle Exceptions**
- **Protect overflow and underflow**
- **Threading**: Avoid operations which must wait for completion of large tasks to proceed. Asynchronous operations
- Identify resource intensive pages and plan ahead.

Session

- **Limit server side session time based on inactivity and a final timeout**: (resource exhaustion) While sessions timeout is most of the time discussed in relation to session security and preventing session hijacking, it is also an important measure to prevent resource exhaustion.
- **Limit session bound information storage:** the less data is linked to a session, the less burden a user session has on webserver's performance.

Input validation

- **Limit file upload size and extensions** (resource exhaustion) to prevent DoS on file space storage or other web application functions which will use the upload as input (e.g. image resizing, PDF creation, etc.) - Checklist.
- **Limit total request size** (resource exhaustion) to make it harder for resource consuming DoS attack to succeed.
- **Prevent input based resource allocation** (resource exhaustion).
- **Prevent input based function and threading interaction** (resource exhaustion). User input could influence how many times a function needs to be executed, or how intensive the CPU consumption becomes. Depending on (unfiltered) user input for resource allocation could allow a DoS scenario through resource exhaustion.
- **Input based puzzles** like captchas or simple math problems are often used to 'protect' a web form. They serve a purpose to protect against functionality abuse. The classic example is a webform that will send out an email after posting the request. A captcha could then prevent the mailbox from getting flooded by a malicious attacker or spambot. Notice that this kind of technology will not help defend against DoS attacks.

Access control

- **Authentication as a means to expose functionality**
- **User lockout** is a scenario where an attacker can take advantage of the application security mechanisms to cause DoS by abusing the login failure.

Network attacks

TODO: (Develop text) Attacks where network bandwidth gets saturation. Volumetric in nature. Amplification techniques make these attacks effective.

TODO: (list attacks) NTP amplification, DNS amplification, UDP flooding, TCP flooding

Network Design Concepts

- **Preventing single point of failure**
- **Pooling**
- **Caching** is the concept that data is stored so future requests for that data can be served faster. The more data is served via caching, to more resilient the application becomes to bandwidth exhaustion.
- **Static resources hosting on a different domain** will reduce the number of http requests on the web application. Images and JavaScript are typical files that are loaded from a different domain.

Rate limiting

Rate limiting is the process of controlling traffic rate from and to a server or component. It can be implemented on infrastructure as well as on an application level. Rate limiting can be based on (offending) IPs, on IP block lists, on geolocation, etc.

- **Define a minimum ingress data rate limit**, and drop all connections below that rate. Note that if the rate limit is set too low, this could impact clients. Inspect the logs to establish a baseline of genuine traffic rate. (Protection against slow HTTP attacks)
- **Define an absolute connection timeout**
- **Define a maximum ingress data rate limit**, and drop all connections above that rate.
- **Define a total bandwidth size limit** to prevent bandwidth exhaustion
- **Define a load limit**, which specifies the number of users allowed to access any given resource at any given time.

ISP-Level remediations

- **Filter invalid sender addresses using edge routers**, in accordance with RFC 2267, to filter out IP-spoofing attacks done with the goal of bypassing block lists.
- **Check your ISP services in terms of DDOS beforehand** (support for multiple internet access points, enough bandwidth (xx-xxx Gbit/s) and special hardware for traffic analysis and defence on application level

Global-Level remediations: Commercial cloud filter services

- Consider using a filter service in order to resist larger attacks (up to 500GBit/s)
- **Filter services** support different mechanics to filter out malicious or non compliant traffic
- **Comply with relevant data protection/privacy laws** - a lot of providers route traffic through USA/UK

References

- CERT-EU Whitepaper
- Denial of Service - OWASP Cheat Sheet

File Upload

Overview

File uploading is one of the most important mechanisms in any application, be it web, hybrid mobile, or native. This mechanism ensures that the user is able to upload his photo, his cv, video, etc.

Good Practices

To ensure that secure file uploading, the following principles should be followed:

- List allowed extensions. Only allow safe and critical extensions for business functionality
  - Ensure that input validation is applied before validating the extensions;
- Validate the file type, don't trust the Content-Type header as it can be spoofed;
- Change the filename to something generated by the application;
- Set a filename length limit. Restrict the allowed characters if possible;
- Set a file size limit;
- Only allow authorized users to upload files;
- Store the files on a different server. If that's not possible, store them outside of the webroot
  - In the case of public access to the files, use a handler that gets mapped to filenames inside the application (someid -> file.ext).
- Run the file through an antivirus or a sandbox if available to validate that it doesn't contain malicious data;
- Ensure that any libraries used are securely configured and kept up to date;
- Protect the file upload from CSRF attacks.

References

- File Upload - OWASP Cheat Sheet Series

HTML5 Security

Introduction

The following is a guidelines for implementing HTML 5 in a secure fashion.

Communication APIs

Web Messaging

Web Messaging (also known as Cross Domain Messaging) provides a means of messaging between documents from different origins in a way that is generally safer than the multiple hacks used in the past to accomplish this task.

However, there are still some recommendations to keep in mind:

- When posting a message, explicitly state the expected origin as the second argument to `postMessage` rather than `*` in order to prevent sending the message to an unknown origin after a redirect or some other means of the target window's origin changing;

  The receiving page should **always**:
  - Check the `origin` attribute of the sender to verify the data is originating from the expected location;
  - Perform input validation on the `data` attribute of the event to ensure that it's in the desired format.
- Don't assume you have control over the `data` attribute;
- Both pages should only interpret the exchanged messages as **data**. Never evaluate passed messages as code (e.g. via `eval()`) or insert it to a page DOM (e.g. via `innerHTML`), as that would create a DOM-based XSS vulnerability;
- To assign the data value to an element, instead of using a insecure method like `element.innerHTML=data;`, use the safer option: `element.textContent=data;`

- Check the origin properly exactly to match the FQDN(s) you expect. Note that the following code: `if(message.origin.indexOf(".owasp.org")!=-1) { /* ... */ }` is very insecure and will not have the desired behavior as `owasp.org.attacker.com` will match.
- If you need to embed external content/untrusted gadgets and allow user-controlled scripts (which is highly discouraged), please check the information on sandboxed frames.

Cross Origin Resource Sharing

- Validate URLs passed to `XMLHttpRequest.open`. Current browsers allow these URLs to be cross domain; this behavior can lead to code injection by a remote attacker. Pay extra attention to absolute URLs;
- Ensure that URLs responding with `Access-Control-Allow-Origin: *` do not include any sensitive content or information that might aid attacker in further attacks. Use the `Access-Control-Allow-Origin` header only on chosen URLs that need to be accessed cross-domain. Don't use the header for the whole domain;
- Allow only selected, trusted domains in the `Access-Control-Allow-Origin` header. Prefer allowing specific domains over blocking or allowing any domain (do not use `*` wildcard nor blindly return the `Origin` header content without any checks);
- Keep in mind that CORS does not prevent the requested data from going to an unauthorized location. It's still important for the server to perform usual CSRF prevention;
- While the Fetch Standard recommends a pre-flight request with the `OPTIONS` verb, current implementations might not perform this request, so it's important that "ordinary" (`GET` and `POST`) requests perform any access control necessary;
- Discard requests received over plain HTTP with HTTPS origins to prevent mixed content bugs;
- Don't rely only on the Origin header for Access Control checks. Browser always sends this header in CORS requests, but may be spoofed outside the browser. Application-level protocols should be used to protect sensitive data.

WebSockets

- Drop backward compatibility in implemented client/servers and use only protocol versions above hybi-00. Popular Hixie-76 version (hiby-00) and older are outdated and insecure;
- The recommended version supported in latest versions of all current browsers is RFC 6455 (supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50, and IE10);
- While it's relatively easy to tunnel TCP services through WebSockets (e.g. VNC, FTP), doing so enables access to these tunneled services for the in-browser attacker in case of a Cross Site Scripting attack. These services might also be called directly from a malicious page or program;
- The protocol doesn't handle authorization and/or authentication. Application-level protocols should handle that separately in case sensitive data is being transferred;
- Process the messages received by the websocket as data. Don't try to assign it directly to the DOM nor evaluate as code. If the response is JSON, never use the insecure `eval()` function; use the safe option JSON.parse() instead;
- Only `wss://` (WebSockets over SSL/TLS) should be used for protection against Man-In-The-Middle attacks;
- Always validate input coming from the remote site, as it might have been altered;
- When implementing servers, check the `Origin:` header in the Websockets handshake. Though it might be spoofed outside a browser, browsers always add the Origin of the page that initiated the Websockets connection;
- Always validate data coming through a WebSockets connection.

Server-Sent Events

- Validate URLs passed to the `EventSource` constructor, even though only same-origin URLs are allowed;
- As mentioned before, process the messages (`event.data`) as data and never evaluate the content as HTML or script code;
- Always check the origin attribute of the message (`event.origin`) to ensure the message is coming from a trusted domain. Use an allow-list approach.

Storage APIs

Local Storage

- Avoid storing any sensitive information in local storage where authentication would be assumed;
- Due to the browser's security guarantees it is appropriate to use local storage where access to the data is not assuming authentication or authorization;
- Use the object sessionStorage instead of localStorage if persistent storage is not needed. sessionStorage object is available only to that window/tab until the window is closed;
- It's recommended not to store sensitive information in local storage to avoid a sigle Cross Site Scripting;
- A single Cross Site Scripting can be used to load malicious data into these objects too, so don't consider objects in these to be trusted;
- Pay extra attention to "localStorage.getItem" and "setItem" calls implemented in HTML5 page. It helps in detecting when developers build solutions that put sensitive information in local storage, which can be a severe risk if authentication or authorization to that data is incorrectly assumed;
- Do not store session identifiers in local storage as the data is always accessible by JavaScript. Cookies can mitigate this risk using the `httpOnly` flag;
- Avoid hosting multiple applications on the same origin, all of them would share the same localStorage object, use different subdomains instead.

Client-side databases

- On November 2010, the W3C announced Web SQL Database (relational SQL database) as a deprecated specification. A new standard Indexed Database API or IndexedDB (formerly WebSimpleDB) is actively developed, which provides key-value database storage and methods for performing advanced queries;

- It's recommended not to store any sensitive information in local storage;
- If utilized, WebDatabase content on the client side can be vulnerable to SQL injection and needs to have proper validation and parameterization;
- Don't store any sensitive information in web database.

Geolocation

- When using the Geolocation API, for privacy reasons, it's recommended to require user input before calling `getCurrentPosition` or `watchPosition`.

Web Workers

- Ensure code in all Web Workers scripts is not malevolent;
- Don't allow creating Web Worker scripts from user supplied input;
- Validate messages exchanged with a Web Worker. Do not try to exchange snippets of JavaScript for evaluation e.g. via `eval()` as that could introduce a DOM Based XSS vulnerability.

Tabnabbing

It's the capacity to act on parent page's content or location from a newly opened page via the back link exposed by the **opener** JavaScript object instance.

It applies to an HTML link or a JavaScript `window.open` function using the attribute/instruction `target` to specify a target loading location that does not replace the current location and then makes the current window/tab available.

To prevent this issue, the following actions are available:

Cut the back link between the parent and the child pages:

- For HTML links:
  - To cut this back link, add the attribute `rel="noopener"` on the tag used to create the link from the parent page to the child page. This attribute value cuts the link, but depending on the browser, lets referrer information be present in the request to the child page;
  - To also remove the referrer information use this attribute value: `rel="noopener noreferrer"`.
- For the JavaScript `window.open` function, add the values `noopener,noreferrer` in the windowFeatures parameter of the `window.open` function;

As the behavior using the elements above is different between the browsers, either use an HTML link or JavaScript to open a window (or tab), then use this configuration to maximize the cross supports:

- For HTML links, add the attribute `rel="noopener noreferrer"` to every link;
- For JavaScript, use this function to open a window (or tab):

```
javascript function openPopup(url, name, windowFeatures){ //Open the popup and set the opener and referrer policy instruction var newWindow = window.open(url, name, 'noopener,noreferrer,' + windowFeatures); //Reset the opener link newWindow.opener = null; }
```

- Add the HTTP response header `Referrer-Policy: no-referrer` to every HTTP response sent by the application (Header Referrer-Policy information.

Compatibility matrix:

- noopener;
- noreferrer;
- referrer-policy.

Sandboxed frames

- Use the `sandbox` attribute of an `iframe` for untrusted content;

  The `sandbox` attribute of an `iframe` enables restrictions on content within an `iframe`. The following restrictions are active when the `sandbox` attribute is set:

  1. All markup is treated as being from a unique origin;
  2. All forms and scripts are disabled;
  3. All links are prevented from targeting other browsing contexts;
  4. All features that trigger automatically are blocked;
  5. All plugins are disabled.

  In old versions of user agents where this feature is not supported, this attribute will be ignored. Use this feature as an additional layer of protection or check if the browser supports sandboxed frames and only show the untrusted content if supported.

- Apart from this attribute, to prevent Clickjacking attacks and unsolicited framing it is encouraged to use the header `X-Frame-Options` which supports the `deny` and `same-origin` values. Other solutions like framebusting `if(window!==window.top) { window.top.location=location;}` are not

recommended.

Credential and Personally Identifiable Information (PII) Input hints

- Protect the input values from being cached by the browser.

  Access a financial account from a public computer. Even though one is logged-off, the next person who uses the machine can log-in because the browser autocomplete functionality. To mitigate this, we tell the input fields not to assist in any way.

```
html <input type="text" spellcheck="false" autocomplete="off" autocorrect="off" autocapitalize="off"></input>
```

Text areas and input fields for PII (name, email, address, phone number) and login credentials (username, password) should be prevented from being stored in the browser. Use these HTML5 attributes to prevent the browser from storing PII from your form:

- `spellcheck="false"`;
- `autocomplete="off"`;
- `autocorrect="off"`;
- `autocapitalize="off"`;

Offline Applications

- Require user input before sending any `manifest` file.
- Users should only cache trusted websites and clean the cache after browsing through open or insecure networks.

Progressive Enhancements and Graceful Degradation Risks

- The best practice now is to determine the capabilities that a browser supports and augment with some type of substitute for capabilities that are not directly supported. This may mean an onion-like element, e.g. falling through to a Flash Player if the `<video>` tag is unsupported, or it may mean additional scripting code from various sources that should be code reviewed.

HTTP Headers to enhance security

Consult the project OWASP Secure Headers in order to obtains the list of HTTP security headers that an application should use to enable defenses at browser level.

WebSocket implementation hints

In addition to the elements mentioned above, this is the list of areas for which caution must be taken during the implementation.

- Access filtering through the "Origin" HTTP request header;
- Input/Output validation;
- Authentication;
- Authorization;
- Access token explicit invalidation;
- Confidentiality and Integrity.

References

- HTML5 Security Cheat Sheet

Securing Cascading Style

Introduction

The goal of this CSS Cheat Sheet is to inform Programmers, Testers, Security Analysts, Front-End Developers and anyone who is interested in Web Application Security to use these recommendations or requirements in order to achieve better security when authoring `Cascading Style Sheets`.

Let's demonstrate this risk with an example:

Santhosh is a programmer who works for a company called **X** and authors a Cascading Style Sheet to implement styling of the web application. The application for which he is writing CSS Code has various roles like **Student**, **Teacher**, **Super User** & **Administrator** and these roles have different permissions (PBAC - Permission Based Access Control) and Roles (RBAC - Role Based Access Control). Not only do these roles have different access controls, but these roles could also have different styling for webpages that might be specific to an individual or group of roles.

Santhosh thinks that it would a great optimized idea to create a "global styling" CSS file which has all the CSS styling/selectors for all of the roles. According to their role, a specific feature or user interface element will be rendered. For instance, Administrator will have different features compared to **Student** or **Teacher** or **SuperUser**. However, some permissions or features maybe common to some roles.

Example: Profile Settings will be applicable to all the users here while *Adding Users* or *Deleting Users* is only applicable for **Administrator**.

Example:

- `.login`
- `.profileStudent`
- `.changePassword`
- `.addUsers`
- `.deleteUsers`
- `.addNewAdmin`
- `.deleteAdmin`
- `.exportUserData`
- `.exportProfileData`
- ...

Now, let's examine what are the risks associated with this style of coding.

Risk #1

Motivated Attackers always take a look at `*.CSS` files to learn the features of the application even without being logged in.

For instance: Jim is a motivated attacker and always tries to look into CSS files from the View-Source even before other attacks. When Jim looks into the CSS file, they see that there are different features and different roles based on the CSS selectors like `.profileSettings, .editUser, .addUser, .deleteUser` and so on. Jim can use the CSS for intel gathering to help gain access to sensitive roles. This is a form of attacker due diligence even before trying to perform dangerous attacks to gain access to the web application.

In a nutshell, having global styling could reveal sensitive information that could be beneficial to the attacker.

Risk #2

Let's say, Santhosh has this habit of writing the descriptive selector names like `.profileSettings, exportUserData, .changePassword, .oldPassword, .newPassword, .confirmNewPassword` etc. Good programmers like to keep code readable and usable by other Code Reviewers of the team. The risk is that attackers could map these selectors to actual features of a web application.

Defensive Mechanisms to Mitigate Attacker's Motivation

Defense Mechanism

As a CSS Coder / Programmer, always keep the CSS isolated by access control level. By this, it means **Student** will have a different CSS file called as `StudentStyling.CSS` while **Administrator** has `AdministratorStyling.CSS` and so on. Make sure these `*.CSS` files are accessed only for a user with the proper access control level. Only users with the proper access control level should be able to access their `*.CSS` file.

If an authenticated user with the **Student** Role tries to access `AdministratorStyling.CSS` through forced browsing, an alert that an intrusion is occurring should be recorded.

Defense Mechanism #2

Being a programmer or a tester, take care of the naming conventions of your CSS (Cascading Style Sheet) Selectors. Obfuscate the selector names in such a fashion that attackers are not informed what a specific selector is linking to.

Example: CSS Selectors for `addUser, addAdmin, profileSettings, changePassword` could be named `aHj879JK, bHjsU, ahkrrE, lOiksn` respectively. These names could be randomly generated per user as well.

This NPM package can be used to perform the renaming of the CSS selector.

Defense Mechanism #3

Web applications that allow users to author content via HTML input could be vulnerable to malicious use of CSS. Uploaded HTML could use styles that are allowed by the web application but could be used for purposes other than intended which could lead to security risks.

Example: You can read about how LinkedIn had a vulnerability which allowed malicious use of CSS that lead to the authoring of a page where the entire page was clickable including overwriting LinkedIn's standard navigation elements.

Injection Prevention

Overview

Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.

Forms of Injection

There are several forms of injection targeting different technologies including SQL queries, LDAP queries, XPath queries and OS commands.

Query languages

The most famous form of injection is SQL Injection where an attacker can modify existing database queries.

But also LDAP, SOAP, XPath and REST based queries can be susceptible to injection attacks allowing for data retrieval or control bypass.

SQL Injection

An SQL injection attack consists of insertion or "injection" of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application.

A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

How to test for the issue

During code review

> Please check for any queries to the database are not done via prepared statements;
>
> If dynamic statements are being made please check if the data is sanitized before used as par of the statement;
>
> Auditors should always look for uses of sp_execute, execute or exec within SQL Server stored procedures. Similar audit guidelines are necessary for similar functions for other vendors.

Automated Exploitation

> To find information about how to perform, tester can use an automated auditing tool such as SQLMap;
>
> Equally Static Code Analysis Data flow rules can detect of unsanitized user controlled input can change the SQL query.

Stored Procedure Injection

- When using dynamic SQL within a stored procedure, the application must properly sanitize the user input to eliminate the risk of code injection;
- If not sanitized, the user could enter malicious SQL that will be executed within the stored procedure.

Time delay Exploitation technique

The time delay exploitation technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation.

Out of band Exploitation technique

This technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation.

Remediation

Defense Option 1: Prepared Statements (with Parameterized Queries)

Defense Option 2: Stored Procedures

Defense Option 3: Allow-List Input Validation

Defense Option 4: Escaping All User-Supplied Input

LDAP Injection

LDAP injection attacks are common due to two factors:

1. The lack of safer, parameterized LDAP query interfaces;
2. The widespread use of LDAP to authenticate users to systems.

How to test for the issue

During code review

Please check for any queries to the LDAP escape special characters, see here.

Automated Exploitation

Use scanner module of tool like OWASP ZAP to detect LDAP injection issue.

Remediation

Escape all variables using the right LDAP encoding function

XPath Injection

TODO

Scripting languages

All scripting languages used in web applications have a form of an `eval` call which receives code at runtime and executes it. If code is crafted using unvalidated and unescaped user input code injection can occur which allows an attacker to subvert application logic and eventually to gain local access.

Every time a scripting language is used, the actual implementation of the 'higher' scripting language is done using a 'lower' language like C. If the scripting language has a flaw in the data handling code 'Null Byte Injection' attack vectors can be deployed to gain access to other areas in memory, which results in a successful attack.

Operating System Commands

OS command injection is a technique used via a web interface in order to execute OS commands on a web server. The user supplies operating system commands through a web interface in order to execute OS commands.

Any web interface that is not properly sanitized is subject to this exploit. With the ability to execute OS commands, the user can upload malicious programs or even obtain passwords. OS command injection is preventable when security is emphasized during the design and development of applications.

How to test for the issue

During code review

- Check if any command execute methods are called and in unvalidated user input are taken as data for that command.
- Out side of that, appending a semicolon to the end of a URL query parameter followed by an operating system command, will execute the command. `%3B` is URL encoded and decodes to semicolon. This is because the `;` is interpreted as a command separator.
- If the application responds with the output of the `/etc/passwd` file then you know the attack has been successful. Many web application scanners can be used to test for this attack as they inject variations of command injections and test the response.
- Equally Static Code Analysis tools check the data flow of untrusted user input into a web application and check if the data is then entered into a dangerous method which executes the user input as a command.

Remediation

If it is considered unavoidable the call to a system command incorporated with user-supplied, the following two layers of defense should be used within software in order to prevent attacks

1. **Parameterization** - If available, use structured mechanisms that automatically enforce the separation between data and command. These mechanisms can help to provide the relevant quoting, encoding.

   Input validation - the values for commands and the relevant arguments should be both validated. There are different degrees of validation for the actual command and its arguments:

   - When it comes to the **commands** used, these must be validated against a list of allowed commands.

     In regards to the **arguments** used for these commands, they should be validated using the following options:

     - Positive or "allow list" input validation - where are the arguments allowed explicitly defined
     - Allow-list Regular Expression - where is explicitly defined a list of good characters allowed and the maximum length of the string. Ensure that metacharacters like `& | ; $ > < \ \! ``` and white-spaces are not part of the Regular Expression. For example, the following regular expression only allows lowercase letters and numbers, and does not contain metacharacters. The length is also being limited to 3-10 characters:

```
^[a-z0-9]{3,10}$
```

Network Protocols

Web applications often communicate with network daemons (like SMTP, IMAP, FTP) where user input becomes part of the communication stream. Here it is possible to inject command sequences to abuse an established session.

Injection Prevention Rules

Rule #1 (Perform proper input validation)

Perform proper input validation. Positive or "allow list" input validation with appropriate canonicalization is also recommended, but **is not a complete defense** as many applications require special characters in their input.

Rule #2 (Use a safe API)

The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful of APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.

Rule #3 (Contextually escape user data)

If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter.

Other Injection Cheatsheets

SQL Injection Prevention Cheat Sheet

OS Command Injection Defense Cheat Sheet

LDAP Injection Prevention Cheat Sheet

Injection Prevention Cheat Sheet in Java

Password Storage

Introduction

It is essential to store passwords in a way that prevents them from being obtained by an attacker even if the application or database is compromised. The majority of modern languages and frameworks provide built-in functionality to help store passwords safely.

After an attacker has acquired stored password hashes, they are always able to brute force hashes offline. As a defender, it is only possible to slow down offline attacks by selecting hash algorithms that are as resource intensive as possible.

This cheat sheet provides guidance on the various areas that need to be considered related to storing passwords. In short:

- **Use Argon2id with a minimum configuration of 15 MiB of memory, an iteration count of 2, and 1 degree of parallelism.**
- **If Argon2id is not available, use scrypt with a minimum CPU/memory cost parameter of (2^16), a minimum block size of 8 (1024 bytes), and a parallelization parameter of 1.**
- **For legacy systems using bcrypt, use a work factor of 10 or more and with a password limit of 72 bytes.**
- **If FIPS-140 compliance is required, use PBKDF2 with a work factor of 310,000 or more and set with an internal hash function of HMAC-SHA-256.**
- **Consider using a pepper to provide additional defense in depth (though alone, it provides no additional secure characteristics).**

Background

Hashing vs Encryption

Hashing and encryption both provide ways to keep sensitive data safe. However, in almost all circumstances, **passwords should be hashed, NOT encrypted.**

Hashing is a one-way function (i.e., it is impossible to "decrypt" a hash and obtain the original plaintext value). Hashing is appropriate for password validation. Even if an attacker obtains the hashed password, they cannot enter it into an application's password field and log in as the victim.

Encryption is a two-way function, meaning that the original plaintext can be retrieved. Encryption is appropriate for storing data such as a user's address since this data is displayed in plaintext on the user's profile. Hashing their address would result in a garbled mess.

In the context of password storage, encryption should only be used in edge cases where it is necessary to obtain the original plaintext password. This might be necessary if the application needs to use the password to authenticate with another system that does not support a modern way to programmatically grant access, such as OpenID Connect (OIDC). Where possible, an alternative architecture should be used to avoid the need to store passwords in an encrypted form.

For further guidance on encryption, see the Cryptographic Storage Cheat Sheet.

How Attackers Crack Password Hashes

Although it is not possible to "decrypt" password hashes to obtain the original passwords, it is possible to "crack" the hashes in some circumstances.

The basic steps are:

- Select a password you think the victim has chosen (e.g. `password1!`)
- Calculate the hash

- Compare the hash you calculated to the hash of the victim. If they match, you have correctly "cracked" the hash and now know the plaintext value of their password.

This process is repeated for a large number of potential candidate passwords. Different methods can be used to select candidate passwords, including:

- Lists of passwords obtained from other compromised sites
- Brute force (trying every possible candidate)
- Dictionaries or wordlists of common passwords

While the number of permutations can be enormous, with high speed hardware (such as GPUs) and cloud services with many servers for rent, the cost to an attacker is relatively small to do successful password cracking especially when best practices for hashing are not followed.

Strong passwords stored with modern hashing algorithms and using hashing best practices should be effectively impossible for an attacker to crack. It is your responsibility as an application owner to select a modern hashing algorithm.

Password Storage Concepts

Salting

A salt is a unique, randomly generated string that is added to each password as part of the hashing process. As the salt is unique for every user, an attacker has to crack hashes one at a time using the respective salt rather than calculating a hash once and comparing it against every stored hash. This makes cracking large numbers of hashes significantly harder, as the time required grows in direct proportion to the number of hashes.

Salting also protects against an attacker pre-computing hashes using rainbow tables or database-based lookups. Finally, salting means that it is impossible to determine whether two users have the same password without cracking the hashes, as the different salts will result in different hashes even if the passwords are the same.

Modern hashing algorithms such as Argon2id, bcrypt, and PBKDF2 automatically salt the passwords, so no additional steps are required when using them.

Peppering

A pepper can be used in addition to salting to provide an additional layer of protection. The purpose of the pepper is to prevent an attacker from being able to crack any of the hashes if they only have access to the database, for example, if they have exploited a SQL injection vulnerability or obtained a backup of the database.

One of several peppering strategies is to hash the passwords as usual (using a password hashing algorithm) and then HMAC or encrypt the hashes with a symmetrical encryption key before storing the password hash in the database, with the key acting as the pepper. Peppering strategies do not affect the password hashing function in any way.

- The pepper is **shared between stored passwords**, rather than being *unique* like a salt.
- Unlike a password salt, the pepper **should not be stored in the database**.
- Peppers are secrets and should be stored in "secrets vaults" or HSMs (Hardware Security Modules).
- Like any other cryptographic key, a pepper rotation strategy should be considered.

Work Factors

The work factor is essentially the number of iterations of the hashing algorithm that are performed for each password (usually, it's actually `2^work` iterations). The purpose of the work factor is to make calculating the hash more computationally expensive, which in turn reduces the speed and/or increases the cost for which an attacker can attempt to crack the password hash. The work factor is typically stored in the hash output.

When choosing a work factor, a balance needs to be struck between security and performance. Higher work factors will make the hashes more difficult for an attacker to crack but will also make the process of verifying a login attempt slower. If the work factor is too high, this may degrade the performance of the application and could also be used by an attacker to carry out a denial of service attack by making a large number of login attempts to exhaust the server's CPU.

There is no golden rule for the ideal work factor - it will depend on the performance of the server and the number of users on the application. Determining the optimal work factor will require experimentation on the specific server(s) used by the application. As a general rule, calculating a hash should take less than one second.

Upgrading the Work Factor

One key advantage of having a work factor is that it can be increased over time as hardware becomes more powerful and cheaper.

The most common approach to upgrading the work factor is to wait until the user next authenticates and then to re-hash their password with the new work factor. This means that different hashes will have different work factors and may result in hashes never being upgraded if the user doesn't log back into the application. Depending on the application, it may be appropriate to remove the older password hashes and require users to reset their passwords next time they need to login in order to avoid storing older and less secure hashes.

Password Hashing Algorithms

There are a number of modern hashing algorithms that have been specifically designed for securely storing passwords. This means that they should be slow (unlike algorithms such as MD5 and SHA-1, which were designed to be fast), and how slow they are can be configured by changing the work factor.

Websites should not hide which password hashing algorithm they use. If you utilize a modern password hashing algorithm with proper configuration parameters, it should be safe to state in public which password hashing algorithms are in use and be listed here.

The main three algorithms that should be considered are listed below:

Argon2id

Argon2 is the winner of the 2015 Password Hashing Competition. There are three different versions of the algorithm, and the Argon2id variant should be used, as it provides a balanced approach to resisting both side-channel and GPU-based attacks.

Rather than a simple work factor like other algorithms, Argon2id has three different parameters that can be configured. Argon2id should use one of the following configuration settings as a base minimum which includes the minimum memory size (m), the minimum number of iterations (t) and the degree of parallelism (p).

- m=37 MiB, t=1, p=1
- m=15 MiB, t=2, p=1

Both of these configuration settings are equivalent in the defense they provide. The only difference is a trade off between CPU and RAM usage.

scrypt

scrypt is a password-based key derivation function created by Colin Percival. While new systems should consider Argon2id for password hashing, scrypt should be configured properly when used in legacy systems.

Like Argon2id, scrypt has three different parameters that can be configured. scrypt should use one of the following configuration settings as a base minimum which includes the minimum CPU/memory cost parameter (N), the blocksize (r) and the degree of parallelism (p).

- N=2^16 (64 MiB), r=8 (1024 bytes), p=1
- N=2^15 (32 MiB), r=8 (1024 bytes), p=2
- N=2^14 (16 MiB), r=8 (1024 bytes), p=4
- N=2^13 (8 MiB), r=8 (1024 bytes), p=8
- N=2^12 (4 MiB), r=8 (1024 bytes), p=15

These configuration settings are equivalent in the defense they provide. The only difference is a trade off between CPU and RAM usage.

bcrypt

The bcrypt password hashing function should be the second choice for password storage if Argon2id is not available or PBKDF2 is required to achieve FIPS-140 compliance.

The minimum work factor for bcrypt should be 10.

Input Limits

bcrypt has a maximum length input length of 72 bytes for most implementations. To protect against this issue, a maximum password length of 72 bytes (or less if the implementation in use has smaller limits) should be enforced when using bcrypt.

Pre-Hashing Passwords

An alternative approach is to pre-hash the user-supplied password with a fast algorithm such as SHA-256, and then to hash the resulting hash with bcrypt (i.e., `bcrypt(base64(hmac-sha256(data:$password, key:$pepper)), $salt, $cost)`). This is a dangerous (but common) practice that **should be avoided** due to password shucking and other issues when combining bcrypt with other hash functions.

PBKDF2

PBKDF2 is recommended by NIST and has FIPS-140 validated implementations. So, it should be the preferred algorithm when these are required.

PBKDF2 requires that you select an internal hashing algorithm such as an HMAC or a variety of other hashing algorithms. HMAC-SHA-256 is widely supported and is recommended by NIST.

The work factor for PBKDF2 is implemented through an iteration count, which should set differently based on the internal hashing algorithm used.

- PBKDF2-HMAC-SHA1: 720,000 iterations
- PBKDF2-HMAC-SHA256: 310,000 iterations
- PBKDF2-HMAC-SHA512: 120,000 iterations

These configuration settings are equivalent in the defense they provide.

When PBKDF2 is used with an HMAC, and the password is longer than the hash function's block size (64 bytes for SHA-256), the password will be automatically pre-hashed. For example, the password "This is a password longer than 512 bits which is the block size of SHA-256" is converted to the hash value (in hex) fa91498c139805af73f7ba275cca071e78d78675027000c99a9925e2ec92eedd. A good implementation of PBKDF2 will perform this step before the expensive iterated hashing phase, but some implementations perform the conversion on each iteration. This can make hashing long passwords significantly more expensive than hashing short passwords. If a user can supply very long passwords, there is a potential denial of service vulnerability, such as the one published in Django in 2013. Manual pre-hashing can reduce this risk but requires adding a salt to the pre-hash step.

Upgrading Legacy Hashes

For older applications built using less secure hashing algorithms such as MD5 or SHA-1, these hashes should be upgraded to modern password hashing algorithms as described above. When the user next enters their password (usually by authenticating on the application), it should be re-hashed using the new algorithm. It would also be good practice to expire the users' current password and require them to enter a new one so that any older (less secure) hashes of their password are no longer useful to an attacker.

However, this approach means that old (less secure) password hashes will be stored in the database until the user logs in. Two main approaches can be taken to avoid this dilemma.

One method is to expire and delete the password hashes of users who have been inactive for an extended period and require them to reset their passwords to login again. Although secure, this approach is not particularly user-friendly. Expiring the passwords of many users may cause issues for support staff or may be interpreted by users as an indication of a breach.

An alternative approach is to use the existing password hashes as inputs for a more secure algorithm. For example, if the application originally stored passwords as `md5($password)`, this could be easily upgraded to `bcrypt(md5($password))`. Layering the hashes avoids the need to know the original password; however, it can make the hashes easier to crack. These hashes should be replaced with direct hashes of the users' passwords next time the user logs in.

Assume that whatever password hashing method is selected will have to be upgraded in the future. Ensure that upgrading your hashing algorithm is as easy as possible. For a transition period, allow for a mix of old and new hashing algorithms. Using a mix of hashing algorithms is easier if the password hashing algorithm and work factor are stored with the password using a standard format, for example, the modular PHC string format.

International Characters

Ensure your hashing library is able to accept a wide range of characters and is compatible with all Unicode codepoints. Users should be able to use the full range of characters available on modern devices, in particular mobile keyboards. They should be able to select passwords from various languages and include pictograms. Prior to hashing the entropy of the user's entry should not be reduced. Password hashing libraries need to be able to use input that may contain a NULL byte.

Session Management

Introduction

Web Authentication, Session Management, and Access Control:

A web session is a sequence of network HTTP request and response transactions associated with the same user. Modern and complex web applications require the retaining of information or status about each user for the duration of multiple requests. Therefore, sessions provide the ability to establish variables – such as access rights and localization settings – which will apply to each and every interaction a user has with the web application for the duration of the session.

Web applications can create sessions to keep track of anonymous users after the very first user request. An example would be maintaining the user language preference. Additionally, web applications will make use of sessions once the user has authenticated. This ensures the ability to identify the user on any subsequent requests as well as being able to apply security access controls, authorized access to the user private data, and to increase the usability of the application. Therefore, current web applications can provide session capabilities both pre and post authentication.

Once an authenticated session has been established, the session ID (or token) is temporarily equivalent to the strongest authentication method used by the application, such as username and password, passphrases, one-time passwords (OTP), client-based digital certificates, smartcards, or biometrics (such as fingerprint or eye retina). See the OWASP Authentication Cheat Sheet.

HTTP is a stateless protocol (RFC2616 section 5), where each request and response pair is independent of other web interactions. Therefore, in order to introduce the concept of a session, it is required to implement session management capabilities that link both the authentication and access control (or authorization) modules commonly available in web applications:

The session ID or token binds the user authentication credentials (in the form of a user session) to the user HTTP traffic and the appropriate access controls enforced by the web application. The complexity of these three components (authentication, session management, and access control) in modern web applications, plus the fact that its implementation and binding resides on the web developer's hands (as web development frameworks do not provide strict relationships between these modules), makes the implementation of a secure session management module very challenging.

The disclosure, capture, prediction, brute force, or fixation of the session ID will lead to session hijacking (or sidejacking) attacks, where an attacker is able to fully impersonate a victim user in the web application. Attackers can perform two types of session hijacking attacks, targeted or generic. In a targeted attack, the attacker's goal is to impersonate a specific (or privileged) web application victim user. For generic attacks, the attacker's goal is to impersonate (or get access as) any valid or legitimate user in the web application.

Session ID Properties

In order to keep the authenticated state and track the users progress within the web application, applications provide users with a **session identifier** (session ID or token) that is assigned at session creation time, and is shared and exchanged by the user and the web application for the duration of the session (it is sent on every HTTP request). The session ID is a `name=value` pair.

With the goal of implementing secure session IDs, the generation of identifiers (IDs or tokens) must meet the following properties.

Session ID Name Fingerprinting

The name used by the session ID should not be extremely descriptive nor offer unnecessary details about the purpose and meaning of the ID.

The session ID names used by the most common web application development frameworks can be easily fingerprinted, such as `PHPSESSID` (PHP), `JSESSIONID` (J2EE), `CFID` & `CFTOKEN` (ColdFusion), `ASP.NET_SessionId` (ASP .NET), etc. Therefore, the session ID name can disclose the technologies and programming languages used by the web application.

It is recommended to change the default session ID name of the web development framework to a generic name, such as `id`.

Session ID Length

The session ID must be long enough to prevent brute force attacks, where an attacker can go through the whole range of ID values and verify the existence of valid sessions.

The session ID length must be at least `128 bits (16 bytes)`.

NOTE:

- The session ID length of 128 bits is provided as a reference based on the assumptions made on the next section *Session ID Entropy*. However, this number should not be considered as an absolute minimum value, as other implementation factors might influence its strength.
- For example, there are well-known implementations, such as Microsoft ASP.NET session IDs: "*The ASP .NET session identifier is a randomly generated number encoded into a 24-character string consisting of lowercase characters from a to z and numbers from 0 to 5*".
- It can provide a very good effective entropy, and as a result, can be considered long enough to avoid guessing or brute force attacks.

Session ID Entropy

The session ID must be unpredictable (random enough) to prevent guessing attacks, where an attacker is able to guess or predict the ID of a valid session through statistical analysis techniques. For this purpose, a good CSPRNG (Cryptographically Secure Pseudorandom Number Generator) must be used.

The session ID value must provide at least `64 bits` of entropy (if a good PRNG is used, this value is estimated to be half the length of the session ID).

Additionally, a random session ID is not enough; it must also be unique to avoid duplicated IDs. A random session ID must not already exist in the current session ID space.

NOTE:

- The session ID entropy is really affected by other external and difficult to measure factors, such as the number of concurrent active sessions the web application commonly has, the absolute session expiration timeout, the amount of session ID guesses per second the attacker can make and the target web application can support, etc.
- If a session ID with an entropy of `64 bits` is used, it will take an attacker at least 292 years to successfully guess a valid session ID, assuming the attacker can try 10,000 guesses per second with 100,000 valid simultaneous sessions available in the web application.
- More information here.

Session ID Content (or Value)

The session ID content (or value) must be meaningless to prevent information disclosure attacks, where an attacker is able to decode the contents of the ID and extract details of the user, the session, or the inner workings of the web application.

The session ID must simply be an identifier on the client side, and its value must never include sensitive information (or PII).

The meaning and business or application logic associated with the session ID must be stored on the server side, and specifically, in session objects or in a session management database or repository.

The stored information can include the client IP address, User-Agent, e-mail, username, user ID, role, privilege level, access rights, language preferences, account ID, current state, last login, session timeouts, and other internal session details. If the session objects and properties contain sensitive information, such as credit card numbers, it is required to duly encrypt and protect the session management repository.

It is recommended to use the session ID created by your language or framework. If you need to create your own sessionID, use a cryptographically secure pseudorandom number generator (CSPRNG) with a size of at least 128 bits and ensure that each sessionID is unique.

Session Management Implementation

The session management implementation defines the exchange mechanism that will be used between the user and the web application to share and continuously exchange the session ID. There are multiple mechanisms available in HTTP to maintain session state within web applications, such as cookies

(standard HTTP header), URL parameters (URL rewriting – RFC2396), URL arguments on GET requests, body arguments on POST requests, such as hidden form fields (HTML forms), or proprietary HTTP headers.

The preferred session ID exchange mechanism should allow defining advanced token properties, such as the token expiration date and time, or granular usage constraints. This is one of the reasons why cookies (RFCs 2109 & 2965 & 6265) are one of the most extensively used session ID exchange mechanisms, offering advanced capabilities not available in other methods.

The usage of specific session ID exchange mechanisms, such as those where the ID is included in the URL, might disclose the session ID (in web links and logs, web browser history and bookmarks, the Referer header or search engines), as well as facilitate other attacks, such as the manipulation of the ID or session fixation attacks.

Built-in Session Management Implementations

Web development frameworks, such as J2EE, ASP .NET, PHP, and others, provide their own session management features and associated implementation. It is recommended to use these built-in frameworks versus building a home made one from scratch, as they are used worldwide on multiple web environments and have been tested by the web application security and development communities over time.

However, be advised that these frameworks have also presented vulnerabilities and weaknesses in the past, so it is always recommended to use the latest version available, that potentially fixes all the well-known vulnerabilities, as well as review and change the default configuration to enhance its security by following the recommendations described along this document.

The storage capabilities or repository used by the session management mechanism to temporarily save the session IDs must be secure, protecting the session IDs against local or remote accidental disclosure or unauthorized access.

Used vs. Accepted Session ID Exchange Mechanisms

A web application should make use of cookies for session ID exchange management. If a user submits a session ID through a different exchange mechanism, such as a URL parameter, the web application should avoid accepting it as part of a defensive strategy to stop session fixation.

NOTE:

- Even if a web application makes use of cookies as its default session ID exchange mechanism, it might accept other exchange mechanisms too.
- It is therefore required to confirm via thorough testing all the different mechanisms currently accepted by the web application when processing and managing session IDs, and limit the accepted session ID tracking mechanisms to just cookies.
- In the past, some web applications used URL parameters, or even switched from cookies to URL parameters (via automatic URL rewriting), if certain conditions are met (for example, the identification of web clients without support for cookies or not accepting cookies due to user privacy concerns).

Transport Layer Security

In order to protect the session ID exchange from active eavesdropping and passive disclosure in the network traffic, it is essential to use an encrypted HTTPS (TLS) connection for the entire web session, not only for the authentication process where the user credentials are exchanged. This may be mitigated by HTTP Strict Transport Security (HSTS) for a client that supports it.

Additionally, the `Secure` cookie attribute must be used to ensure the session ID is only exchanged through an encrypted channel. The usage of an encrypted communication channel also protects the session against some session fixation attacks where the attacker is able to intercept and manipulate the web traffic to inject (or fix) the session ID on the victim's web browser (see here and here).

The following set of best practices are focused on protecting the session ID (specifically when cookies are used) and helping with the integration of HTTPS within the web application:

- Do not switch a given session from HTTP to HTTPS, or vice-versa, as this will disclose the session ID in the clear through the network.
  - When redirecting to HTTPS, ensure that the cookie is set or regenerated **after** the redirect has occurred.
- Do not mix encrypted and unencrypted contents (HTML pages, images, CSS, JavaScript files, etc) in the same page, or from the same domain.
- Where possible, avoid offering public unencrypted contents and private encrypted contents from the same host. Where insecure content is required, consider hosting this on a separate insecure domain.
- Implement HTTP Strict Transport Security (HSTS) to enforce HTTPS connections.

See the OWASP Transport Layer Protection Cheat Sheet for more general guidance on implementing TLS securely.

It is important to emphasize that TLS does not protect against session ID prediction, brute force, client-side tampering or fixation; however, it does provide effective protection against an attacker intercepting or stealing session IDs through a man in the middle attack.

Cookies

The session ID exchange mechanism based on cookies provides multiple security features in the form of cookie attributes that can be used to protect the exchange of the session ID:

Secure Attribute

The `Secure` cookie attribute instructs web browsers to only send the cookie through an encrypted HTTPS (SSL/TLS) connection. This session protection mechanism is mandatory to prevent the disclosure of the session ID through MitM (Man-in-the-Middle) attacks. It ensures that an attacker cannot simply capture the session ID from web browser traffic.

Forcing the web application to only use HTTPS for its communication (even when port TCP/80, HTTP, is closed in the web application host) does not protect against session ID disclosure if the `Secure` cookie has not been set - the web browser can be deceived to disclose the session ID over an unencrypted HTTP connection. The attacker can intercept and manipulate the victim user traffic and inject an HTTP unencrypted reference to the web application that will force the web browser to submit the session ID in the clear.

See also: SecureFlag

HttpOnly Attribute

The `HttpOnly` cookie attribute instructs web browsers not to allow scripts (e.g. JavaScript or VBscript) an ability to access the cookies via the DOM document.cookie object. This session ID protection is mandatory to prevent session ID stealing through XSS attacks. However, if an XSS attack is combined with a CSRF attack, the requests sent to the web application will include the session cookie, as the browser always includes the cookies when sending requests. The `HttpOnly` cookie only protects the confidentiality of the cookie; the attacker cannot use it offline, outside of the context of an XSS attack.

See the OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet.

See also: HttpOnly

SameSite Attribute

SameSite defines a cookie attribute preventing browsers from sending a SameSite flagged cookie with cross-site requests. The main goal is to mitigate the risk of cross-origin information leakage, and provides some protection against cross-site request forgery attacks.

See also: SameSite

Domain and Path Attributes

The `Domain` cookie attribute instructs web browsers to only send the cookie to the specified domain and all subdomains. If the attribute is not set, by default the cookie will only be sent to the origin server. The `Path` cookie attribute instructs web browsers to only send the cookie to the specified directory or subdirectories (or paths or resources) within the web application. If the attribute is not set, by default the cookie will only be sent for the directory (or path) of the resource requested and setting the cookie.

It is recommended to use a narrow or restricted scope for these two attributes. In this way, the `Domain` attribute should not be set (restricting the cookie just to the origin server) and the `Path` attribute should be set as restrictive as possible to the web application path that makes use of the session ID.

Setting the `Domain` attribute to a too permissive value, such as `example.com` allows an attacker to launch attacks on the session IDs between different hosts and web applications belonging to the same domain, known as cross-subdomain cookies. For example, vulnerabilities in `www.example.com` might allow an attacker to get access to the session IDs from `secure.example.com`.

Additionally, it is recommended not to mix web applications of different security levels on the same domain. Vulnerabilities in one of the web applications would allow an attacker to set the session ID for a different web application on the same domain by using a permissive `Domain` attribute (such as `example.com`) which is a technique that can be used in session fixation attacks.

Although the `Path` attribute allows the isolation of session IDs between different web applications using different paths on the same host, it is highly recommended not to run different web applications (especially from different security levels or scopes) on the same host. Other methods can be used by these applications to access the session IDs, such as the `document.cookie` object. Also, any web application can set cookies for any path on that host.

Cookies are vulnerable to DNS spoofing/hijacking/poisoning attacks, where an attacker can manipulate the DNS resolution to force the web browser to disclose the session ID for a given host or domain.

Expire and Max-Age Attributes

Session management mechanisms based on cookies can make use of two types of cookies, non-persistent (or session) cookies, and persistent cookies. If a cookie presents the `Max-Age` (that has preference over `Expires`) or `Expires` attributes, it will be considered a persistent cookie and will be stored on disk by the web browser based until the expiration time.

Typically, session management capabilities to track users after authentication make use of non-persistent cookies. This forces the session to disappear from the client if the current web browser instance is closed. Therefore, it is highly recommended to use non-persistent cookies for session management purposes, so that the session ID does not remain on the web client cache for long periods of time, from where an attacker can obtain it.

- Ensure that sensitive information is not compromised by ensuring that it is not persistent, encrypting it, and storing it only for the duration of the need
- Ensure that unauthorized activities cannot take place via cookie manipulation
- Ensure secure flag is set to prevent accidental transmission over the wire in a non-secure manner
- Determine if all state transitions in the application code properly check for the cookies and enforce their use
- Ensure entire cookie should be encrypted if sensitive data is persisted in the cookie
- Define all cookies being used by the application, their name and why they are needed

HTML5 Web Storage API

The Web Hypertext Application Technology Working Group (WHATWG) describes the HTML5 Web Storage APIs, `localStorage` and `sessionStorage`, as mechanisms for storing name-value pairs client-side. Unlike HTTP cookies, the contents of `localStorage` and `sessionStorage` are not automatically shared within requests or responses by the browser and are used for storing data client-side.

The localStorage API

Scope

Data stored using the `localStorage` API is accessible by pages which are loaded from the same origin, which is defined as the scheme (`https://`), host (`example.com`), port (`443`) and domain/realm (`example.com`). This provides similar access to this data as would be achieved by using the `secure` flag on a cookie, meaning that data stored from `https` could not be retrieved via `http`. Due to potential concurrent access from separate windows/threads, data stored using `localStorage` may be susceptible to shared access issues (such as race-conditions) and should be considered non-locking (Web Storage API Spec).

Duration

Data stored using the `localStorage` API is persisted across browsing sessions, extending the timeframe in which it may be accessible to other system users.

Offline Access

The standards do not require `localStorage` data to be encrypted-at-rest, meaning it may be possible to directly access this data from disk.

Use Case

WHATWG suggests the use of `localStorage` for data that needs to be accessed across windows or tabs, across multiple sessions, and where large (multi-megabyte) volumes of data may need to be stored for performance reasons.

The sessionStorage API

Scope

The `sessionStorage` API stores data within the window context from which it was called, meaning that Tab 1 cannot access data which was stored from Tab 2. Also, like the `localStorage` API, data stored using the `sessionStorage` API is accessible by pages which are loaded from the same origin, which is defined as the scheme (`https://`), host (`example.com`), port (`443`) and domain/realm (`example.com`). This provides similar access to this data as would be achieved by using the `secure` flag on a cookie, meaning that data stored from `https` could not be retrieved via `http`.

Duration

The `sessionStorage` API only stores data for the duration of the current browsing session. Once the tab is closed, that data is no longer retrievable. This does not necessarily prevent access, should a browser tab be reused or left open. Data may also persist in memory until a garbage collection event.

Offline Access

The standards do not require `sessionStorage` data to be encrypted-at-rest, meaning it may be possible to directly access this data from disk.

Use Case

WHATWG suggests the use of `sessionStorage` for data that is relevant for one-instance of a workflow, such as details for a ticket booking, but where multiple workflows could be performed in other tabs concurrently. The window/tab bound nature will keep the data from leaking between workflows in separate tabs.

References

- Web Storage APIs
- LocalStorage API
- SessionStorage API
- WHATWG Web Storage Spec

Web Workers

Web Workers run JavaScript code in a global context separate from the one of the current window. A communication channel with the main execution window exists, which is called `MessageChannel`.

Use Case

Web Workers are an alternative for browser storage of (session) secrets when storage persistence across page refresh is not a requirement. For Web Workers to provide secure browser storage, any code that requires the secret should exist within the Web Worker and the secret should never be transmitted to the main window context.

Storing secrets within the memory of a Web Worker offers the same security guarantees as an HttpOnly cookie: the confidentiality of the secret is protected. Still, an XSS attack can be used to send messages to the Web Worker to perform an operation that requires the secret. The Web Worker will return the result of the

operation to the main execution thread.

The advantage of a Web Worker implementation compared to an HttpOnly cookie is that a Web Worker allows for some isolated JavaScript code to access the secret; an HttpOnly cookie is not accessible to any JavaScript. If the frontend JavaScript code requires access to the secret, the Web Worker implementation is the only browser storage option that preserves the secret confidentiality.

Session ID Life Cycle

Session ID Generation and Verification: Permissive and Strict Session Management

There are two types of session management mechanisms for web applications, permissive and strict, related to session fixation vulnerabilities. The permissive mechanism allows the web application to initially accept any session ID value set by the user as valid, creating a new session for it, while the strict mechanism enforces that the web application will only accept session ID values that have been previously generated by the web application.

The session tokens should be handled by the web server if possible or generated via a cryptographically secure random number generator.

Although the most common mechanism in use today is the strict one (more secure), PHP defaults to permissive. Developers must ensure that the web application does not use a permissive mechanism under certain circumstances. Web applications should never accept a session ID they have never generated, and in case of receiving one, they should generate and offer the user a new valid session ID. Additionally, this scenario should be detected as a suspicious activity and an alert should be generated.

Manage Session ID as Any Other User Input

Session IDs must be considered untrusted, as any other user input processed by the web application, and they must be thoroughly validated and verified. Depending on the session management mechanism used, the session ID will be received in a GET or POST parameter, in the URL or in an HTTP header (e.g. cookies). If web applications do not validate and filter out invalid session ID values before processing them, they can potentially be used to exploit other web vulnerabilities, such as SQL injection if the session IDs are stored on a relational database, or persistent XSS if the session IDs are stored and reflected back afterwards by the web application.

Renew the Session ID After Any Privilege Level Change

The session ID must be renewed or regenerated by the web application after any privilege level change within the associated user session. The most common scenario where the session ID regeneration is mandatory is during the authentication process, as the privilege level of the user changes from the unauthenticated (or anonymous) state to the authenticated state though in some cases still not yet the authorized state. Common scenarios to consider include; password changes, permission changes, or switching from a regular user role to an administrator role within the web application. For all sensitive pages of the web application, any previous session IDs must be ignored, only the current session ID must be assigned to every new request received for the protected resource, and the old or previous session ID must be destroyed.

The most common web development frameworks provide session functions and methods to renew the session ID, such as `request.getSession(true)` & `HttpSession.invalidate()` (J2EE), `Session.Abandon()` & `Response.Cookies.Add(new...)` (ASP .NET), or `session_start()` & `session_regenerate_id(true)` (PHP).

The session ID regeneration is mandatory to prevent session fixation attacks, where an attacker sets the session ID on the victim user's web browser instead of gathering the victim's session ID, as in most of the other session-based attacks, and independently of using HTTP or HTTPS. This protection mitigates the impact of other web-based vulnerabilities that can also be used to launch session fixation attacks, such as HTTP response splitting or XSS (see here and here).

A complementary recommendation is to use a different session ID or token name (or set of session IDs) pre and post authentication, so that the web application can keep track of anonymous users and authenticated users without the risk of exposing or binding the user session between both states.

Considerations When Using Multiple Cookies

If the web application uses cookies as the session ID exchange mechanism, and multiple cookies are set for a given session, the web application must verify all cookies (and enforce relationships between them) before allowing access to the user session.

It is very common for web applications to set a user cookie pre-authentication over HTTP to keep track of unauthenticated (or anonymous) users. Once the user authenticates in the web application, a new post-authentication secure cookie is set over HTTPS, and a binding between both cookies and the user session is established. If the web application does not verify both cookies for authenticated sessions, an attacker can make use of the pre-authentication unprotected cookie to get access to the authenticated user session (see here and here).

Web applications should try to avoid the same cookie name for different paths or domain scopes within the same web application, as this increases the complexity of the solution and potentially introduces scoping issues.

Session Expiration

In order to minimize the time period an attacker can launch attacks over active sessions and hijack them, it is mandatory to set expiration timeouts for every session, establishing the amount of time a session will remain active. Insufficient session expiration by the web application increases the exposure of other session-based attacks, as for the attacker to be able to reuse a valid session ID and hijack the associated session, it must still be active.

The shorter the session interval is, the lesser the time an attacker has to use the valid session ID. The session expiration timeout values must be set accordingly with the purpose and nature of the web application, and balance security and usability, so that the user can comfortably complete the operations within the web

application without his session frequently expiring.

Both the idle and absolute timeout values are highly dependent on how critical the web application and its data are. Common idle timeouts ranges are 2-5 minutes for high-value applications and 15-30 minutes for low risk applications. Absolute timeouts depend on how long a user usually uses the application. If the application is intended to be used by an office worker for a full day, an appropriate absolute timeout range could be between 4 and 8 hours.

When a session expires, the web application must take active actions to invalidate the session on both sides, client and server. The latter is the most relevant and mandatory from a security perspective.

For most session exchange mechanisms, client side actions to invalidate the session ID are based on clearing out the token value. For example, to invalidate a cookie it is recommended to provide an empty (or invalid) value for the session ID, and set the `Expires` (or `Max-Age`) attribute to a date from the past (in case a persistent cookie is being used): `Set-Cookie: id=; Expires=Friday, 17-May-03 18:45:00 GMT`

In order to close and invalidate the session on the server side, it is mandatory for the web application to take active actions when the session expires, or the user actively logs out, by using the functions and methods offered by the session management mechanisms, such as `HttpSession.invalidate()` (J2EE), `Session.Abandon()` (ASP .NET) or `session_destroy()/unset()` (PHP).

Automatic Session Expiration

Idle Timeout

All sessions should implement an idle or inactivity timeout. This timeout defines the amount of time a session will remain active in case there is no activity in the session, closing and invalidating the session upon the defined idle period since the last HTTP request received by the web application for a given session ID.

The idle timeout limits the chances an attacker has to guess and use a valid session ID from another user. However, if the attacker is able to hijack a given session, the idle timeout does not limit the attacker's actions, as they can generate activity on the session periodically to keep the session active for longer periods of time.

Session timeout management and expiration must be enforced server-side. If the client is used to enforce the session timeout, for example using the session token or other client parameters to track time references (e.g. number of minutes since login time), an attacker could manipulate these to extend the session duration.

Absolute Timeout

All sessions should implement an absolute timeout, regardless of session activity. This timeout defines the maximum amount of time a session can be active, closing and invalidating the session upon the defined absolute period since the given session was initially created by the web application. After invalidating the session, the user is forced to (re)authenticate again in the web application and establish a new session.

The absolute session limits the amount of time an attacker can use a hijacked session and impersonate the victim user.

Renewal Timeout

Alternatively, the web application can implement an additional renewal timeout after which the session ID is automatically renewed, in the middle of the user session, and independently of the session activity and, therefore, of the idle timeout.

After a specific amount of time since the session was initially created, the web application can regenerate a new ID for the user session and try to set it, or renew it, on the client. The previous session ID value would still be valid for some time, accommodating a safety interval, before the client is aware of the new ID and starts using it. At that time, when the client switches to the new ID inside the current session, the application invalidates the previous ID.

This scenario minimizes the amount of time a given session ID value, potentially obtained by an attacker, can be reused to hijack the user session, even when the victim user session is still active. The user session remains alive and open on the legitimate client, although its associated session ID value is transparently renewed periodically during the session duration, every time the renewal timeout expires. Therefore, the renewal timeout complements the idle and absolute timeouts, specially when the absolute timeout value extends significantly over time (e.g. it is an application requirement to keep the user sessions open for long periods of time).

Depending on the implementation, potentially there could be a race condition where the attacker with a still valid previous session ID sends a request before the victim user, right after the renewal timeout has just expired, and obtains first the value for the renewed session ID. At least in this scenario, the victim user might be aware of the attack as her session will be suddenly terminated because her associated session ID is not valid anymore.

Manual Session Expiration

Web applications should provide mechanisms that allow security aware users to actively close their session once they have finished using the web application.

Logout Button

Web applications must provide a visible and easily accessible logout (logoff, exit, or close session) button that is available on the web application header or menu and reachable from every web application resource and page, so that the user can manually close the session at any time. As described in *Session_Expiration* section, the web application must invalidate the session at least on server side.

NOTE: Unfortunately, not all web applications facilitate users to close their current session. Thus, client-side enhancements allow conscientious users to protect their sessions by helping to close them diligently.

Web Content Caching

Even after the session has been closed, it might be possible to access the private or sensitive data exchanged within the session through the web browser cache. Therefore, web applications must use restrictive cache directives for all the web traffic exchanged through HTTP and HTTPS, such as the `Cache-Control` and `Pragma` HTTP headers, and/or equivalent META tags on all or (at least) sensitive web pages.

Independently of the cache policy defined by the web application, if caching web application contents is allowed, the session IDs must never be cached, so it is highly recommended to use the `Cache-Control: no-cache="Set-Cookie, Set-Cookie2"` directive, to allow web clients to cache everything except the session ID (see here).

Additional Client-Side Defenses for Session Management

Web applications can complement the previously described session management defenses with additional countermeasures on the client side. Client-side protections, typically in the form of JavaScript checks and verifications, are not bullet proof and can easily be defeated by a skilled attacker, but can introduce another layer of defense that has to be bypassed by intruders.

Initial Login Timeout

Web applications can use JavaScript code in the login page to evaluate and measure the amount of time since the page was loaded and a session ID was granted. If a login attempt is tried after a specific amount of time, the client code can notify the user that the maximum amount of time to log in has passed and reload the login page, hence retrieving a new session ID.

This extra protection mechanism tries to force the renewal of the session ID pre-authentication, avoiding scenarios where a previously used (or manually set) session ID is reused by the next victim using the same computer, for example, in session fixation attacks.

Force Session Logout On Web Browser Window Close Events

Web applications can use JavaScript code to capture all the web browser tab or window close (or even back) events and take the appropriate actions to close the current session before closing the web browser, emulating that the user has manually closed the session via the logout button.

Disable Web Browser Cross-Tab Sessions

Web applications can use JavaScript code once the user has logged in and a session has been established to force the user to re-authenticate if a new web browser tab or window is opened against the same web application. The web application does not want to allow multiple web browser tabs or windows to share the same session. Therefore, the application tries to force the web browser to not share the same session ID simultaneously between them.

NOTE: This mechanism cannot be implemented if the session ID is exchanged through cookies, as cookies are shared by all web browser tabs/windows.

Automatic Client Logout

JavaScript code can be used by the web application in all (or critical) pages to automatically logout client sessions after the idle timeout expires, for example, by redirecting the user to the logout page (the same resource used by the logout button mentioned previously).

The benefit of enhancing the server-side idle timeout functionality with client-side code is that the user can see that the session has finished due to inactivity, or even can be notified in advance that the session is about to expire through a count down timer and warning messages. This user-friendly approach helps to avoid loss of work in web pages that require extensive input data due to server-side silently expired sessions.

Session Attacks Detection

Session ID Guessing and Brute Force Detection

If an attacker tries to guess or brute force a valid session ID, they need to launch multiple sequential requests against the target web application using different session IDs from a single (or set of) IP address(es). Additionally, if an attacker tries to analyze the predictability of the session ID (e.g. using statistical analysis), they need to launch multiple sequential requests from a single (or set of) IP address(es) against the target web application to gather new valid session IDs.

Web applications must be able to detect both scenarios based on the number of attempts to gather (or use) different session IDs and alert and/or block the offending IP address(es).

Detecting Session ID Anomalies

Web applications should focus on detecting anomalies associated to the session ID, such as its manipulation. The OWASP AppSensor Project provides a framework and methodology to implement built-in intrusion detection capabilities within web applications focused on the detection of anomalies and unexpected behaviors, in the form of detection points and response actions. Instead of using external protection layers, sometimes the business logic details and advanced intelligence are only available from inside the web application, where it is possible to establish multiple session related detection points, such as when an existing cookie is modified or deleted, a new cookie is added, the session ID from another user is reused, or when the user location or User-Agent changes in the middle of a session.

Binding the Session ID to Other User Properties

With the goal of detecting (and, in some scenarios, protecting against) user misbehaviors and session hijacking, it is highly recommended to bind the session ID to other user or client properties, such as the client IP address, User-Agent, or client-based digital certificate. If the web application detects any change or anomaly between these different properties in the middle of an established session, this is a very good indicator of session manipulation and hijacking attempts, and this simple fact can be used to alert and/or terminate the suspicious session.

Although these properties cannot be used by web applications to trustingly defend against session attacks, they significantly increase the web application detection (and protection) capabilities. However, a skilled attacker can bypass these controls by reusing the same IP address assigned to the victim user by sharing the same network (very common in NAT environments, like Wi-Fi hotspots) or by using the same outbound web proxy (very common in corporate environments), or by manually modifying his User-Agent to look exactly as the victim users does.

Logging Sessions Life Cycle: Monitoring Creation, Usage, and Destruction of Session IDs

Web applications should increase their logging capabilities by including information regarding the full life cycle of sessions. In particular, it is recommended to record session related events, such as the creation, renewal, and destruction of session IDs, as well as details about its usage within login and logout operations, privilege level changes within the session, timeout expiration, invalid session activities (when detected), and critical business operations during the session.

The log details might include a timestamp, source IP address, web target resource requested (and involved in a session operation), HTTP headers (including the User-Agent and Referer), GET and POST parameters, error codes and messages, username (or user ID), plus the session ID (cookies, URL, GET, POST…).

Sensitive data like the session ID should not be included in the logs in order to protect the session logs against session ID local or remote disclosure or unauthorized access. However, some kind of session-specific information must be logged in order to correlate log entries to specific sessions. It is recommended to log a salted-hash of the session ID instead of the session ID itself in order to allow for session-specific log correlation without exposing the session ID.

In particular, web applications must thoroughly protect administrative interfaces that allow to manage all the current active sessions. Frequently these are used by support personnel to solve session related issues, or even general issues, by impersonating the user and looking at the web application as the user does.

The session logs become one of the main web application intrusion detection data sources, and can also be used by intrusion protection systems to automatically terminate sessions and/or disable user accounts when (one or many) attacks are detected. If active protections are implemented, these defensive actions must be logged too.

Simultaneous Session Logons

It is the web application design decision to determine if multiple simultaneous logons from the same user are allowed from the same or from different client IP addresses. If the web application does not want to allow simultaneous session logons, it must take effective actions after each new authentication event, implicitly terminating the previously available session, or asking the user (through the old, new or both sessions) about the session that must remain active.

It is recommended for web applications to add user capabilities that allow checking the details of active sessions at any time, monitor and alert the user about concurrent logons, provide user features to remotely terminate sessions manually, and track account activity history (logbook) by recording multiple client details such as IP address, User-Agent, login date and time, idle time, etc.

Session Management WAF Protections

There are situations where the web application source code is not available or cannot be modified, or when the changes required to implement the multiple security recommendations and best practices detailed above imply a full redesign of the web application architecture, and therefore, cannot be easily implemented in the short term.

In these scenarios, or to complement the web application defenses, and with the goal of keeping the web application as secure as possible, it is recommended to use external protections such as Web Application Firewalls (WAFs) that can mitigate the session management threats already described.

Web Application Firewalls offer detection and protection capabilities against session based attacks. On the one hand, it is trivial for WAFs to enforce the usage of security attributes on cookies, such as the `Secure` and `HttpOnly` flags, applying basic rewriting rules on the `Set-Cookie` header for all the web application responses that set a new cookie.

On the other hand, more advanced capabilities can be implemented to allow the WAF to keep track of sessions, and the corresponding session IDs, and apply all kind of protections against session fixation (by renewing the session ID on the client-side when privilege changes are detected), enforcing sticky sessions (by verifying the relationship between the session ID and other client properties, like the IP address or User-Agent), or managing session expiration (by forcing both the client and the web application to finalize the session).

The open-source ModSecurity WAF, plus the OWASP Core Rule Set, provide capabilities to detect and apply security cookie attributes, countermeasures against session fixation attacks, and session tracking features to enforce sticky sessions.

Transport Layer Protection

Introduction

This cheat sheet provides guidance on how to implement transport layer protection for an application using Transport Layer Security (TLS). When correctly implemented, TLS can provides a number of security benefits:

- Confidentiality - protection against an attacker from reading the contents of traffic.
- Integrity - protection against an attacker modifying traffic.

- Replay prevention - protection against an attacker replaying requests against the server.
- Authentication - allowing the client to verify that they are connected to the real server (note that the identity of the *client* is not verified unless client certificates are used).

TLS is used by many other protocols to provide encryption and integrity, and can be used in a number of different ways. This cheatsheet is primarily focused on how to use TLS to protect clients connecting to a web application over HTTPS; although much of the guidance is also applicable to other uses of TLS.

SSL vs TLS

Secure Socket Layer (SSL) was the original protocol that was used to provide encryption for HTTP traffic, in the form of HTTPS. There were two publicly released versions of SSL - versions 2 and 3. Both of these have serious cryptographic weaknesses and should no longer be used.

For various reasons the next version of the protocol (effectively SSL 3.1) was named Transport Layer Security (TLS) version 1.0. Subsequently TLS versions 1.1, 1.2 and 1.3 have been released.

The terms "SSL", "SSL/TLS" and "TLS" are frequently used interchangeably, and in many cases "SSL" is used when referring to the more modern TLS protocol. This cheatsheet will use the term "TLS" except where referring to the legacy protocols.

Server Configuration

Only Support Strong Protocols

The SSL protocols have a large number of weaknesses, and should not be used in any circumstances. General purpose web applications should default to TLS 1.3 (support TLS 1.2 if necessary) with all other protocols disabled. Where it is known that a web server must support legacy clients with unsupported an insecure browsers (such as Internet Explorer 10), it may be necessary to enable TLS 1.0 to provide support.

Where legacy protocols are required, the "TLS_FALLBACK_SCSV" extension should be enabled in order to prevent downgrade attacks against clients.

Note that PCI DSS forbids the use of legacy protocols such as TLS 1.0.

Only Support Strong Ciphers

There are a large number of different ciphers (or cipher suites) that are supported by TLS, that provide varying levels of security. Where possible, only GCM ciphers should be enabled. However, if it is necessary to support legacy clients, then other ciphers may be required.

At a minimum, the following types of ciphers should always be disabled:

- Null ciphers
- Anonymous ciphers
- EXPORT ciphers

See the TLS Cipher String Cheat Sheet for full details on securely configuring ciphers.

Use Strong Diffie-Hellman Parameters

Where ciphers that use the ephemeral Diffie-Hellman key exchange are in use (signified by the "DHE" or "EDH" strings in the cipher name) sufficiently secure Diffie-Hellman parameters (at least 2048 bits) should be used

The following command can be used to generate 2048 bit parameters:

```
bash openssl dhparam 2048 -out dhparam2048.pem
```

The Weak DH website provides guidance on how various web servers can be configured to use these generated parameters.

Disable Compression

TLS compression should be disabled in order to protect against a vulnerability (nicknamed CRIME) which could potentially allow sensitive information such as session cookies to be recovered by an attacker.

Patch Cryptographic Libraries

As well as the vulnerabilities in the SSL and TLS protocols, there have also been a large number of historic vulnerability in SSL and TLS libraries, with Heartbleed being the most well known. As such, it is important to ensure that these libraries are kept up to date with the latest security patches.

Test the Server Configuration

Once the server has been hardened, the configuration should be tested. The OWASP Testing Guide chapter on SSL/TLS Testing contains further information on testing.

There are a number of online tools that can be used to quickly validate the configuration of a server, including:

- SSL Labs Server Test
- CryptCheck
- CypherCraft
- Hardenize
- ImmuniWeb
- Observatory by Mozilla
- Scanigma
- OWASP PurpleTeam `cloud`

Additionally, there are a number of offline tools that can be used:

- O-Saft - OWASP SSL advanced forensic tool
- CipherScan
- CryptoLyzer
- SSLScan - Fast SSL Scanner
- SSLyze
- testssl.sh - Testing any TLS/SSL encryption
- tls-scan
- OWASP PurpleTeam `local`

Certificates

Use Strong Keys and Protect Them

The private key used to generate the cipher key must be sufficiently strong for the anticipated lifetime of the private key and corresponding certificate. The current best practice is to select a key size of at least 2048 bits. Additional information on key lifetimes and comparable key strengths can be found here and in NIST SP 800-57.

The private key should also be protected from unauthorized access using filesystem permissions and other technical and administrative controls.

Use Strong Cryptographic Hashing Algorithms

Certificates should use SHA-256 for the hashing algorithm, rather than the older MD5 and SHA-1 algorithms. These have a number of cryptographic weaknesses, and are not trusted by modern browsers.

Use Correct Domain Names

The domain name (or subject) of the certificate must match the fully qualified name of the server that presents the certificate. Historically this was stored in the `commonName` (CN) attribute of the certificate. However, modern versions of Chrome ignore the CN attribute, and require that the FQDN is in the `subjectAlternativeName` (SAN) attribute. For compatibility reasons, certificates should have the primary FQDN in the CN, and the full list of FQDNs in the SAN.

Additionally, when creating the certificate, the following should be taken into account:

- Consider whether the "www" subdomain should also be included.
- Do not include non-qualified hostnames.
- Do not include IP addresses.
- Do not include internal domain names on externally facing certificates.
  - If a server is accessible using both internal and external FQDNs, configure it with multiple certificates.

Carefully Consider the use of Wildcard Certificates

Wildcard certificates can be convenient, however they violate the principal of least privilege, as a single certificate is valid for all subdomains of a domain (such as *.example.org). Where multiple systems are sharing a wildcard certificate, the likelihood that the private key for the certificate is compromised increases, as the key may be present on multiple systems. Additionally, the value of this key is significantly increased, making it a more attractive target for attackers.

The issues around the use of wildcard certificates are complicated, and there are various other discussions of them online.

When risk assessing the use of wildcard certificates, the following areas should be considered:

- Only use wildcard certificates where there is a genuine need, rather than for convenience.
  - Consider the use of the ACME to allow systems to automatically request and update their own certificates instead.
- Never use a wildcard certificates for systems at different trust levels.
  - Two VPN gateways could use a shared wildcard certificate.
  - Multiple instances of a web application could share a certificate.
  - A VPN gateway and a public webserver **should not** share a wildcard certificate.

    - A public webserver and an internal server **should not** share a wildcard certificate.
- Consider the use of a reverse proxy server which performs TLS termination, so that the wildcard private key is only present on one system.
- A list of all systems sharing a certificate should be maintained to allow them all to be updated if the certificate expires or is compromised.
- Limit the scope of a wildcard certificate by issuing it for a subdomain (such as `*.foo.example.org`), or a for a separate domain.

Use an Appropriate Certification Authority for the Application's User Base

In order to be trusted by users, certificates must be signed by a trusted certificate authority (CA). For Internet facing applications, this should be one of the CAs which are well-known and automatically trusted by operating systems and browsers.

The LetsEncrypt CA provides free domain validated SSL certificates, which are trusted by all major browsers. As such, consider whether there are any benefits to purchasing a certificate from a CA.

For internal applications, an internal CA can be used. This means that the FQDN of the certificate will not be exposed (either to an external CA, or publicly in certificate transparency lists). However, the certificate will only be trusted by users who have imported and trusted the internal CA certificate that was used to sign them.

Use CAA Records to Restrict Which CAs can Issue Certificates

Certification Authority Authorization (CAA) DNS records can be used to define which CAs are permitted to issue certificates for a domain. The records contains a list of CAs, and any CA who is not included in that list should refuse to issue a certificate for the domain. This can help to prevent an attacker from obtaining unauthorized certificates for a domain through a less-reputable CA. Where it is applied to all subdomains, it can also be useful from an administrative perspective by limiting which CAs administrators or developers are able to use, and by preventing them from obtaining unauthorized wildcard certificates.

Always Provide All Needed Certificates

In order to validate the authenticity of a certificate, the user's browser must examine the certificate that was used to sign it and compare it to the list of CAs trusted by their system. In many cases the certificate is not directly signed by a root CA, but is instead signed by an intermediate CA, which is in turn signed by the root CA.

If the user does not know or trust this intermediate CA then the certificate validation will fail, even if the user trusts the ultimate root CA, as they cannot establish a chain of trust between the certificate and the root. In order to avoid this, any intermediate certificates should be provided alongside the main certificate.

Consider the use of Extended Validation Certificates

Extended validation (EV) certificates claim to provide a higher level of verification of the entity, as they perform checks that the requestor is a legitimate legal entity, rather than just verifying the ownership of the domain name like normal (or "Domain Validated") certificates. This can effectively be viewed as the difference between "This site is really run by Example Company Inc." vs "This domain is really example.org".

Historically these displayed differently in the browser, often showing the company name or a green icon or background in the address bar. However, as of 2019 both Chrome and Firefox have announced that they will be removing these indicators, as they do not believe that EV certificates provide any additional protection.

There is no security downside to the use of EV certificates. However, as they are significantly more expensive than domain validated certificates, an assessment should be made to determine whether they provide any additional value

Application

Use TLS For All Pages

TLS should be used for all pages, not just those that are considered sensitive such as the login page. If there are any pages that do not enforce the use of TLS, these could give an attacker an opportunity to sniff sensitive information such as session tokens, or to inject malicious JavaScript into the responses to carry out other attacks against the user.

For public facing applications, it may be appropriate to have the web server listening for unencrypted HTTP connections on port 80, and then immediately redirecting them with a permanent redirect (HTTP 301) in order to provide a better experience to users who manually type in the domain name. This should then be supported with the HTTP Strict Transport Security (HSTS) header to prevent them accessing the site over HTTP in the future.

Do Not Mix TLS and Non-TLS Content

A page that is available over TLS should not include any resources (such as JavaScript or CSS) files which are loaded over unencrypted HTTP. These unencrypted resources could allow an attacker to sniff session cookies or inject malicious code into the page. Modern browsers will also block attempts to load active content over unencrypted HTTP into secure pages.

Use the "Secure" Cookie Flag

All cookies should be marked with the "Secure" attribute, which instructs the browser to only send them over encrypted HTTPS connections, in order to prevent them from being sniffed from an unencrypted HTTP connection. This is important even if the website does not listen on HTTP (port 80), as an attacker performing an active man in the middle attack could present a spoofed webserver on port 80 to the user in order to steal their cookie.

Prevent Caching of Sensitive Data

Although TLS provides protection of data while it is in transit, it does not provide any protection for data once it has reached the requesting system. As such, this information may be stored in the cache of the user's browser, or by any intercepting proxies which are configured to perform TLS decryption.

Where sensitive data is returned in responses, HTTP headers should be used to instruct the browser and any proxy servers not to cache the information, in order to prevent it being stored or returned to other users. This can be achieved by setting the following HTTP headers in the response:

```
text Cache-Control: no-cache, no-store, must-revalidate Pragma: no-cache Expires: 0
```

Use HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) instructs the user's browser to always request the site over HTTPS, and also prevents the user from bypassing certificate warnings. See the HTTP Strict Transport Security cheatsheet for further information on implementing HSTS.

Consider the use of Client-Side Certificates

In a typical configuration, TLS is used with a certificate on the server so that the client is able to verify the identity of the server, and to provide an encrypted connection between them. However, there are two main weaknesses with this approach:

- The server does not have any mechanism to verify the identity of the client
- The connection can be intercepted by an attacker who is able to obtain a valid certificate for the domain.
  - This is most commonly used by businesses to carry out inspection of TLS traffic by installing a trusted CA certificate on there client systems.

Client certificates address both of these issues by requiring that the client proves their identity to the server with their own certificate. This not only provides strong authentication of the identity of the client, but also prevents an intermediate party from performing TLS decryption, even if they have trusted CA certificate on the client system.

Client certificates are rarely used on public systems due to a number of issues:

- Issuing and managing client certificates introduces significant administrative overheads.
- Non-technical users may struggle to install client certificates.
- TLS decryption used by many organisations will cause client certificate authentication to fail.

However, they should be considered for high-value applications or APIs, especially where there are a small number of technically sophisticated users, or where all users are part of the same organisation.

Consider Using Public Key Pinning

Public key pinning can be used to provides assurance that the server's certificate is not only valid and trusted, but also that it matches the certificate expected for the server. This provides protection against an attacker who is able to obtain a valid certificate, either by exploiting a weakness in the validation process, compromising a trusted certificate authority, or having administrative access to the client.

Public key pinning was added to browsers in the HTTP Public Key Pinning (HPKP) standard. However, due to a number of issues, it has subsequently been deprecated and is no longer recommended or supported by modern browsers.

However, public key pinning can still provide security benefits for mobile applications, thick clients and server-to-server communication. This is discussed in further detail in the Pinning Cheat Sheet.

Related Articles

- OWASP - TLS Cipher String Cheat Sheet
- OWASP - Testing for SSL-TLS, and OWASP Guide to Cryptography
- OWASP - Application Security Verification Standard (ASVS) - Communication Security Verification Requirements (V9)
- Mozilla - Mozilla Recommended Configurations
- NIST - SP 800-52 Rev. 1 Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations
- NIST - NIST SP 800-57 Recommendation for Key Management, Revision 3, Public DRAFT
- NIST - SP 800-95 Guide to Secure Web Services
- IETF - RFC 5280 Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
- IETF - RFC 2246 The Transport Layer Security (TLS) Protocol Version 1.0 (JAN 1999)
- IETF - RFC 4346 The Transport Layer Security (TLS) Protocol Version 1.1 (APR 2006)
- IETF - RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2 (AUG 2008)
- Bettercrypto - Applied Crypto Hardening: HOWTO for secure crypto settings of the most common services)

Input Validation

Goals of Input Validation

Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components. Input validation should happen as early as possible in the data flow, preferably as soon as the data is received from the external party.

Data from all potentially untrusted sources should be subject to input validation, including not only Internet-facing web clients but also backend feeds over extranets, from suppliers, partners, vendors or regulators, each of which may be compromised on their own and start sending malformed data.

Input Validation should not be used as the *primary* method of preventing XSS, SQL Injection and other attacks which are covered in respective cheat sheets but can significantly contribute to reducing their impact if implemented properly.

Input validation strategies

Input validation should be applied on both **syntactical** and **Semantic** level.

Syntactic validation should enforce correct syntax of structured fields (e.g. SSN, date, currency symbol).

Semantic validation should enforce correctness of their *values* in the specific business context (e.g. start date is before end date, price is within expected range).

It is always recommended to prevent attacks as early as possible in the processing of the user's (attacker's) request. Input validation can be used to detect unauthorized input before it is processed by the application.

Implementing input validation

Input validation can be implemented using any programming technique that allows effective enforcement of syntactic and semantic correctness, for example:

- Data type validators available natively in web application frameworks (such as Django Validators, Apache Commons Validators etc).
- Validation against JSON Schema and XML Schema (XSD) for input in these formats.
- Type conversion (e.g. `Integer.parseInt()` in Java, `int()` in Python) with strict exception handling
- Minimum and maximum value range check for numerical parameters and dates, minimum and maximum length check for strings.
- Array of allowed values for small sets of string parameters (e.g. days of week).
- Regular expressions for any other structured data covering the whole input string (`^...$`) and **not** using "any character" wildcard (such as `.` or `\S`)

Allow list vs block list

It is a common mistake to use block list validation in order to try to detect possibly dangerous characters and patterns like the apostrophe `'` character, the string `1=1`, or the `<script>` tag, but this is a massively flawed approach as it is trivial for an attacker to bypass such filters.

Plus, such filters frequently prevent authorized input, like `O'Brian`, where the `'` character is fully legitimate. For more information on XSS filter evasion please see this wiki page.

Allow list validation is appropriate for all input fields provided by the user. Allow list validation involves defining exactly what IS authorized, and by definition, everything else is not authorized.

If it's well structured data, like dates, social security numbers, zip codes, email addresses, etc. then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input.

If the input field comes from a fixed set of options, like a drop down list or radio buttons, then the input needs to match exactly one of the values offered to the user in the first place.

Validating free-form Unicode text

Free-form text, especially with Unicode characters, is perceived as difficult to validate due to a relatively large space of characters that need to be allowed.

It's also free-form text input that highlights the importance of proper context-aware output encoding and quite clearly demonstrates that input validation is **not** the primary safeguards against Cross-Site Scripting. If your users want to type apostrophe `'` or less-than sign < in their comment field, they might have perfectly legitimate reason for that and the application's job is to properly handle it throughout the whole life cycle of the data.

The primary means of input validation for free-form text input should be:

- **Normalization:** Ensure canonical encoding is used across all the text and no invalid characters are present.
- **Character category allow-listing:** Unicode allows listing categories such as "decimal digits" or "letters" which not only covers the Latin alphabet but also various other scripts used globally (e.g. Arabic, Cyrillic, CJK ideographs etc).
- **Individual character allow-listing:** If you allow letters and ideographs in names and also want to allow apostrophe `'` for Irish names, but don't want to allow the whole punctuation category.

References:

- Input validation of free-form Unicode text in Python

Regular expressions

Developing regular expressions can be complicated, and is well beyond the scope of this cheat sheet.

There are lots of resources on the internet about how to write regular expressions, including this site and the OWASP Validation Regex Repository.

When designing regular expression, be aware of RegEx Denial of Service (ReDoS) attacks. These attacks cause a program using a poorly designed Regular Expression to operate very slowly and utilize CPU resources for a very long time.

In summary, input validation should:

- Be applied to all input data, at minimum.
- Define the allowed set of characters to be accepted.
- Define a minimum and maximum length for the data (e.g. `{1,25}`).

Allow List Regular Expression Examples

Validating a U.S. Zip Code (5 digits plus optional -4)

```
text ^\d{5}(-\d{4})?$
```

Validating U.S. State Selection From a Drop-Down Menu

```
text ^(AA|AE|AP|AL|AK|AS|AZ|AR|CA|CO|CT|DE|DC|FM|FL|GA|GU| HI|ID|IL|IN|IA|KS|KY|LA|ME|MH|MD|MA|MI|MN|MS|MO|MT|NE|
NV|NH|NJ|NM|NY|NC|ND|MP|OH|OK|OR|PW|PA|PR|RI|SC|SD|TN| TX|UT|VT|VI|VA|WA|WV|WI|WY)$
```

Java Regex Usage Example:

Example validating the parameter "zip" using a regular expression.

```java private static final Pattern zipPattern = Pattern.compile("^\d{5}(-\d{4})?$");

public void doPost( HttpServletRequest request, HttpServletResponse response) { try {  String zipCode = request.getParameter( "zip" );
    if ( !zipPattern.matcher( zipCode ).matches()  {  throw new YourValidationException( "Improper zipcode format." );  }
    // do what you want here, after its been validated ..  } catch(YourValidationException e ) {
    response.sendError( response.SC_BAD_REQUEST, e.getMessage() );  } } ```
```

Some Allow list validators have also been predefined in various open source packages that you can leverage. For example:

- Apache Commons Validator

Client Side vs Server Side Validation

Be aware that any JavaScript input validation performed on the client can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed on the client is also performed on the server.

Validating Rich User Content

It is very difficult to validate rich content submitted by a user. For more information, please see the XSS cheatsheet on Sanitizing HTML Markup with a Library Designed for the Job.

Preventing XSS and Content Security Policy

All user data controlled must be encoded when returned in the HTML page to prevent the execution of malicious data (e.g. XSS). For example `<script>` would be returned as `&lt;script&gt;`

The type of encoding is specific to the context of the page where the user controlled data is inserted. For example, HTML entity encoding is appropriate for data placed into the HTML body. However, user data placed into a script would need JavaScript specific output encoding.

Detailed information on XSS prevention here: OWASP XSS Prevention Cheat Sheet

File Upload Validation

Many websites allow users to upload files, such as a profile picture or more. This section helps provide that feature securely.

Check the File Upload Cheat Sheet.

Upload Verification

- Use input validation to ensure the uploaded filename uses an expected extension type.
- Ensure the uploaded file is not larger than a defined maximum file size.
- If the website supports ZIP file upload, do validation check before unzip the file. The check includes the target path, level of compress, estimated unzip size.

Upload Storage

- Use a new filename to store the file on the OS. Do not use any user controlled text for this filename or for the temporary filename.
- When the file is uploaded to web, it's suggested to rename the file on storage. For example, the uploaded filename is *test.JPG*, rename it to *JAI1287uaisdjhf.JPG* with a random filename. The purpose of doing it to prevent the risks of direct file access and ambiguous filename to evalide the filter, such as `test.jpg;.asp or /../../../../../test.jpg`.
- Uploaded files should be analyzed for malicious content (anti-malware, static analysis, etc).
- The file path should not be able to specify by client side. It's decided by server side.

Public Serving of Uploaded Content

- Ensure uploaded images are served with the correct content-type (e.g. image/jpeg, application/x-xpinstall)

Beware of "special" files

The upload feature should be using an allow-list approach to only allow specific file types and extensions. However, it is important to be aware of the following file types that, if allowed, could result in security vulnerabilities:

- **crossdomain.xml / clientaccesspolicy.xml:** allows cross-domain data loading in Flash, Java and Silverlight. If permitted on sites with authentication this can permit cross-domain data theft and CSRF attacks. Note this can get pretty complicated depending on the specific plugin version in question, so its best to just prohibit files named "crossdomain.xml" or "clientaccesspolicy.xml".
- **.htaccess** and **.htpasswd:** Provides server configuration options on a per-directory basis, and should not be permitted. See HTACCESS documentation.
- Web executable script files are suggested not to be allowed such as `aspx, asp, css, swf, xhtml, rhtml, shtml, jsp, js, pl, php, cgi`.

Image Upload Verification

- Use image rewriting libraries to verify the image is valid and to strip away extraneous content.
- Set the extension of the stored image to be a valid image extension based on the detected content type of the image from image processing (e.g. do not just trust the header from the upload).
- Ensure the detected content type of the image is within a list of defined image types (jpg, png, etc)

Email Address Validation

Syntactic Validation

The format of email addresses is defined by RFC 5321, and is far more complicated than most people realise. As an example, the following are all considered to be valid email addresses:

- `"><script>alert(1);</script>"@example.org`
- `user+subaddress@example.org`
- `user@[IPv6:2001:db8::1]`
- `" "@example.org`

Properly parsing email addresses for validity with regular expressions is very complicated, although there are a number of publicly available documents on regex.

The biggest caveat on this is that although the RFC defines a very flexible format for email addresses, most real world implementations (such as mail servers) use a far more restricted address format, meaning that they will reject addresses that are *technically* valid. Although they may be technically correct, these addresses are of little use if your application will not be able to actually send emails to them.

As such, the best way to validate email addresses is to perform some basic initial validation, and then pass the address to the mail server and catch the exception if it rejects it. This means that any the application can be confident that its mail server can send emails to any addresses it accepts. The initial validation could be as simple as:

- The email address contains two parts, separated with an `@` symbol.
- The email address does not contain dangerous characters (such as backticks, single or double quotes, or null bytes).
  - Exactly which characters are dangerous will depend on how the address is going to be used (echoed in page, inserted into database, etc).
- The domain part contains only letters, numbers, hyphens (`-`) and periods (`.`).
- The email address is a reasonable length:
  - The local part (before the `@`) should be no more than 63 characters.
  - The total length should be no more than 254 characters.

Semantic Validation

Semantic validation is about determining whether the email address is correct and legitimate. The most common way to do this is to send an email to the user, and require that they click a link in the email, or enter a code that has been sent to them. This provides a basic level of assurance that:

- The email address is correct.

- The application can successfully send emails to it.
- The user has access to the mailbox.

The links that are sent to users to prove ownership should contain a token that is:

- At least 32 characters long.
- Generated using a secure source of randomness.
- Single use.
- Time limited (e.g, expiring after eight hours).

After validating the ownership of the email address, the user should then be required to authenticate on the application through the usual mechanism.

Disposable Email Addresses

In some cases, users may not want to give their real email address when registering on the application, and will instead provide a disposable email address. These are publicly available addresses that do not require the user to authenticate, and are typically used to reduce the amount of spam received by users' primary email addresses.

Blocking disposable email addresses is almost impossible, as there are a large number of websites offering these services, with new domains being created every day. There are a number of publicly available lists and commercial lists of known disposable domains, but these will always be incomplete.

If these lists are used to block the use of disposable email addresses then the user should be presented with a message explaining why they are blocked (although they are likely to simply search for another disposable provider rather than giving their legitimate address).

If it is essential that disposable email addresses are blocked, then registrations should only be allowed from specifically-allowed email providers. However, if this includes public providers such as Google or Yahoo, users can simply register their own disposable address with them.

Sub-Addressing

Sub-addressing allows a user to specify a *tag* in the local part of the email address (before the @ sign), which will be ignored by the mail server. For example, if that `example.org` domain supports sub-addressing, then the following email addresses are equivalent:

- `user@example.org`
- `user+site1@example.org`
- `user+site2@example.org`

Many mail providers (such as Microsoft Exchange) do not support sub-addressing. The most notable provider who does is Gmail, although there are many others that also do.

Some users will use a different *tag* for each website they register on, so that if they start receiving spam to one of the sub-addresses they can identify which website leaked or sold their email address.

Because it could allow users to register multiple accounts with a single email address, some sites may wish to block sub-addressing by stripping out everything between the + and @ signs. This is not generally recommended, as it suggests that the website owner is either unaware of sub-addressing or wishes to prevent users from identifying them when they leak or sell email addresses. Additionally, it can be trivially bypassed by using disposable email addresses, or simply registering multiple email accounts with a trusted provider.

User Privacy Protection

Introduction

This OWASP Cheat Sheet introduces mitigation methods that web developers may utilize in order to protect their users from a vast array of potential threats and aggressions that might try to undermine their privacy and anonymity. This cheat sheet focuses on privacy and anonymity threats that users might face by using online services, especially in contexts such as social networking and communication platforms.

Guidelines

Strong Cryptography

Any online platform that handles user identities, private information or communications must be secured with the use of strong cryptography. User communications must be encrypted in transit and storage. User secrets such as passwords must also be protected using strong, collision-resistant hashing algorithms with increasing work factors, in order to greatly mitigate the risks of exposed credentials as well as proper integrity control.

To protect data in transit, developers must use and adhere to TLS/SSL best practices such as verified certificates, adequately protected private keys, usage of strong ciphers only, informative and clear warnings to users, as well as sufficient key lengths. Private data must be encrypted in storage using keys with sufficient lengths and under strict access conditions, both technical and procedural. User credentials must be hashed regardless of whether or not they are encrypted in storage.

For detailed guides about strong cryptography and best practices, read the following OWASP references:

1. Cryptographic Storage Cheat Sheet.
2. Authentication Cheat Sheet.
3. Transport Layer Protection Cheat Sheet.
4. Guide to Cryptography.
5. Testing for TLS/SSL.

Support HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) is an HTTP header set by the server indicating to the user agent that only secure (HTTPS) connections are accepted, prompting the user agent to change all insecure HTTP links to HTTPS, and forcing the compliant user agent to fail-safe by refusing any TLS/SSL connection that is not trusted by the user.

HSTS has average support on popular user agents, such as Mozilla Firefox and Google Chrome. Nevertheless, it remains very useful for users who are in consistent fear of spying and Man in the Middle Attacks.

If it is impractical to force HSTS on all users, web developers should at least give users the choice to enable it if they wish to make use of it.

For more details regarding HSTS, please visit:

1. HTTP Strict Transport Security in Wikipedia.
2. IETF for HSTS RFC.
3. OWASP Appsec Tutorial Series - Episode 4: Strict Transport Security.

Digital Certificate Pinning

Certificate Pinning is the practice of hardcoding or storing a predefined set of information (usually hashes) for digital certificates/public keys in the user agent (be it web browser, mobile app or browser plugin) such that only the predefined certificates/public keys are used for secure communication, and all others will fail, even if the user trusted (implicitly or explicitly) the other certificates/public keys.

Some advantages for pinning are:

- In the event of a CA compromise, in which a compromised CA trusted by a user can issue certificates for any domain, allowing evil perpetrators to eavesdrop on users.
- In environments where users are forced to accept a potentially-malicious root CA, such as corporate environments or national PKI schemes.
- In applications where the target demographic may not understand certificate warnings, and is likely to just allow any invalid certificate.

For details regarding certificate pinning, please refer to the following:

1. OWASP Certificate Pinning Cheat Sheet.
2. Public Key Pinning Extension for HTTP RFC.
3. Securing the SSL channel against man-in-the-middle attacks: Future technologies - HTTP Strict Transport Security and Pinning of Certs, by Tobias Gondrom.

Panic Modes

A panic mode is a mode that threatened users can refer to when they fall under direct threat to disclose account credentials.

Giving users the ability to create a panic mode can help them survive these threats, especially in tumultuous regions around the world. Unfortunately many users around the world are subject to types of threats that most web developers do not know of or take into account.

Examples of panic modes are modes where distressed users can delete their data upon threat, log into fake inboxes/accounts/systems, or invoke triggers to backup/upload/hide sensitive data.

The appropriate panic mode to implement differs depending on the application type. A disk encryption software such as VeraCrypt might implement a panic mode that starts up a fake system partition if the user entered their distressed password.

Email providers might implement a panic mode that hides predefined sensitive emails or contacts, allowing reading innocent email messages only, usually as defined by the user, while preventing the panic mode from overtaking the actual account.

An important note about panic modes is that they must not be easily discoverable, if at all. An adversary inside a victim's panic mode must not have any way, or as few possibilities as possible, of finding out the truth. This means that once inside a panic mode, most non-sensitive normal operations must be allowed to continue (such as sending or receiving email), and that further panic modes must be possible to create from inside the original panic mode (If the adversary tried to create a panic mode on a victim's panic mode and failed, the adversary would know they were already inside a panic mode, and might attempt to hurt the victim).

Another solution would be to prevent panic modes from being generated from the user account, and instead making it a bit harder to spoof by adversaries. For example it could be only created Out Of Band, and adversaries must have no way to know a panic mode already exists for that particular account.

The implementation of a panic mode must always aim to confuse adversaries and prevent them from reaching the actual accounts/sensitive data of the victim, as well as prevent the discovery of any existing panic modes for a particular account.

For more details regarding VeraCrypt's hidden operating system mode, please refer to:

- VeraCrypt Hidden Operating System.

Remote Session Invalidation

In case user equipment is lost, stolen or confiscated, or under suspicion of cookie theft; it might be very beneficial for users to able to see view their current online sessions and disconnect/invalidate any suspicious lingering sessions, especially ones that belong to stolen or confiscated devices. Remote session invalidation can also helps if a user suspects that their session details were stolen in a Man-in-the-Middle attack.

For details regarding session management, please refer to:

- OWASP Session Management Cheat Sheet.

Allow Connections from Anonymity Networks

Anonymity networks, such as the Tor Project, give users in tumultuous regions around the world a golden chance to escape surveillance, access information or break censorship barriers. More often than not, activists in troubled regions use such networks to report injustice or send uncensored information to the rest of the world, especially mediums such as social networks, media streaming websites and email providers.

Web developers and network administrators must pursue every avenue to enable users to access services from behind such networks, and any policy made against such anonymity networks need to be carefully re-evaluated with respect to impact on people around the world.

If possible, application developers should try to integrate or enable easy coupling of their applications with these anonymity networks, such as supporting SOCKS proxies or integration libraries (e.g. OnionKit for Android).

For more information about anonymity networks, and the user protections they provide, please refer to:

1. The Tor Project.
2. I2P Network.
3. OnionKit: Boost Network Security and Encryption in your Android Apps.

Prevent IP Address Leakage

Preventing leakage of user IP addresses is of great significance when user protection is in scope. Any application that hosts external third-party content, such as avatars, signatures or photo attachments; must take into account the benefits of allowing users to block third-party content from being loaded in the application page.

If it was possible to embed 3rd-party, external domain images, for example, in a user's feed or timeline; an adversary might use it to discover a victim's real IP address by hosting it on his domain and watch for HTTP requests for that image.

Many web applications need user content to operate, and this is completely acceptable as a business process; however web developers are advised to consider giving users the option of blocking external content as a precaution. This applies mainly to social networks and forums, but can also apply to web-based e-mail, where images can be embedded in HTML-formatted emails.

A similar issue exists in HTML-formatted emails that contain third-party images, however most email clients and providers block loading of third-party content by default; giving users better privacy and anonymity protection.

Honesty & Transparency

If the web application cannot provide enough legal or political protections to the user, or if the web application cannot prevent misuse or disclosure of sensitive information such as logs, the truth must be told to the users in a clear understandable form, so that users can make an educated choice about whether or not they should use that particular service.

If it doesn't violate the law, inform users if their information is being requested for removal or investigation by external entities.

Honesty goes a long way towards cultivating a culture of trust between a web application and its users, and it allows many users around the world to weigh their options carefully, preventing harm to users in various contrasting regions around the world.

More insight regarding secure logging can be found at:

- OWASP Logging Cheat Sheet

Cryptography

An architectural decision must be made to determine the appropriate method to protect data at rest. There are such wide varieties of products, methods and mechanisms for cryptographic storage.The general practices and required minimum key length depending on the scenario listed below:

Good practices

- Cryptographic algorithms are up to date and in-line with industry standards. This includes, but is not limited to outdated block ciphers (e.g. DES), stream ciphers (e.g. RC4), as well as hash functions (e.g. MD5) and broken random number generators like Dual_EC_DRBG (even if they are NIST certified). All of these should be marked as insecure and should not be used and removed from the application and server.
- Key lengths are in-line with industry standards and provide protection for sufficient amount of time. A comparison of different key lengths and protection they provide taking into account Moore's law is available online.
- Cryptographic means are not mixed with each other: e.g. you do not sign with a public key, or try to reuse a keypair used for a signature to do encryption.
- Cryptographic parameters are well defined within reasonable range. This includes, but is not limited to: cryptographic salt, which should be at least the same length as hash function output, reasonable choice of password derivation function and iteration count (e.g. PBKDF2, scrypt or bcrypt), IVs being random and unique, fit-for-purpose block encryption modes (e.g. ECB should not be used, except specific cases), key management being done properly (e.g. 3DES should have three independent keys) and so on.

Recommended Algorithms

- Confidentiality algorithms: AES-GCM-256 or ChaCha20-Poly1305;
- Integrity algorithms: SHA-256, SHA-384, SHA-512, Blake2;
- Digital signature algorithms: RSA (3072 bits and higher), ECDSA with NIST P-384;
- Key establishment algorithms: RSA (3072 bits and higher), DH (3072 bits or higher), ECDH with NIST P-384;
- Application must be capable of using end-to-end encryption via SSL / TLS in relation to sensitive data in transit and at rest.

Additionally, you should always rely on secure hardware (if available) for storing encryption keys, performing cryptographic operations, etc.

Secure Cryptographic Storage Design

- All protocols and algorithms for authentication and secure communication should be well vetted by the cryptographic community.
- Ensure certificates are properly validated against the hostnames users whom they are meant for.
- Avoid using wildcard certificates unless there is a business need for it
- Maintain a cryptographic standard to ensure that the developer community knows about the approved ciphersuits for network security protocols, algorithms, permitted use, cryptoperiods and Key Management.
- Only store sensitive data that you need

Use strong approved Authenticated Encryption

CCM or GCM are approved Authenticated Encryption modes based on AES algorithm.

Use strong approved cryptographic algorithms

- Do not implement an existing cryptographic algorithm on your own, no matter how easy it appears. * Instead, use widely accepted algorithms and widely accepted implementations.
- Only use approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing.
- Do not use weak algorithms, such as MD5 or SHA1.
- Avoid hashing for password storage,instead use Argon2, PBKDF2, bcrypt or scrypt.
- See NIST approved algorithms or ISO TR 14742 "Recommendations on Cryptographic Algorithms or Algorithms", key size and parameters by European Union Agency for Network and Information Security.
- If a password is being used to protect keys then the password strength should be sufficient for the strength of the keys it is protecting. * When 3DES is used, ensure K1 != K2 != K3, and the minimum key length must be 192 bits .
- Do not use ECB mode for encrypting lots of data (the other modes are better because they chain the blocks of data together to improve the data security).

Use strong random numbers

- Ensure that all random numbers, especially those used for cryptographic parameters (keys, IV's, MAC tags), random file names, random GUIDs, and random strings are generated in a cryptographically strong fashion.
- Ensure that random algorithms are seeded with sufficient entropy.
- Tools like NIST RNG Test tool can be used to comprehensively assess the quality of a Random Number Generator by reading e.g. 128MB of data from the RNG source and then assessing its randomness properties with the tool.

The following libraries are considered weak random numbers generators and should not be used:

- C library: random(), rand(), use getrandom(2) instead;
- Java library: java.util.Random() instead use java.security.SecureRandom instead.

For secure random number generation, refer to NIST SP 800-90A. CTR-DRBG, HASH-DRBG, HMAC-DRBG are recommended.

https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cryptographic_Storage_Cheat_Sheet.md

Application Regular Updates

Mobile devices and platforms, such as, for example, smartphones, typically provide the capability for operating system (OS), firmware (FW) and applications updates or re-installations with reduced user involvement. The user involvement may often be limited to clicking an icon or accepting an agreement. While this reduced level of involvement may provide convenience and an improved user experience, it fails to address the issue of secure user authentication.

Mobile devices and platforms, such as smartphones, typically provide features for operating system (OS), firmware (FW) upgrades, and applications or reinstallations with reduced user engagement. User engagement may be limited to clicking an icon or accepting a contract. While this reduced level of engagement can provide convenience and enhance the user experience, it does not address the issue of secure user authentication. Thus, it is necessary to create a secure channel that provides confidentiality, integrity, authentication and data updating.

Requirements for a secure software update

Data Confidentiality: the contents of transmitted data should be kept confidential. This also includes software updates. Thus, secure channels between the mobile device and the network manager must be set up. The standard approach to keep sensitive data secret is to encrypt the data with a key that is shared only between the intended receivers;

Data integrity: it must be possible to ensure that data packets have not been modified in transit. For mobile devices, control requests, and software updates it is critically important to verify that the contents in the packets have not been tampered with;

Data Authentication: To prevent an attacker from injecting packets it is important to make sure that the receiver can verify the sender of the packets. Data authentication ensures this property such that the receiver can verify that the received packets really are from the claimed sender. For example, for software updates, data authentication is needed such that the device can verify that the received software comes from a trusted source. Data authentication can be achieved using a MAC or Digital Signature;

Data Freshness: to protect against replay attacks, e.g., during the key establishment phase, the protocol must ensure that the messages are fresh. Data freshness ensures the security property that the data is recent and that an attacker is not replaying old data.

Third-Party Applications

Many social networks also offer the possibility to create additional applications that extend the functionality of the network. The two major platforms for such applications are the Facebook Platform and Open Social. While applications designed for the Facebook Platform can only be executed in Facebook, Open Social is a combined effort to allow developers to run their applications on any social network that supports the Open Social platform (e.g., MySpace and Orkut).

Requirements for a secure third-party applications

- Secure Computation: No external entity should be able to observe or interfere with the computation performed by trustworthy remote entity (TRE). Since the adversary has physical access to the platform, the confidentiality and integrity must be protected from untrusted sofware on the same platform;
- Secure Communication: Communication between the TRE and other parties must be confidential and integrity-prtected;
- Strong Attestation: The root of trust used for attestation must be trusted by all parties. The attestation mus be unambiguously linked to the communicating entity (authenticity) to avoid masquerading attacks, and must concey the current state of the system (freshness) to prevent replay attacks.

Apps that process or query sensitive information should run in a trusted and secure environment. To create this environment, the app can check the device for the following:

- PIN - or password-protected device locking;
- Recent Mobile Plataform or OS version;
- USB Debugging activation;
- Device encryption;
- Device rooting (see also "Testing Root Detection").