About this Project

Insecure software is undermining our financial, healthcare, defense, energy, and other critical infrastructure worldwide. As our digital, global infrastructure gets increasingly complex and interconnected, the difficulty of achieving application security increases exponentially. We can no longer afford to tolerate relatively simple security problems.

Aim & Objective

The goal of the **OWASP Top 10 Proactive Controls project** (OPC) is to raise awareness about application security by describing the most important areas of concern that software developers must be aware of. We encourage you to use the OWASP Proactive Controls to get your developers started with application security. Developers can learn from the mistakes of other organizations. We hope that the OWASP Proactive Controls is useful to your efforts in building secure software.

Copyright and Licence

Project Leaders (in alphabetical order)

- Andreas Happe, connect through linkedin, github, twitter/x
- Jim Manico, connect through linkedin, github, twitter/x
- Katy Anton , connect through linkedin, github, twitter/x

Contributors

- Chris Romeo
- Jasmin Mair
- Abdessamad Temmar
- Carl Sampson
- Eyal Estrin
- Israel Chorzevski

About OWASP

The *Open Web Application Security Project* (OWASP) is a 501c3 non for profit educational charity dedicated to enabling organizations to design, develop, acquire, operate, and maintain secure software. All OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. We can be found at www.owasp.org.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, cost effective information about application security.

OWASP is not affiliated with any technology company. Similar to many open source software projects, OWASP produces many types of materials in a collaborative and open way. The OWASP Foundation is a not-for-profit entity that ensures the project's long-term success.

Related OWASP Projects

OWASP is a volunteer-driven organization. Those volunteers contributed many useful documents, and this section points to some related OWASP documents and projects:

- The best-known OWASP document is the OWASP Top 10. They detail the most common web application vulnerabilities and are also the base for this document. In contrast, this document is focused on defensive techniques and controls as opposed to risks. Each control in this document will map to one or more items in the risk-based OWASP Top 10. This mapping information is included at the end of each control description.
- The OWASP ASVS: The OWASP Application Security Verification Standard (ASVS) is a catalog of available security requirements and verification criteria. OWASP ASVS can be a source of detailed security requirements for development teams. Security requirements are categorized into different buckets based on a shared higher order security function. For example, the ASVS contains categories such as authentication, access control, error handling / logging, and web services. Each category contains a collection of requirements that represent the best practices for that category drafted as verifiable statements.
- OWASP SAMM Software Assurance Maturity Model (SAMM) is an open framework to help organizations implement a strategy for maturing the software security tailored to the specific risks of the organization. . [SAMM] (https://owaspsamm.org/about/) supports the complete software lifecycle and can be used to identify what
- Threat Modeling is an important part of secure application development, which can help identify potential security threats, derive security requirements, and tailor security controls to prevent potential threats. Successful use of security requirements involves four steps: discovery, documentation, implementation, and verification of the correct implementation of the functionality within an application. Threat modelling is one way to derive security requirements. Other sources are: industry standards, applicable laws, history of past vulnerabilities. Modeling tools, like OWASP Threat Dragon can be used to create threat model diagrams as part of a secure development lifecycle.

It is important to notice that this document primarily focuses on web applications, but other Top 10s could apply to your application, too. Examples of those are: - OWASP API Top 10 - OWASP Mobile Application Top 10

Introduction

For years, developers have suffered through introducing the same security issues into the things they build. The most common issues, which have existed for decades, have been documented by the OWASP Top Ten. Many of the issues in the earliest version still exist in some form today. A mechanism is needed to counter these challenges, and that mechanism is proactive controls.

Proactive controls are a catalog of better practices, a set of items developers can embrace and implement in their code bases to avoid many common security issues. Proactive controls provide positive patterns to implement solutions considered secure by design.

Imagine working on a web-based software, releasing a new version, and then getting reports that it contained a security vulnerability that is now being exploited in the wild. You are now forced to act reactively: analyzing the vulnerability, fixing it, creating a new software release, and getting it to your users. All this is tedious and a bit awkward, esp. When the security vulnerability was critical.

Proactive controls prevent this by focusing on the development itself. Their idea is to prevent common vulnerabilities during an application's inception so that those tedious and embarrassing bug fixes can be avoided altogether. Common knowledge is that a proactive approach will save resources and money in the long run.

Please note that while complying with best proactive practices will reduce the chance of a vulnerability being in your code, there is no guarantee that code is security-bug free. And that's okay: You have to break an egg to make an omelet — the only way of not introducing any security bugs is not to code at all. We accept that while striving to prevent as many security issues as possible.

The OWASP Top 10 Proactive Controls 2024

The OWASP Top 10 Proactive Controls 2024 is a list of security techniques every software architect and developer should know and heed. The main goal of this document is to provide concrete, practical guidance that helps developers build secure software. These techniques should be applied proactively at the early stages of software development to ensure maximum effectiveness.

How to Use this Document

This document's main purpose is to provide a solid foundation of topics to help drive introductory software security developer training. To be effective, these controls should be used consistently and thoroughly throughout all applications.

However, this document is a starting point rather than a comprehensive set of techniques and practices.

A fully secure development process should include comprehensive requirements from a standard such as the OWASP ASVS in addition to including a range of software development activities described in maturity models such as OWASP SAMM and BSIMM.

Target Audience

This document is primarily written for developers. However, development managers, product owners, Q/A professionals, program managers, and anyone involved in building software can also benefit from this document.

How this List Was Created

This list was originally created by the current project leads with contributions from several volunteers. The document was then shared globally so even anonymous suggestions could be considered. Hundreds of changes were accepted from this open community process.

Security Controls

The description of each control has the same structure. The control itself has an unique name preceeded by the control number: **Cx: Control Name**, e.g., *C1: Implement Access Control*.

Each control has the same sections:

- **Description**: A detailed description of the control including some best practices to consider.
- **Threat(s):** A threat or threats that this control counters.
- **Implementation**: Best practices and examples to illustrate how to implement each control.
- **Vulnerabilities Prevented**: List of prevented vulnerabilities or risks addressed (OWASP TOP 10 Risk, CWE, etc.)
- **References**: List of references for further study (OWASP Cheat sheet, Security Hardening Guidelines, etc.)
- **Tools**: Set of tools/projects to easily introduce/integrate security controls into your software.

C1: Implement Access Control

Description

Access Control (or Authorization) is allowing or denying specific requests from a user, program, or process. With each access control decision, a given subject requests access to a given object. Access control is the process that considers the defined policy and determines if a given subject is allowed to access a given object. Access control also involves the act of granting and revoking those privileges. Access Control often applies on multiple levels, e.g., given an application with a database backend, it applies both on the business logic level as well as on a database row level. In addition, applications can offer multiple ways of performing operations (e.g., through APIs or the website). All those different levels and access paths must be aligned, i.e., use the same access control checks, to protect against security vulnerabilities. Authorization (verifying access to specific features or resources) is not equivalent to authentication (verifying identity).

Threats

- An attacker could take advantage of a loosely configured access control policy to access data the organization did not intend to make accessible.
- An attacker could discover multiple access control components within an application and exploit the weakest.
- An administrator could forget to decommission an old account, and an attacker could discover that account and use it to access data.
- An attacker could gain access to data that had a policy that dropped into a final step of allowing access. (Lack of default deny)

Implementation

Below is a minimum set of access control design requirements that should be considered at the initial stages of application development.

1) Design Access Control Thoroughly Up Front

Once you have chosen a specific access control design pattern, it is often difficult and time-consuming to re-engineer access control in your application with a new pattern. Access Control is one of the main areas of application security design that must be thoroughly designed up front, especially when addressing requirements like multi-tenancy and horizontal (data dependent) access control.

Two core types of access control design should be considered.

- Role Based Access Control (RBAC) is a model for controlling access to resources where permitted actions on resources are identified with roles rather than with individual subject identities.
- Attribute Based Access Control (ABAC) will grant or deny user access based on arbitrary attributes of the user and arbitrary attributes of the object, and environment conditions that may be globally recognized, and more relevant to the policies at hand. Access Control design may start simple but can often become complex and feature-heavy security control. When evaluating the access control capability of software frameworks, ensure that your access control functionality will allow for customization for your specific access control feature need.

2) Force Every Access Request to Go Through an Access Control Check

Ensure that all access requests are forced to go through an access control verification layer. Technologies like Java filters or other automatic request processing mechanisms are ideal programming components that will ensure that all requests go through an access control check. This is refered to as *Policy Enforcement Point* in RFC 2904.

3) Consolidate the access control check

Use a single access control procedure or routine. This prevents the scenario where you have multiple access control implementations, where most are correct, but some are flawed. By using a centralized approach, you can focus security resources on reviewing and fixing one central library or function that performs the access control check, and then reuse it throughout your code base and organization.

4) Deny by Default

Ensure that by default, all the requests are denied, unless they are specifically allowed. This also includes accessing API (REST or webhooks) with missing access controls. There are many ways that this rule will manifest in the application code. Some examples are:

1. Application code may throw an error or exception while processing access control requests. In these cases, access control should always be denied.

2. When an administrator creates a new user or a user registers for a new account, that account should have minimal or no access by default until that access is configured.

3. When a new feature is added to an application, all users should be denied to use it until it's properly configured.

5) Principle of Least Privilege / Just in Time (JIT), Just Enough Access (JEA)

An example of implementing that principle is to create dedicated privileged roles and accounts for every organization function that requires highly privileged activities and avoid using an "admin" role/account that is fully privileged daily.

To further improve the security, you can implement Just-in-Time (JIT) or Just-enough-Access (JEA): ensure that all users, programs, or processes are only given just enough access to achieve their current mission. This access should be provided just in time, when the subject makes the request, and the access should be granted for a short time. Be wary of systems that do not provide granular access control configuration capabilities.

6) Do not Hardcode Roles

Many application frameworks default to access control that is role based. It is common to find application code filled with checks of this nature.

```
if (user.hasRole("ADMIN")) || (user.hasRole("MANAGER")) {
    deleteAccount();
}
```

Be careful about this type of role-based programming in code. It has the following limitations or dangers:

- Role-based programming of this nature is fragile. It is easy to create incorrect or missing role checks in code.
- Hard-Coded Roles do not allow for multi-tenancy. Extreme measures like forking the code or adding checks for each customer will be required to allow role-based systems to have different rules for different customers.
- Large codebases with many access control checks can make it difficult to audit or verify the overall application access control policy.
- Hard coded roles can also be seen as a backdoor when discovered during audits.

7) ABAC Policy Enforcement Point Example

Please consider the following access control enforcement points using this following programming methodology:

```
if (user.hasPermission("DELETE_ACCOUNT")) {
    deleteAccount();
}
```

Attribute or feature-based access control checks of this nature are the starting point to building well-designed and feature-rich access control systems. This type of programming also allows for greater access control customization capability over time.

Vulnerabilities Prevented

- OWASP Top 10 2021-A01_2021-Broken Access Control
- CWE Top 25 - 11:CWE-862 Missing Authorization
- CWE Top 25 - 24:CWE-863 Incorrect Authorization

References

- OWASP Cheat Sheet: Authorization Cheat Sheet
- OWASP Cheat Sheet: Logging Cheat Sheet
- OWASP ASVS V4 Access Control
- OWASP Testing Guide: Authorization Testing
- OAuth2.0 protocol for authorization
- Draft OAuth2.1
- Policy Enforcement in RFC 2904

Tools

- ZAP with the optional Access Control Testing add-on
- Open Policy Agent

C2: Use Cryptography the proper way

Description

Sensitive data such as passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws (EU's General Data Protection Regulation GDPR), financial data protection rules such as PCI Data Security Standard (PCI DSS) or other regulations.

Attackers can steal data from web and web service applications in a number of ways. For example, if sensitive information is sent over the internet without communications security, then an attacker on a shared wireless connection could capture and steal another user's data. Also, an attacker could use SQL Injection to steal passwords and other credentials from an applications database and expose that information to the public.

Privacy is assurance that the confidentiality of, and access to, certain information about an entity is protected. Users of the things developers build want their information protected from disclosure.

Protect sensitive data such as passwords, credit card numbers, health records, personal information, and business secrets.

Regulations exist to force companies to protect user's personal information. The European Union values individual privacy, such that they created the EU General Data Protection Regulation GDPR). Financial data protection rules such as PCI Data Security Standard (PCI DSS) also exist to protect cardholder privacy.

Cryptography is the art or science concerning the principles, means, and methods for rendering plain information unintelligible and restoring encrypted information to intelligible form. Individual user data requires cryptography to ensure it is properly cared for when stored.

Classify data types in your application

It's critical to classify data in your system and determine which level of sensitivity each piece of data belongs to. Each data category can then be mapped to protection rules necessary for each level of sensitivity. For example, public marketing information that is not sensitive may be categorized as public data which is ok to place on the public website. Credit card numbers may be classified as private user data which will need to be encrypted while stored, processed or in transit.

Data classification can also be mandated by legislation, e.g., GDPR when serving users within the European Union.

Classify the data sent, processed, and stored in your system and determine what level of sensitivity the data belongs to. Categorize the data to define specific protection rules for each type. The rule creation enables your team to perform data minimization and try not to store sensitive data whenever possible.

For example, public marketing information that is not sensitive may be categorized as public data, which is okay to place on the public website and does not need to be encrypted. Credit card numbers need to be encrypted while stored, processed, and in transit.

Implementation

When it comes to cryptography, there are a few simple rules:

- Never transmit plain-text data. The technical capability exists to easily encrypt all data that is sent between any point A and B. Embrace the use of

cryptography to protect all data at rest and in transit.
- Do not create your own cryptographic protocols. The creation of a cryptographic protocol is a tricky proposition. When NIST created AES, they had an open competition where the best cryptographers worldwide submitted proposals and then looked for flaws in the other proposals. Instead of using your developer cycles to create a new crypto protocol, use an existing, battle-tested standard. Focus your innovation on making your feature or product better.
- Do not implement cryptographic routines. Use an existing library that implements cryptographic routines.

Protect data at rest

The first rule of sensitive data management is to avoid storing sensitive data when at all possible. If you must store sensitive data then make sure it is cryptographically protected in some way to avoid unauthorized disclosure and modification. Cryptography (or crypto) is one of the more advanced topics of information security and one whose understanding requires the most schooling and experience. It is difficult to get right because there are many approaches to encryption, each with advantages and disadvantages that need to be thoroughly understood by web solution architects and developers. In addition, serious cryptography research is typically based on advanced mathematics and number theory, providing a serious barrier to entry. Designing or building cryptographic algorithms is very error-prone (see side-channel attacks). Instead of building cryptographic capability from scratch, it is strongly recommended that peer-reviewed and open solutions be used, such as the Google Tink project, Libsodium, and secure storage capability built into many software frameworks and cloud services.

Store passwords safely

Most web applications will face the challenge of storing user's passwords to set up authentication services. Store the passwords safely to ensure an attacker cannot quickly obtain them. Do not store the passwords in plain text anywhere in the database. Always use a hashing function to store passwords. Enhance the hashing function by adding a random salt for each item to increase the randomness of hashes.

Special Case: Application Secrets management

Applications contain numerous "secrets" that are needed for security operations. These include certificates, SQL connection passwords, third party service account credentials, passwords, SSH keys, encryption keys and more. The unauthorized disclosure or modification of these secrets could lead to complete system compromise. In managing application secrets, consider the following: Don't store secrets in code, config files or pass them through environment variables. Use tools like GitRob or TruffleHog to scan code repos for secrets. Your code should be written in a way that even if your code would be disclosed, e.g., due to a defective configured github repository, your running applications are still secure. Keep keys and your other application-level secrets in a secrets vault like KeyWhiz or Hashicorp's Vault project , Amazon KMS, or AWS Secrets Manager to provide secure storage and access to application-level secrets at run-time. Many web-frameworks such as Ruby on Rails provide integrated ways of dealing with secrets and credentials.

Key Lifecycle

Secret keys are used in applications with a number of sensitive functions. For example, secret keys can be used to sign JWTs, protect credit cards, provide various forms of authentication as well as facilitate other sensitive security features. In managing keys, a number of rules should be followed including

- Ensure that any secret key is protected from unauthorized access
- All authorized access to a secret key is logged for forensic purposes
- Store keys in a proper secrets vault as described below
- Use independent keys when multiple keys are required
- Build support for changing cryptographic algorithms to prepare for future needed changes
- Build application features to support and handle key rotation gracefully. This can happen on a periodic base or after a key has been compromised.

Protect data in transit

Sensitive data such as passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws (EU's General Data Protection Regulation GDPR), financial data protection rules such as PCI Data Security Standard (PCI DSS) or other regulations. Attackers can steal data from web and web service applications in a number of ways. For example, if sensitive information is sent over the internet without communications security, then an attacker on a shared wireless connection could capture and steal another user's data. Also, an attacker could use SQL Injection to steal passwords and other credentials from an applications database and expose that information to the public.

Use current cryptographic protocols

When developing web applications, use TLSv1.2 or TLSv1.3, preferably TLSv1.3. If possible, investigate the usage of HTTP/2 or HTTP/3 as they warrant the usage of security TLS versions/algorithms. - Directly turn off other older protocols to avoid protocol downgrade attacks. - Do not offer HTTP. Disable both HTTP and SSL compression. - Always utilize a secure random number generator (RNG).

Instruct Clients to enforce Transport Level Encryption

Web servers can instruct web browsers to uphold minimal transport-level security: - Use the Strict-Transport-Security Header to enforce opportunistic encryption and certificate validation checks. - Content-Security-Policy allows for automatic client-side upgrade from HTTP to HTTPS. - When setting cookies, always utilize the "secure" flag to prevent transmission over HTTP.

Support Cryptographic Agility: Cryptography changes over Time

Cryptographic recommendations change over time. To allow for this, make cryptographic choices such as used algorithms or key sizes configurable. This is called Cryptographic Agility

If the application needs to support high availability, design key-rollover procedures.

Vulnerabilities Prevented

- https://owasp.org/Top10/A02_2021-Cryptographic_Failures/
- https://mas.owasp.org/MASVS/controls/MASVS-CRYPTO-1/

References

- OWASP Cheat Sheet: Transport Layer Protection
- Ivan Ristic: SSL/TLS Deployment Best Practices
- OWASP Cheat Sheet: HSTS
- OWASP Cheat Sheet: Cryptographic Storage
- OWASP Cheat Sheet: Password Storage
- OWASP Cheat Sheet: IOS Developer - Insecure Data Storage
- OWASP Testing Guide: Testing for TLS
- OWASP WrongSecrets : vulnerable application with example of how to NOT use secrets

Tools

- https://github.com/nabla-c0d3/sslyze
- https://testssl.sh/
- SSLyze - SSL configuration scanning library and CLI tool
- SSLLabs - Free service for scanning and checking TLS/SSL configuration
- OWASP O-Saft TLS Tool - TLS connection testing tool
- GitRob - Command line tool to find sensitive information in publicly available files on GitHub
- TruffleHog - Searches for secrets accidentally committed
- Hashicorp Vault - Secrets manager
- Amazon KMS - Manage keys on AWS
- AWS Secrets Manager - Manage secrets on AWS
- Azure Key Vault - Manage keys and secrets on Azure
- Google Cloud KMS - Manage keys on Google Cloud Platform
- Google Secret Manager - Manage secrets on Google Cloud Platform

C3: Validate all Input & Handle Exceptions

Description

Input validation is a programming technique that ensures only properly formatted data may enter a software system component. When the injection attack targets a client (for example JavaScript based attacks), web servers can perform quoting/encoding on the attacker-provided data before forwarding it to the client.

Injection attacks commonly occur if an application confuses data input as executable commands and are often possible where input validation is forgotten or implemented wrong. For example, imagine that a web application accepts an email address as input from a user. The email address would be the expected "data". Attackers now search for ways to confuse applications to execute this (supposed) data as commands. Different injection attacks target different areas:

- When an attacker tricks an application into interpreting user input (data) as SQL commands (or parts thereof), there is a SQL injection attack. The injected command executes within the database server.
- Remote command injection (RCE) happens if an application confuses user data with commands that execute on the web application server/host. Server-Side Template Injections are another example of injections executed within the application server.
- When Javascript-Injections happen, a web application has accepted user data but is coerced to execute that data as code. Injected javascript code is typically executed within another user's web browser, thus not directly attacking the web server but other users.

Syntactic and Semantic Validity

An application should check that data is **syntactically** and **semantically** valid (in that order) before using it in any way (including displaying it back to the user).

- **Syntactic validity** means that the data is in the expected form. For example, an application may allow users to select a four-digit "account ID" to perform some operation. The application should assume the user is entering a SQL injection payload and check that the data entered by the user is precisely four digits in length and consists only of numbers (in addition to utilizing proper query parameterization).
- **Semantic validity** includes only accepting input within an acceptable range for application functionality and context. For example, a start date must be before an end date when choosing date ranges.

Implementation

Protection against Injection Attacks is typically based upon a defense-in-depth approach and incorporates input filtering, output escaping, and utilization of hardening mechanisms. The former two are only dependent upon implemented security measures, and the latter is mostly dependent upon client-support, e.g., when protecting against XSS, filtering XSS from input and escaping output data server-side would prevent XSS regardless of the used web browser; adding a Content-Security-Policy prevents XSS, but only if the user's browser supports it. Due to this, security must never depend upon optional hardening measures alone.

Prevent malicious data from entering the system

Never trust provided data! Screen all data for malicious patterns or, even better, check all data against an allow list.

Allowlisting vs Denylisting

There are two general approaches to performing syntactic validation, commonly known as allow and deny lists:

- Denylisting or **denylist validation** attempts to check that given data does not contain "known bad" content. For example, a web application may block input containing the exact text <SCRIPT> to help prevent XSS. However, this defense could be evaded with a lowercase script tag or a script tag of mixed case.
- Allowlisting or **allowlist validation** attempts to check that a given data matches a set of "known good" rules. For example, a allowlist validation rule for a US state would be a 2-letter code that is only one of the valid US states. Allowlisting is the recommended minimal approach. Denylisting is prone to error, can be bypassed with various evasion techniques, and can be dangerous when dependent on itself. Even though denylisting is often evaded, it can be useful to help detect obvious attacks. So while allowlisting helps limit the attack surface by ensuring data is of the right syntactic and semantic validity, denylisting helps detect and potentially stop obvious attacks.

Client side and Server side Validation

Always perform Input validation on the server side for security. While client-side validation is useful for both functional and security purposes, it is easily bypassed. Therefore, client-side validation is performed for usability purposes, but the application's security must not depend upon it. For example, JavaScript validation may alert the user that a particular field must consist of numbers. Still, the server-side application must validate that the submitted data only consists of numbers in the appropriate numerical range for that feature. Another benefit of using both client AND server-side validation is that any server-side validation warnings can be logged to inform operations of a potential hacker as the client-side validation had been bypassed.

Regular Expressions

Regular expressions offer a way to check whether data matches a specific pattern. Let's start with a basic example. The following regular expression defines an allowlist rule to validate usernames.

```
^\[a-z0-9_\]{3,16}$
```

This regular expression allows only lowercase letters, numbers, and the underscore character. The username is also restricted to a length of 3 and 16 characters.

Caution: Potential for Denial of Service

Care should be exercised when creating regular expressions. Poorly designed expressions may result in potential denial of service conditions (aka ReDoS). Various tools can be tested to verify that regular expressions are not vulnerable to ReDoS.

Caution: Complexity

Regular expressions are just one way to accomplish validation. Regular expressions can be difficult to maintain or understand for some developers. Other validation alternatives involve writing validation methods programmatically, which can be easier to maintain for some developers.

Unexpected User Input (Mass Assignment)

Some frameworks support automatic binding of HTTP requests parameters to server-side objects used by the application. This auto-binding feature can allow an attacker to update server-side objects that were not meant to be modified. The attacker can possibly modify their access control level or circumvent the intended business logic of the application with this feature. This attack has a number of names including: mass assignment, autobinding and object injection. As a simple example, if the user object has a field privilege which specifies the user's privilege level in the application, a malicious user can look for pages where user data is modified and add privilege=admin to the HTTP parameters sent. If auto-binding is enabled in an insecure fashion, the server-side object representing the user will be modified accordingly.

Two approaches can be used to handle this:

- Avoid binding input directly and use Data Transfer Objects (DTOs) instead.
- Enable auto-binding but set up allowlist rules for each page or feature to define which fields are allowed to be auto-bound.

More examples are available in the OWASP Mass Assignment Cheat Sheet

Limits of Input Validation

Input validation does not always make data "safe" since certain complex input forms may be "valid" but still dangerous. For example, a valid email address may contain a SQL injection attack, or a valid URL may contain a Cross Site Scripting attack. Additional defenses besides input validation should always be applied to data, such as query parameterization or escaping.

Use mechanisms that uphold the separation of data and commands

Even if malicious data has passed the input checking, applications can prevent injection attacks by never executing those malicious data as commands/code. Multiple measures can achieve this goal, most of them are technology-dependent. For Example:

- When using relational databases through SQL, utilize Prepared-Statements. SQL Injection attacks typically happen if an attacker can provide input data that "escapes" from SQL-Commands created through string concatenation. Using Prepared Statements allows the computer to automatically encode input data in a way that does not allow it to "escape" from the command template.
- When using an ORM, be sure that you know how objects are mapped to SQL commands. While their layer of indirection might prevent common SQLi, specially prepared attacks are often still feasible.
- Server-Side Template Injection (SSTI) uses a templating engine on the server-side to dynamically generate content that is then displayed to the user. SSTI engines often allow configuration of sandboxing, i.e., only allow execution of a limited amount of methods.
- Executing System Commands with user input as parameters is prone to injection attacks. If feasible, this should be avoided.

JavaScript Injection Attacks

A special case are JavaScript based Injection attacks (XSS). The injected malicious code is commonly executed within a victim's browser. Typically, attackers try to steal the user's session information from the browser and not directly execute commands (as they do on the server-side). In addition to server-side input filtering and output escaping, multiple client-side hardening measurements can be taken (these also protect against the special case of DOM-based XSS where no server-side logic is involved and thus cannot filter malicious code): - Mark sensitive cookies with httpOnly so JavaScript cannot access them - Utilize a Content-Security-Policy to reduce the attack-surface for JavaScript-based Attacks - Use a secure by default framework like Angular

Validating and Sanitizing HTML

Consider an application that needs to accept HTML from users (via a WYSIWYG editor that represents content as HTML or features that directly accept HTML in input). In this situation, validation or escaping will not help. - Regular expressions are not expressive enough to understand the complexity of HTML5. - Encoding or escaping HTML will not help since it will cause the HTML not to render properly.

Therefore, you need a library to parse and clean HTML formatted text. Please see the XSS Prevention Cheat Sheet on HTML Sanitization for more information on HTML Sanitization.

Special Case: Validate Data During Deserialization

Some forms of input are so complex that validation can only minimally protect the application. For example, it's dangerous to deserialize untrusted data or data that can be manipulated by an attacker. The only safe architectural pattern is to not accept serialized objects from untrusted sources or to only deserialize in limited capacity for only simple data types. You should avoid processing serialized data formats and use easier to defend formats such as JSON when possible.

If that is not possible then consider a series of validation defenses when processing serialized data.

- Implement integrity checks and encryption of the serialized objects to prevent hostile object creation or data tampering.
- Enforce strict type constraints during deserialization before object creation; typically code is expecting a definable set of classes. Bypasses to this technique have been demonstrated.
- Isolate code that deserializes, such that it runs in very low privilege environments, such as temporary containers.
- Log security deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitor deserialization, alerting if a user deserializes constantly.

Vulnerabilities Prevented

- Input validation reduces the attack surface of applications and can sometimes make attacks more difficult against an application.
- Input validation is a technique that provides security to certain forms of data, specific to certain attacks and cannot be reliably applied as a general security rule.
- Input validation should not be used as the primary method of preventing XSS, SQL Injection and other attacks.
- 2023 CWE Top 25 - 3 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- 2023 CWE Top 25 - 5 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- 2023 CWE Top 25 - 16 Improper Neutralization of Special Elements used in a Command ('Command Injection')
- 2023 CWE Top 25 - 23 Improper Control of Generation of Code ('Code Injection')

References

Regarding Input Validation: - OWASP Cheat Sheet: Input Validation - OWASP Cheat Sheet: iOS - Security Decisions via Untrusted Inputs - OWASP Testing Guide: Testing for Input Validation - Injection Prevention Cheat Sheet - Injection Prevention Cheat Sheet in Java - Hardening with CSP: CSP with Google - Deploying CSP in Single Page Applications

Tools

Helping with Input Validation: - OWASP Java HTML Sanitizer Project - Java JSR-303/JSR-349 Bean Validation - Java Hibernate ValidatorApache Commons ValidatorPHP's filter functions

Testing for Injection Attacks: - Sqlmap.py - OWASP ZAP-based scans Helping with Hardening: - CSP Evaluator

C4: Address Security from the Start

Description

When designing a new application, creating a secure architecture prevents vulnerabilties before they even become part of the application. This prevents costly repairs and repudiation problems.

There are design principles that lead to secure architectures:

- **keep it simple, stupid** (KISS): the easier an application is to understand, the easier it is to reason about its components and their interactions. This allows to reason about the application's security behavior.
- **Make it easy to do the right thing**: don't expect the user to read documentation or invest time to "do things the right way". By default the application should behave in a secure manner. To make it insecure, an explicit action by the user has to take place.
- **don't rely on obscurity**: if the only security is due to the intransparency of the application or its source code, the application is not secure at all.

- **Identify and minimize your exposed components** ("attack surface"): attackers cannot attack what's not there.
- **Design for Defense-in-Depth**: think about what happens, if a component is breached and about the potential blast radius of an attack.

Threats

- If the application is only protected by security-by-obscurity, an attacker that reverse-engineers the application has full permissions as soon the obfuscation is cleared-up. In addtion, an attacker is able to monitor network traffic: while the obfuscation might be performed on the code-level, the operations on the network level can easily be analyzed.
- A web-application with a complex authorization scheme is deployed. A new software developer is tasked with extending one of the components. Due to the complexity, they misconfigure the authorization scheme and an attacker is able to exploit IDOR.
- A web-application with a complex authorization scheme is deployed. A new software developer adds a new plugin to the system. The system makes it hard to do the right thing, and all security configuration must be manually added to the plugin, by-default no security measures are taken. The new developer is not configuring anything thus the new plugin introduces an IDOR into the system.
- A web-application has many components, all of which are exposed to the public internet. The resulting attack surface is massive. For example, a database management tool (e.g., phpmyadmin) is deployed. After a 0day was found in mysqladmin, the whole database was extracted. During normal use, nobody uses phpmyadmin.

Implementation

The mantra "Complexity is the enemy of enemy of security" can be seen throughout this implementation guidance.

Design for Clarity and Transparency

Architecture should focus upon simplicity: the designed software should be only as complex as the intended user's requirements warant. Focusing upon simplicity brings multiple benefits for the created software:

- It is easier to reason about a simple system. This allows to reason about potential security impacts of changes.
- Long-term maintenance is aided through the simpler design.
- The design should focus upon transprency, i.e., the security should not depend upon security-by-obscurity.

Make it easy to do the right thing

Two terms often heard are "security by design" and "security by default". The former implies, that the software system should be usable in a secure manner while the latter means that the initial configuration of the software system is secure.

This implies, that an system administrator has to make an explicit choice to introduce insecure configuration into the system. In constrast, the path of least resistance should always result in a secure system.

When focusing upon end-user interactions, this aspect is important for designing user interfaces and flows. When focusing upon developer interactions, develeper-facing facilities such as framework, APIs, network APIs should be designed that when using them with default values, only secure operations should occur. Think about this when designing your configuration files too

Clearly articulate what's trusted to do what, and ensure those relationships are enforced

Clearly articulate what's trusted to do what, and ensure those relationships are enforced, e.g., trust boundaries deliniate blast radius and are enforced by controls, such as firewalls or gateways.

Attenuate what's allowed by careful validation at each step. Go deeper with threat modeling mnemonics like stride or methodologies like stride per element.

Identify and minimize your exposed components ("attack surface")

Identify all areas that an attacker can access, review them and try to minimize them: attackers cannot attack what's not there.

In addition, exposing only a minimal set of operations makes long-term maintenance easier.

Use well-known Architecture Patterns

Experts have shared their wisdom about best practices in an easily digestible format called secure architecture patterns. Architecture patterns are reusable and can be applied across multiple applications.

For a solution to be considered a pattern, it must have these characteristics:

- First, a secure architecture pattern must solve a security problem.
- Second, a secure architecture pattern must not be tied to a specific vendor or technology.
- Third, a secure architecture pattern must demonstrate how it mitigates threats.
- Fourth, a secure architecture pattern must use standardized terms for threats and controls for easy reuse.[1]

An architecture pattern is a way to solve a problem using a standard solution versus creating a custom solution. A secure architecture pattern is a standard solution that has been reviewed and hardened against known security threats.

Implementation:

1. Identify the problem that requires solving.
2. Consider the catalog of available secure architecture patterns.
3. Choose a secure architecture pattern for the design.
4. Implement the secure architecture pattern.

Vulnerabilities Prevented

- Business Logic Flaws: These patterns can help in structuring the application to avoid complex and often overlooked business logic vulnerabilities.
- OWASP Top 10 2021-A04 (Insecure Design): Secure architecture patterns directly target the mitigation of risks associated with insecure design, a key concern highlighted by OWASP.

References

- https://securitypatterns.io/what-is-a-security-pattern/
- https://owasp.org/www-pdf-archive/Vanhilst_owasp_140319.pdf
- https://cheatsheetseries.owasp.org/cheatsheets/Microservices_based_Security_Arch_Doc_Cheat_Sheet.html
- https://cheatsheetseries.owasp.org/cheatsheets/Secure_Product_Design_Cheat_Sheet.html

Tools

- maybe add some threat modeling tools here?

C5: Secure By Default Configurations

Description

"Secure-by-Default" means products are resilient against prevalent exploitation techniques out of the box without additional charge.

[2]

The benefit of having an application secure from the start is that it removes the burden away from developers on how to lock a system down, providing them with an already secure product. It reduces the effort required to deploy products in a secure manner and gives greater confidence that they will remain secure over time.

Implementation

In modern cloud applications, when developers build applications, they are also building the infrastructure for their applications, making infrastructure decisions, including security-critical configurations, while writing their code. These are deployed on infrastructure created and configured via code, Infrastructure-as-code (IaC), using configurations applied at the application level (including web server and database), at container, Function as a Service, or infrastructure level. For example, in the case of web applications, folder permissions need to follow the principle of least privilege to limit resource access rights. When web and mobile applications are deployed in production, the debugging should be disabled.

Is it important that when developers put together their infrastructure components, they: 1. Implement configurations based on the Least privilege principle - for example: ensure that your cloud storage (S3 or other) is configured to be private and accessed by the minimum amount of time 2. Access is denied by default and allowed via an allowed list 3. Use container images that have been scanned for package and component vulnerabilities and pulled from a private container registry 4. Prefer declarative infrastructure configuration over manual configuration activities. On a low-level, utilize Infrastructure-as-Code templates for automated provisioning and configuration of your cloud and on-premises infrastructure. On a high-level utilize Policy-as-Code to enforce policies including privilege assignments.
Using a declarative format allows those policies to be managed similar to source code: checked-in into a source code management system, versioned, access-controlled, subject to change management, etc. 5. Traffic encryption - by default or do not implement unencrypted communication channels in the first place

Continuous Configurations Verification

As part of software development, a developer needs to ensure that software is configured securely by default at the application level. For example,

- The code which defines the infrastructure should follow the principle of least privilege.
- Configurations and features that aren't required such as accounts, software, and demo capabilities should be disabled.

Vulnerabilities Prevented

- OWASP Top 10 2021 A05 – Security Misconfiguration

References

- OWASP Cheat Sheet: Infrastructure as Code Security Cheatsheet
- OWASP ASVS: Application Security Verification Standard V14 Configuration
- Cloud security guidance - NCSC.GOV.UK

Tools

- Tfsec - open source static analysis for your Terraform templates
- Terrascan - scan for Infrastructure-as-Code vulnerabilities
- Checkov - Scan for open-source and Infrastructure-as-Code vulnerabilities
- Scout Suite is an open source multi-cloud security-auditing tool which currently supports: Amazon Web Services, Microsoft Azure, Google Cloud Platform
- prowler

- Cloudmapper
- Snyk - Scan for open-source, code, container, and Infrastructure-as-Code vulnerabilities
- Trivy - Scan for open-source, code, container, and Infrastructure-as-Code vulnerabilities
- KICS - Scan for Infrastructure-as-Code vulnerabilities
- Kubescape - Scan for Kubernetes vulnerabilities
- Kyverno - Securing Kubernetes using Policies

C6: Keep your Components Secure

Description

It is a common practice in software development to leverage libraries and frameworks. Secure libraries and software frameworks with embedded security help software developers prevent security-related design and implementation flaws. A developer writing an application from scratch might not have sufficient knowledge, time, or budget to properly implement or maintain security features. Leveraging security frameworks (both open source and vendor) help accomplish security goals more efficiently and accurately.

When possible, the emphasis should be on using the existing secure features of frameworks rather than importing yet another third party libraries, which requires regular updates and maintenance. It is preferable to have developers take advantage of what they're already using instead of forcing yet another library on them.

When incorporating third party libraries or frameworks into your software, it is important to consider the following two categories of best practices:

1. Identify trusted libraries and frameworks to bring into your software.
2. Monitor and update packages to ensure that your software is not vulnerable to the possible security vulnerabilities introduced by the third party components .

Implementation

Below each of these categories are further detailed to help secure your software.

Best Practices to Identify Trusted libraries

Below are listed a few criteria you can use to select the next library or framework for your software. This is not an exhaustive list, but is a good start.

1. **Sources**: Download recommended security libraries from official sources over secure links and prefer signed packages to reduce the chance of including a modified, malicious component (See A08:2021-Software and Data Integrity Failures).
2. **Popularity**: Leverage libraries and frameworks used by many applications which have around large communities. Consider data points such as the number of GitHub stars a package's source code repository has received, and number of downloads from within a package manager.
3. **Activity:** Ensure that the library/ framework is actively maintained and issues are resolved in a timely fashion.
4. **Maturity**: Use stable versions . Projects in early stages of development area higher risk to your software .
5. **Complexity**: A large, complex library with lots of dependencies, is more difficult to incorporate into your software. Also, a high number of dependencies indicates a higher number of future upgrades to ensure all those dependencies are up-to-date and secured.
6. **Security**: If the package is open source, you can use static application security testing (SAST) or Software Composition Analysis (SCA) to help identify malicious code or security weaknesses, before first including them.

Best Practices to Keep them Secure

New security vulnerabilities are disclosed every day and are published in public databases like the NIST National Vulnerability Database (NVD) which identifies publicly known vulnerabilities using Common Vulnerabilities and Exposures (CVE). Furthermore, exploits made available in public databases allow attackers to automate their attacks. As a result of this, it is important to ensure on a regular basis that your software is free of well-known security vulnerabilities.

1. **Maintain an inventory catalog** of all the third party components. It is recommended to automatically create SBOMs (Software-Bill-Of-Materials) from within the build pipeline. A SBOM contains all used third-party dependencies and their versions and can be automatically monitored by a variety of supply chain management tools.
2. **Perform continuous checks.** Use your SBOMs together with periodic or continuous monitoring tools such as OWASP dependency-track to automatically detect well-known publicly disclosed vulnerabilities.
3. **Verify for security early and often** - integrate SCA tools in early stages of software development, to gain visibility in the number and criticality of security vulnerabilities of the software and its dependencies from every stage of the software development lifecycle.
4. **Proactively** update libraries and components. Updating software must be a recurring task that occurs throughout the lifecycle of the application or product, from ideation to retirement.

Vulnerabilities Prevented

Secure frameworks and libraries can help to prevent a wide range of web application vulnerabilities. It is critical to keep these frameworks and libraries up to date as described in using vulnerable and outdated components with known vulnerabilities Top 10 2021.

References

- OWASP Cheat Sheet: Third Party Javascript Management
- OpenSSF Scorecard - Security health metrics for Open Source

Tools

- OWASP Dependency-Check - to identify project dependencies and check for publicly disclosed vulnerabilities
- OWASP Dependency-Track - periodically monitor SBOM files for new vulnerabilities
- Retire.JS scanner for JavaScript libraries
- Renovate for automated dependencies updates
- Harbor : an open source registry that secures artifacts with policies and role-based access control

C7: Implement Digital Identity

Description

Digital Identity is a unique representation of an individual, organization (or another subject) as they engage in an online transaction. Authentication is the process of verifying that an individual or entity is who they claim to be.

Session management is the process by which a server maintains the state of the user's authentication so that the user may continue to use the system without re-authenticating. Digital identity, authentication, and session management are very complex topics. We're scratching the surface of the topic of Digital Identity here. Ensure that your most capable engineering talent is responsible for maintaining the complexity involved with most Identity solutions. The NIST Special Publication 800-63B: Digital Identity Guidelines (Authentication and Lifecycle Management provide solid guidance on implementing digital identity, authentication, and session management controls. Below are some recommendations for secure implementation to ensure strong digital identity controls are implemented in applications.

Authentication Assurance Levels

NIST 800-63b describes three levels of authentication assurance called Authentication Assurance Level (AAL):

- **Level 1 : Passwords**: The first level, AAL level 1 is reserved for lower-risk applications that do not contain PII or other private data. At AAL level 1 only single-factor authentication is required, typically through the use of a password (something you know). The security of passwords (or credentials in general) is of utmost importance, this includes both secure storage (using a key-derivation function and such) as well as corresponding processes, e.g. having a secure password-reset flow.
- **Level 2 : Multi-Factor Authentication**: NIST 800-63b AAL level 2 is reserved for higher-risk applications that contain "self-asserted PII or other personal information made available online." At AAL level 2 multi-factor authentication is required including OTP or other forms of multi-factor implementation.
- **Level 3 : Cryptographic Based Authentication**: NIST 800-63b Authentication Assurance Level 3 (AAL3) is required when the impact of compromised systems could lead to personal harm, significant financial loss, harm the public interest or involve civil or criminal violations. AAL3 requires authentication that is "based on proof of possession of a key through a cryptographic protocol." This type of authentication is used to achieve the strongest level of authentication assurance. This is typically done through hardware cryptographic modules. When developing web applications, this will commonly lead to WebAuthn or PassKeys.

Level 2 : Multi-Factor Authentication

NIST 800-63b AAL level 2 is reserved for higher-risk applications that contain "self-asserted PII or other personal information made available online." At AAL level 2 multi-factor authentication is required including OTP or other forms of multi-factor implementation. Multi-factor authentication (MFA) ensures that users are who they claim to be by requiring them to identify themselves with a combination of:

- Something you know – password or PIN
- Something you own – token or phone, when using a phone please use a standard authenticator application heeding standardized protocols such as FIDO2.
- Something you are – biometrics, such as a fingerprint Using passwords as a sole factor provides weak security. Multi-factor solutions provide a more robust solution by requiring an attacker to acquire more than one element to authenticate with the service. It is worth noting that biometrics, when employed as a single factor of authentication, are not considered acceptable secrets for digital authentication. They can be obtained online or by taking a picture of someone with a camera phone (e.g., facial images) with or without their knowledge, lifted from objects someone touches (e.g., latent fingerprints), or captured with high-resolution images (e.g., iris patterns). Biometrics must be used only as part of multi-factor authentication with a physical authenticator (something you own). For example, accessing a multi-factor one-time password (OTP) device will generate a one-time password that the user manually enters for the verifier.

Level 3 : Cryptographic Based Authentication

NIST 800-63b Authentication Assurance Level 3 (AAL3) is required when the impact of compromised systems could lead to personal harm, significant financial loss, harm the public interest or involve civil or criminal violations. AAL3 requires authentication that is "based on proof of possession of a key through a cryptographic protocol." This type of authentication is used to achieve the strongest level of authentication assurance. This is typically done through hardware cryptographic modules. When developing web applications, this will commonly lead to WebAuthn or PassKeys.

Session Management: client- vs server-side sessions

HTTP on its own is a session-less protocol: no data is shared between requests. When you look at how we are using the web, this is clearly not what is user-visible as for example you log into a website and stay logged in during subsequent requests. This is possible as session-management has been implemented on top of HTTP. Once the initial successful user authentication has taken place, an application may choose to track and maintain this authentication state for a limited amount of time. This will allow the user to continue using the application without having to keep re-authentication with each request. Tracking of this user state is called Session Management. Session-Management can be roughly categorized in client- and server-side session management. In the former, all session data is stored within the client and transmitted on each request to the server. The latter stores session-

specific data on the server, e.g., in a database, and only transmits an identifier to the client. The client then submits only the session-identifier on each request and the server retrieves the session-data from the server-side storage.

From a security-perspective server-side sessions have multiple benefits: - Data is not directly stored on the client: this can be problematic, e.g., when handling sensitive data. In addition, client-side session-management solutions must ensure that client-side data has not been tampered with. - Less data is transmitted between client and server (which is not as relevant as network bandwidth has increased) - Server-side session-management allows for session-invalidation, e.g., a user can logout all of their sessions By default, always use server-side session management..

Implementation

When using Passwords

Password Requirements

Passwords should comply with the following requirements at the very least: - be at least 8 characters in length if multi-factor authentication (MFA) and other controls are also used. If MFA is not possible, this should be increased to at least 10 characters - all printing ASCII characters as well as the space character should be acceptable in memorized secrets - encourage the use of long passwords and passphrases - remove complexity requirements as these have been found to be of limited effectiveness. Instead, the adoption of MFA or longer password lengths is recommended - ensure that passwords used are not commonly used passwords that have been already been leaked in a previous compromise. You may choose to block the top 1000 or 10000 most common passwords which meet the above length requirements and are found in compromised password lists. The following link contains the most commonly found passwords: https://github.com/danielmiessler/SecLists/tree/master/Passwords - Enforce password rotation, to avoid potential breaches due to the fact the same password is being used for a very long period of time

Implement Secure Password Recovery Mechanism

It is common for an application to have a mechanism for a user to gain access to their account in the event they forget their password. A good design workflow for a password recovery feature will use multi-factor authentication elements. For example, it may ask a security question - something they know, and then send a generated token to a device - something they own. Please see the Forgot_Password_Cheat_Sheet and Choosing_and_Using_Security_Questions_Cheat_Sheet for further details.

Implement Secure Password Storage

In order to provide strong authentication controls, an application must securely store user credentials. Furthermore, cryptographic controls should be in place such that if a credential (e.g., a password) is compromised, the attacker does not immediately have access to this information. Please see the OWASP Password Storage Cheat Sheet for further details.

Server-Side Session-Management

Typically server-side session management is implemented with HTTP cookies which are used to store a session-identifier. When a new session is requested, the server generates a new session-identifier and transmits it to the client (browser). On each subsequent request, the session-identifier is transmitted from the client to the server, and the server uses this session-identifier to lookup session-data within a server-side database.

Session Generation and Expiration

User state is tracked in a session. This session is typically stored on the server for traditional web based session management. A session identifier is then given to the user so the user can identify which server-side session contains the correct user data. The client only needs to maintain this session identifier, which also keeps sensitive server-side session data off of the client. Here are a few controls to consider when building or implementing session management solutions: - Ensure that the session id is long, unique and random, i.e., is of high entropy. - The application should generate a new session during authentication and re-authentication. - The application should implement an idle timeout after a period of inactivity and an absolute maximum lifetime for each session, after which users must re-authenticate. The length of the timeouts should be inversely proportional with the value of the data protected.

Please see the Session Management Cheat Sheet further details. ASVS Section 3 covers additional session management requirements.

Client-Side Session-Management

Server-side sessions can be limiting for some forms of authentication. "Stateless services" allow for client side management of session data for performance purposes so the server has less of a burden to store user sessions. These "stateless" applications typically generate a short-lived access token containing all of the current user's access permissions which is then included in all subsequent requests. Cryptography must be employed so that the client cannot alter the permissions stored within the token. When a client requests a server operation, the client includes the retrieved access token and the server verifies that the token has not been tampered with and extracts the permissions from the token. These permissions are then used for subsequent permission checks.

JWT (JSON Web Tokens)

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted as long as it is digitally signed by a trusted authority. A JWT token is created during authentication and is verified by the server (or servers) before any processing. However, JWTs are often not saved by the server after initial creation. JWTs are typically created and then handed to a client without being saved by the server in any way. The integrity of the token is maintained through the use of digital signatures so a server can later verify that the JWT is still valid and was not tampered with since its creation. This approach is both stateless and portable in the way that client and server technologies can be different yet still interact. Please note, that if you are using JWTs you have to make sure that the returned JWT is actually using one of the signing algorithms that you are using. Otherwise, an attacker could try to create a JWT signed with the NULL algorithm, use a MAC-vs-Signature confusion attack, or provide a custom JWS key for signing. When you are issuing JWTs, make double-sure that you are using a secure private key for signing the JWTs: each output JWT gives an attacker all information needed to perform an offline cracking attack, so you should rotate keys frequently too.

Browser Cookies

Browser cookies are a common method for web applications to store session identifiers for web applications implementing standard session management techniques. Here are some defenses to consider when using browser cookies.

- When browser cookies are used as the mechanism for tracking the session of an authenticated user, these should be accessible to a minimum set of domains and paths and should be tagged to expire at, or soon after, the session's validity period.
- Please be aware that not explicitly stating a domain during cookie setup will use the current origin as domain. This is a sensible default.
- Please be aware, that while stating a path during cookie setup will limit the browser to only submit the cookie if the request lies within the stated path. This protects the cookie of one application from being accessed by another application within a different path on the same server. This protection is brittle: if the "other" application has an XSS vulnerability and the attacker can introduce iframes, the "path" protection can be circumvented.
- The 'secure' flag should be set to ensure the transfer is done via secure channel only (TLS).
- HttpOnly flag should be set to prevent the cookie from being accessed via JavaScript.
- Adding "samesite" attributes to cookies prevents some modern browsers from sending cookies with cross-site requests and provides protection against cross-site request forgery and information leakage attacks.

Vulnerabilities Prevented

- A07:2021 – Identification and Authentication Failures
- OWASP Mobile Top 10 2016-M4- Insecure Authentication

References

- OWASP Cheat Sheet: Authentication
- OWASP Cheat Sheet: Password Storage
- OWASP Cheat Sheet: Forgot Password
- OWASP Cheat Sheet: Choosing and Using Security Questions
- OWASP Cheat Sheet: Session Management
- NIST Special Publication 800-63 Revision 3 - Digital Identity Guidelines

Tools

- Daniel Miessler: Most commonly found passwords

C8: Leverage Browser Security Features

Description

Browsers are the gateway to the web for most users. As such, it's critical to employ robust security measures to protect the user from various threats. This section outlines the techniques and policies that can be implemented to bolster browser security.

While we are currently focusing upon traditional web browsers, please note that there is a diverse world of other client programs out there, ranging from API clients to smart-tvs.

Opportunistic Security and Browser-Support

Instructing the web browser to enforce security measures is always opportunistic: the web application cannot verify that the browser heeds the guidance given and thus these security measures should always be seen as additional (and optional) **Hardening Measures** that further complicate an attacker's life.

In addition, web browsers must actually support the security guidance offered by web applications. The level of support differs between different browsers and their versions. Web sites such as https://caniuse.com can be used to check which web browser (versions) support which features. The supported security features can change over time, e.g., the X-XSS-Protection header has been removed from all major browsers; the browsers' default behavior can change over time as seen with Referrer-Policy; and even the semantics of existing headers can change over time as seen with X-Content-Type-Options.

While the changing browser feature set can be problematic, typically newer browser provide more security features. They sometimes even enable them by default. Explicitly setting those security headers can unify the different browsers' behaviors and thus reduces maintenance effort.

A fully compromised browser might not heed security guidance but if an adversary was able to take full control of a browser, they already have far more damaging attack paths than just ignoring security guidance.

Implementation

Typically, there are two (Security Header specific) ways that web applications can use to instruct web browsers about security: HTTP headers and HTML tags.

The behavior taken if a security directive is given more than one time is security header specific. For example, a duplicate X-Frame-Options header will disable its protection while a duplicate Content-Security-Policy header will lead to a stricter policy thus tightening its security. The following is a non-exhaustive list of potential Hardening mechanisms:

Configure the Browser to prevent Information Disclosure

Information disclosure occurs if the browser transmits information over unencrypted channels (HTTP instead of HTTPS) or sends our too much information in the first place (e.g., through the Referer-Header). The following mechanisms reduce the possibility of information disclosure: - **HTTP Strict Transport Security (HSTS)**: Ensures that browsers only connect to your website over HTTPS, preventing SSL stripping attacks. - **Content Security Policy (CSP)**: CSP policies can instruct the browser to automatically upgrade HTTP connections to HTTPS. In addition directives such as the form-src directive can be used to prevent forms from transmitting data to external sites. - **Referrer-Policy**: when navigating between pages, the browser's HTTP request includes the current URL within the outgoing request. This URL can include sensitive information. Using Referrer-Policy, a web-site can unify the browser's behavior and select which information should be transmitted between web sites. - The cookie's **secure** flag: while not a HTTP header, this security flag is related to information disclosure. If set, the web browser will not transmit a cookie over unencrypted HTTP transports.

Reduce the potential Impact of XSS

Javascript based XSS attacks have been very common for decades. To reduce the potential impact of vulnerabilities, browsers offer rich defensive mechanisms that should reduce the potential impact of XSS attacks: - **Content Security Policy (CSP)**: CSP is a powerful tool that helps prevent a wide range of attacks including Cross-Site Scripting (XSS) and data injection. Strict CSP policies can effectively disable inline JavaScript and style, making it much harder for attackers to inject malicious content.
Host Allowlist CSP: Blocking all third-party JavaScript can significantly reduce the attack surface and prevent the exploitation of vulnerabilities in third-party libraries.
Trusted Types: This is a browser API that helps prevent DOM-based cross-site scripting vulnerabilities by ensuring only secure data types can be inserted into the DOM. - The cookie's **httpOnly** flag: while not a HTTP header, setting this flag prevents Javascript from accessing this cookie and should be done esp. For Session cookies.

Prevent Clickjacking

Clickjacking, also known as UI-redress attacks, try to confuse users by overlaying a malicious site on top of a benign one. The user believes to interact with the benign one while in reality they are interacting with the malicious one. - **X-Frame-Options (XFO)**: Prevents clickjacking attacks by ensuring your content is not embedded into other sites. This header is finicky to use, e.g., when used twice it is disabled. - **Content Security Policy (CSP)**: the different frame-* directives allow for fine-grained control of which sites are allowed to include the current website as well as which other sites can be included within the current website.

Control the Browser's Advanced Capabilities

Modern browsers do not only display HTML code but are used to interface with multiple system components such as WebCams, Microphones, USB Devices, etc. While many websites do not utilize those features, attackers can abuse those. - **Permission Policy**: through a permission policy a web-site can instruct the browser that the defined features will not be used by the web-site. For example, a web-site can state that it will never capture user audio. Even if an attacker is able to inject malicious code, they can thus not instruct the web-browser to capture audio.

Prevent CSRF Attacks

CSRF attacks abuse an existing trust relationship between the web browser and web sites. - Same-Origin Cookies: Marking cookies as SameSite can mitigate the risk of cross-origin information leakage, as well as provide some protection against cross-site request forgery attacks.

Vulnerabilities Prevented

Implementing these browser defenses can help mitigate a range of vulnerabilities, including but not limited to: - Cross-Site Scripting (XSS) - Cross-Site Request Forgery (CSRF) - Clickjacking - Data Theft through insecure transmission - Session Hijacking - Abusing unintended browser hardware access (microphone, cameras, etc.)

Tools

- Web Check
- Security Headers
- Mozilla Observatory

References

- Content Security Policy (CSP)
- OWASP Secure Headers Project
- Fetch Metadata Request Headers
- https://caniuse.com/

C9: Implement Security Logging and Monitoring

Description

Logging is a concept that most developers already use for debugging and diagnostic purposes. Security logging is an equally basic concept: to log security information during the runtime operation of an application. Monitoring is the live review of application and security logs using various forms of automation. The same tools and patterns can be used for operations, debugging and security purposes.

Benefits of Security Logging

Security logging can be used for: 1. Feeding intrusion detection systems 2. Forensic analysis and investigations 3. Satisfying regulatory compliance requirements

Logging for Intrusion Detection and Response

Use logging to identify activity that indicates that a user is behaving maliciously. Potentially malicious activity to log includes: - Submitted data that is outside of an expected numeric range. - Submitted data that involves changes to data that should not be modifiable (select list, checkbox or other limited entry component). - Requests that violate server-side access control rules. - A more comprehensive list of possible detection points is available here. When your application encounters such activity, your application should at the very least log the activity and mark it as a high severity issue. Ideally, your application should also respond to a possible identified attack, by for example invalidating the user's session and locking the user's account. The response mechanisms allow the software to react in realtime to possible identified attacks.

Secure Logging Design

Logging solutions must be built and managed in a secure way. Secure Logging design may include the following: - Allow expected characters only and/or encode the input based on the target to prevent log injection attacks. The preferred approach would be that the logging solution performs input escaping instead of dropping data: otherwise the logging solution might discard data which would be needed for a later analysis. - Do not log sensitive information. For example, do not log password, session ID, credit cards, or social security numbers. - Protect log integrity. An attacker may attempt to tamper with the logs. Therefore, the permission of log files and log changes audit should be considered. - Forward logs from distributed systems to a central, secure logging service. This will ensure log data cannot be lost if one node is compromised. This also allows for centralized or even automated monitoring.

Implementation

The following is a list of security logging implementation best practices. - Follow a common logging format and approach within the system and across systems of an organization. An example of a common logging framework is the Apache Logging Services which helps provide logging consistency between Java, PHP, .NET, and C++ applications. - Do not log too much or too little. For example, make sure to always log the timestamp and identifying information including the source IP and user-id, but be careful not to log private (such as username) or confidential data (such as business data) unless extra care is taken. - Pay close attention to time syncing across nodes to ensure that timestamps are consistent.

Vulnerabilities Prevented

- Brute-Force Attacks against Login-Mechanisms

References

- Logging Cheat Sheet
- OWASP Logging Guide

Tools

C10: Stop Server Side Request Forgery

Description

While Injection Attacks typically target the victim server itself, Server-Side Request Forgery (SSRF) attacks try to coerce the server to perform a request on behalf of the attacker. Why is this beneficial for the attacker? The outgoing request will be performed with the identity of the victim server and thus the attacker might execute operations with elevated operations.

Threats

Examples of this contain: - If an SSRF attack is possible on an server within the DMZ, an attacker might be able to access other servers within the DMZ without passing a perimeter firewall - Many servers have local services running on localhost, often without any authentication/authorization as localhost. This can be abused by SSRF attacks. - If SSO is used, SSRF can be used to extract tokens/tickets/hashes from servers. - etc.

Implementation

There multiple ways of preventing SSRF: - Input validation - If outgoing requests have to be made, check the target against an allow-list - If using XML, configure parsers securely to prevent XEE Be aware of Unicode and other Character transformations when performing input validation.

Vulnerabilities Prevented

- A10:2021 – Server-Side Request Forgery (SSRF)

References

- Server-Side Request Forgery Prevention Cheat Sheet
- A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages!

Tools

- SSRFmap

Final word

This document should be seen as a starting point rather than a comprehensive set of techniques and practices. We want to again emphasize that this document is intended to provide initial awareness around building secure software.

Good next steps to help build an application security program include:

1. To understand some of the risks in web application security please review the OWASP Top Ten .
2. A secure development program should include a *comprehensive list of security requirements* . Use Threat Modeling to identify potential security threats, derive security requirements, and tailor security controls to prevent those. Use standards such as the OWASP (Web) ASVS and the OWASP (Mobile) MASVS which provides a catalog of available security requirements along wiht the relevant verification criteria.
3. To understand the core building blocks of a secure software program from a more macro point of view please review the OWASP OpenSAMM project.

If you have any questions for the project leadership team, please contact with your questions, comments, and ideas at our GitHub project repository: https://github.com/OWASP/www-project-proactive-controls/issues

---

1. https://securitypatterns.io/what-is-a-security-pattern/ ↩