Secure Application Development Best Practice

**TABLE OF CONTENTS**

[Follow OWASP Security Design Principles 1](#_Toc525807538)

[1. Minimise attack surface area 1](#_Toc1927534954)

[2. Establish secure defaults 1](#_Toc1753282352)

[3. The principle of Least privilege 1](#_Toc874831360)

[4. The principle of Defence in depth 1](#_Toc1177751099)

[5. Fail securely 2](#_Toc1438496776)

[6. Don’t trust services 2](#_Toc1286289778)

[7. Separation of duties 2](#_Toc350375656)

[8. Avoid security by obscurity 2](#_Toc659585714)

[9. Keep security simple 2](#_Toc627739304)

[10. Fix security issues correctly 2](#_Toc1711190201)

[Tooling Types 2](#_Toc278699728)

[secret scanning 3](#_Toc883505945)

[dependance analysis 3](#_Toc237486197)

[license analysis 3](#_Toc1258791626)

[static vulnerability src analysis 3](#_Toc163260138)

[container analysis 3](#_Toc117290304)

[Coverage Gates 3](#_Toc1851412722)

[Software Best Practices 3](#_Toc1413290747)

[For Local Developer 3](#_Toc1144103261)

[For Continuous Integration Servers 3](#_Toc658620275)

[Cron / Regular Online Scanning 3](#_Toc218787507)

[Data interchange format 3](#_Toc38333595)

## Follow OWASP Security Design Principles

### 1. Minimise attack surface area

* Every time a programmer adds a feature to their application, they are increasing the risk of a security vulnerability.
* The principle of minimizing attack surface area restricts the functions that users are allowed to access, to reduce potential vulnerabilities.
* For example, you might code a search feature into an application. That search feature is potentially vulnerable to file inclusion attacks and SQL injection attacks.
* The developer could limit access to the search function, so only registered users could use it — reducing the attack surface and the risk of a successful attack.

### 2. Establish secure defaults

* This principle states that the application must be secure by default. That means a new user must take steps to obtain higher privileges and remove additional security measures (if allowed)
* Establishing safe defaults means there should be strong security rules for how user registrations are handled, how often passwords must be updated, how complex passwords should be, and so on.
* Application users may be able to turn off some of these features, but they should be set to a high-security level by default.

### 3. The principle of Least privilege

* The Principle of Least Privilege (POLP) states that a user should have the minimum set of privileges required to perform a specific task.
* The POLP can be applied to all aspects of a web application, including user rights and resource access. For example, a user who is signed up to a blog application as an “author” should not have administrative privileges that allow them to add or remove users. They should only be allowed to post articles to the application.

### 4. The principle of Defence in depth

* The principle of defense in depth states that multiple security controls that approach risks in different ways are the best option for securing an application.
* So, instead of having one security control for user access, you would have multiple layers of validation, additional security auditing tools, and logging tools.
* For example, instead of letting a user login with just a username and password, you would use an IP check, a Captcha system, logging of their login attempts, brute force detection, and so on.

### 5. Fail securely

* There are many reasons why a web application would fail to process a transaction. Perhaps a database connection failed, or the data inputted from a user was incorrect.
* This principle states that applications should fail in a secure way. Failure should not give the user additional privileges, and it should not show the user sensitive information like database queries or logs.

### 6. Don’t trust services

* Many web applications use third-party services for accessing additional functionality or obtaining additional data. This principle states that you should never trust these services from a security perspective.
* That means the application should always check the validity of data that third-party services send and not give those services high-level permissions within the app.

### 7. Separation of duties

* Separation of duties can be used to prevent individuals from acting fraudulently. For example, a user of an eCommerce website should not be promoted to also be an administrator as they will be able to alter orders and give themselves products.
* The reverse is also true — an administrator should not have the ability to do things that customers do, like order items from the front end of the website.

### 8. Avoid security by obscurity

* This OWASP principle states that security by obscurity should never be relied upon. If your application requires its administration URL to be hidden so it can remain secure, then it is not secure at all.
* There should be sufficient security controls in place to keep your application safe without hiding core functionality or source code.

### 9. Keep security simple

* Developers should avoid the use of very sophisticated architecture when developing security controls for their applications. Having mechanisms that are very complex can increase the risk of errors.

### 10. Fix security issues correctly

* If a security issue has been identified in an application, developers should determine the root cause of the problem.
* They should then repair it and test the repairs thoroughly. If the application uses design patterns, it is likely that the error may be present in multiple systems. Programmers should be careful to identify all affected systems.
* <https://patchstack.com/security-design-principles-owasp/>

## Tooling Types

### secret scanning

* purpose / recommendations of minium scan
* examples of use
* free software
* premium software

### dependance analysis

* purpose / recommendations of minium scan
* examples of use
* free software
* premium software

### license analysis

* purpose / recommendations of minium scan
* examples of use
* free software
* premium software

### static vulnerability src analysis

* purpose / recommendations of minium scan
* examples of use
* free software
* premium software

### container analysis

* purpose / recommendations of minium scan
* examples of use
* free software
* premium software

### Coverage Gates

* good quality gate levels are 0 critical / high
* should fail build and prevent commit

## Software Best Practices

### For Local Developer

* run xxx types of tools
* ensure quality gate in place to prevent low quality commits

### For Continuous Integration Servers

* run xxx types of tools on pr to develop
* ensure quality gate in place to prevent low quality commits

### Cron / Regular Online Scanning

* ensure regular dependance analysis for production software
* ensure quality gate in place to alert to high level vulnerabilities

## Data interchange format

* Santise all incoming user input, and protect against injection and malformed input
* Favour simple flat formats for sending payloads (aka json) over more complex data interchange formats (yaml / xml)  
  Reason: *complex formats often have hidden/obscure vulnerabilities / capabilities that require special processing / knowledge*  
  *see* [*https://en.wikipedia.org/wiki/Billion\_laughs\_attack*](https://en.wikipedia.org/wiki/Billion_laughs_attack)