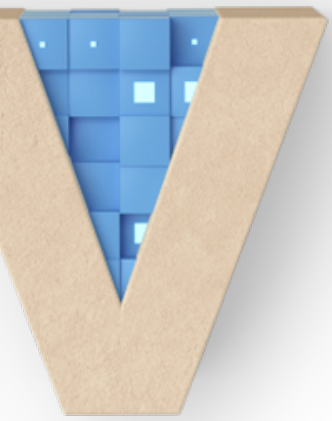
WHITEPAPER

# The definitive guide to secure coding principles

VULCAN.



# Introducing security early

Security is an essential part of every software architecture - or at least it should be. But more often than not, security is neglected at multiple levels of the company hierarchy. Engineers may view it as a distraction from their actual work of building products and fixing bugs; while managers see it as something that costs money and slows everything else down without delivering direct value.

While security might not seem like a benefit in the short run, it can mitigate significant risks for a company in the long run. For example, Facebook experienced a data breach that exposed the private data of 30 million users. Meanwhile, the decentralized finance app Poly - which runs on the Ethereum blockchain - lost roughly $600 million to a smart contract hack.

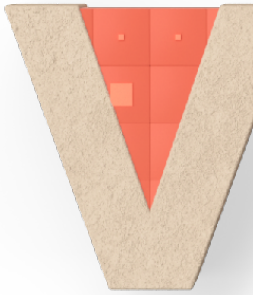
Often, failure to include security right from the start of a new application is a problem with multiple causes. Most engineers want to build secure systems but don't have the required skills. And while product managers aren't particularly fond of pouring time into non-feature-related tasks, they know what kind of impact an insecure application can have on the company.

Then one day, things go wrong, and an application gets hacked. Now, managers start hassling engineers with overburdening rules, which slow them down and kill their motivation; and while leading to secure software, this can also lead to a decline in overall software quality. After all, the level of security an application provides isn't the only measure potential customers go by when choosing what to buy.

A good security strategy should include a set of rules that can be autonomously followed and verified by engineers. The sooner in the development pipeline, the better. Wherever possible, you should apply the "shift-left" principle to every rule, so it can be verified early in the development process, when it's still cheap to make changes to the system.

In the first half of this white paper, we cover the **OWASP list of well-known security risks**, so you know what to look out for. The second half focuses on **best practices that can be included in your development pipeline.**

# The OWASP Top 10 web application security risks



In order to understand the threat landscape, let's start by looking at the OWASP Top 10 web application security risks. The list is updated annually by security experts from across the globe. The risks are ranked by their frequency of occurrence in web applications tested by OWASP auditors.

01

### BROKEN ACCESS CONTROL

While your application has access control, it doesn't always control access sufficiently. Examples of this includes users seeing data they shouldn't have access to, or allowing users to act with admin or moderation roles they haven't been assigned to. Cloud providers offer access control services: AWS has IAM and Azure Active Directory. But if these aren't set up correctly, it can leave your doors wide open for attacks.

Following the "principle of least privilege" and "deny by default" helps to mitigate this risk. In addition, missing permission configuration should lock down the system; this way, while you might forget to allow access, it's impossible for you to forget to restrict it.

02

### CRYPTOGRAPHIC FAILURES

This occurs when you attempt to secure your access keys with cryptographical methods but choose insufficient or outdated practices and libraries to do so, like hard-coding passwords into the source of your application or hashing passwords with MD5.

One solution is to classify your data and then choose appropriate methods to store it. Cloud providers offer certain key stores for access keys. Encrypt private data at rest and only store it if necessary. Also, try to keep your crypto libraries up to date. The sooner you know that an algorithm has been broken, the quicker you can replace a library that relies on it.

03

### INJECTION

When you fail to sanitize user input, users can add dubious data into your system. This can include—but is not limited to—JavaScript on the client side and SQL code, which your database will execute, which your backend will then execute.

To ensure this doesn't happen, you should never use your programming language's interpreter directly on user input. Keeping a layer of abstraction as an ORM between your users and your database helps too.

04

### INSECURE DESIGN

Insecure design is a broad topic, as it includes risks created when designing the architecture of an application. For example, standardized error messages can lead to exposure of sensitive data. This also includes designing data structures that contain information of different security levels.

The shift-left movement of security practices needs to reach into the system's design phase. Before actual code is written, the stakeholders have to think about threat models and confidentiality.

05

### CRYPTOGRAPHIC FAILURES

The people who deploy and maintain the software aren't always the same as those who implemented it. This can lead to installed plugins or open ports that aren't needed.

Automated hardening processes can help keep things in check even if the person handling it can't. Supplying your operations team with a configuration that only enables the minimal features required to run the application also clarifies optional parts. A simple example of this would be the network port of a server; most of the time, you would use the default ports, but sometimes you need to set a specific port. This could be the case when you are running multiple instances of one server on one computer, or if you need to adhere to security groups' configurations.

Today, it is considered best practice to use cloud configuration scanners. These are like linters for your config files which help you to avoid common errors when setting up your infrastructure.

06

### VULNERABLE AND OUTDATED COMPONENTS

Today, building software is essentially bundling together several third-party components to form a whole. This practice has become such a core task, it often includes unmaintained packages that contain security vulnerabilities. This includes all third-party software you use—your container or virtual machine runtimes have to be updated too.

Make sure to keep track of your dependencies, so you only include what you use. In addition, use an automated audit system that notifies you when one of your dependencies becomes compromised.

07

### IDENTIFICATION AND AUTHENTICATION FAILURES

Missing multi-factor authentication or exposing session identifiers in URLs are failures that make it easy for malicious actors to use automation tools to brute force their way into your system.

Make sure you're using up-to-date processes for authentication. Use multi-factor authentication when possible and make sure your password recovery doesn't rely on knowledge-based answers. Generate random, unguessable session IDs, and never store them in the URL.

08

### SOFTWARE AND DATA INTEGRITY FAILURES

This issue relates to data integrity checks such as cryptographic signatures and the widespread use of third-party libraries to build applications. You need to download these dependencies from a package repository, but these are compromised more and more often these days.

Always check the validity of the library signatures you use. If possible, use a tool like OWASP Dependency-Check, so you can be sure you're only including safe dependencies.

09

### SECURITY LOGGING AND MONITORING FAILURES

Applications log their activity to make it easier for the operations team to maintain them. But often, only the basics are logged, and information like important transactions or failed logins are missing. Unclear error messages are a problem too. Multicloud architectures only exacerbate the problem since you have to consolidate all the logs somehow to avoid wasting time looking in multiple places.

To avoid this, add security-related information to your logs, like sensitive data access or logins. Use append-only data storage for your logs, so they can't be modified later.

10

### SERVER-SIDE REQUEST FORGERY

In many applications, the backend doesn't have all the necessary data within easy reach; often, it has to request it from an upstream service in order to complete the task at hand. If the request includes data from unsanitized user input, an attacker could load sensitive data from unsuspecting internal services.

As with injections, data coming from users should always be sanitized. If requests are generated from user inputs, ensure that server-side requests are filtered by an allowlist of secure hostnames and ports.

# Security best practices

The first step is static analysis. The goal here is to check your code for problems in an automated fashion right when you write it.

### STATIC ANALYSIS

- A **code linter** is the next step here. It checks if you use idiomatic ways of implementation and often catches essential issues. The nice thing about a linter is that it also works on dynamically typed languages, like JavaScript or Ruby, to improve code quality even if no static type check is available.
- Static typing and linters are more general tools, and while they can improve security, that isn't their primary focus. Vulnerability scanners, like Checkmarx, are all about security issues. They check your code against giant databases of vulnerabilities and can find even the most obscure problems.

Overall, these static analysis tools free engineers from having to keep everything in mind all the time, and thus allow for greater focus on business-related tasks, without compromising security. It's a good idea to incorporate these scanners into your CI/CD pipeline so they're executed on the code even if an engineer forgot to do so on their local machine.

### STATIC ANALYSIS

What static analysis does before the code is built, dynamic analysis attempts to do after this stage. Dynamic analysis allows you to check your application's runtime behavior and thus to catch different issues than static analysis is able to.

- **Code coverage** checks which parts of your code were actually executed when the application ran. This can be used in conjunction with tests to see if you really tested crucial parts of your code.
- **Code instrumentation** can give you more insights while your application runs in the cloud. It makes your system more observable and allows you to see what resources are accessed while your customers use it.
- **Runtime-call tree graphs** are a way to see how your code behaves in terms of runtime complexity. This allows you to eliminate call stack overflows coming from too deeply nested recursions.

Dynamic analysis is an excellent addition to static analysis tools because they catch different problems.

## SOFTWARE COMPOSITION ANALYSIS

Software composition analysis (SCA), is an excellent addition to your security practices. Most of our software today is built by integrating multiple third-party software libraries. This practice saves time and money, but it also opens your software to the security vulnerabilities of these third-party libraries.

SCA services scan the most popular such libraries for you, notifying of any issues. One of the advantages of SCA is that it can be done independently of your software. The code of such libraries lives in open source repositories that are publicly available for the SCA scanners. When using such a service, you can read the results of the scans that have already happened in the past for your immediate benefit.

## TESTING

Automated testing has become an industry standard. Executing your code and checking if it runs before shipping it to your customers is crucial. There are many types of tests, each of which with a different focus. Let's look at the most prominent ones in the field of security:

- **Unit tests** are automated tests that focus on a small unit of code. What that means has to be decided on a case-to-case basis. You can use unit tests to check your sanitation procedures in an encapsulated way to ensure they're airtight.
- **Integration tests** check the interaction of multiple units of the application, making them more complex than unit tests. They can help to locate data leaking from one part of the application to another.
- **End-to-end tests** are the most complex tests; they try to mimic complete user interaction with the application. After making sure your application is secure, these types of tests can be used to check if it's still usable. They also help to replicate problematic user interactions that reveal security vulnerabilities.
- **Property tests,** like unit tests, try to run a specific part of the application. They differ from unit tests because they generate multiple unit inputs instead of just some predefined inputs. This type of testing can uncover the sort of edge cases an attacker could use to hack your system.
- **Fuzzing** is another type of testing. It's a bit like a property test; the difference is that a property test uses type annotations to generate inputs for a unit, while fuzzing brute forces all inputs into an executable, even invalid ones. This testing method is used mainly to harden binary executables.
- **Mutation** tests for your tests. The rationale is that if you introduce a bug into your codebase, your tests should fail. Mutation tests modify—or mutate—your code base with minor changes and check if your tests fail. This helps to locate weakly tested code as an alternative to the code coverage approach, which only checks for executed lines of code.

### CODE REVIEWS

Code reviews are a manual task executed by your team members. One engineer writes code, while another reads it to check there aren't any issues with it. This is often necessary because code can impact other parts of the architecture implicitly. Having a security professional review critical code paths is recommended to improve security.

However, people tend to switch to autopilot when they have to complete repeated manual tasks, which undermines the whole premise of code reviews: finding issues automated tools can't detect. Checklists for code reviews can help here by ensuring that no vital step of the review is omitted.

### REMEDIATION INTELLIGENCE

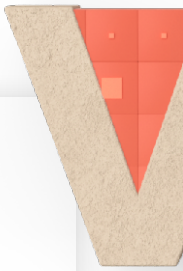
Finding holes in your defense is just the first step, but fixing security vulnerabilities isn't always straightforward. Often a naive fix, which doesn't take root causes into account, can lead to other issues in the future. Fixing security problems sometimes means writing code in a non-obvious way, for example, when you have to override parts of the memory manually to make sure secrets can't be extracted from it after the program ends.

That's where remediation intelligence comes into play. Using BI dashboards powered by advanced analytics, it takes the problems discovered by the tools discussed and provides actionable insights to guide your next steps.

# Conclusion

Developing secure systems is crucial for your business, and it should be a top priority for every R&D team. Ensuring security best practices is key, but this requires proper guidance. Failure to inform your developers of what's essential can lead to serious problems in the long run.
Not only do you have to make sure that your teams are notified when holes in your security are found; you also have to supply them with the right information to identify and fix the problem. Developers need to know what to look out for, how to look out for it, and how to remediate vulnerabilities when they are found.

The Vulcan Cyber® risk management platform offers end-to-end cyber risk management—finding, prioritizing, and fixing your vulnerabilities. It provides your engineers with actionable remediation data right when they need it, covering all your attack surfaces—from networking to cloud infrastructure and application code. It also integrates with your existing tools (Jira, Slack, Microsoft Teams, ServiceNow, and more), making collaboration easier than ever. Embrace the DevSecOps approach with your risk remediation to get the most from your tools and your team. Request a demo to start owning your risk.



## About Vulcan Cyber

Vulcan Cyber® breaks down organizational cyber risk into measurable, manageable processes to help security teams go beyond their scan data and actually reduce risk. With powerful prioritization, orchestration and mitigation capabilities, the Vulcan Cyber risk management SaaS platform provides clear solutions to help manage risk effectively. Vulcan enhances teams' existing cyber environments by connecting with all the tools they already use, supporting every stage of the cyber security lifecycle across cloud, IT and application attack surfaces. The unique capability of the Vulcan Cyber platform has garnered Vulcan recognition as a 2019 Gartner Cool Vendor and as a 2020 RSA Conference Innovation Sandbox finalist.

# Own your risk.

CONTACT US

VULCAN.