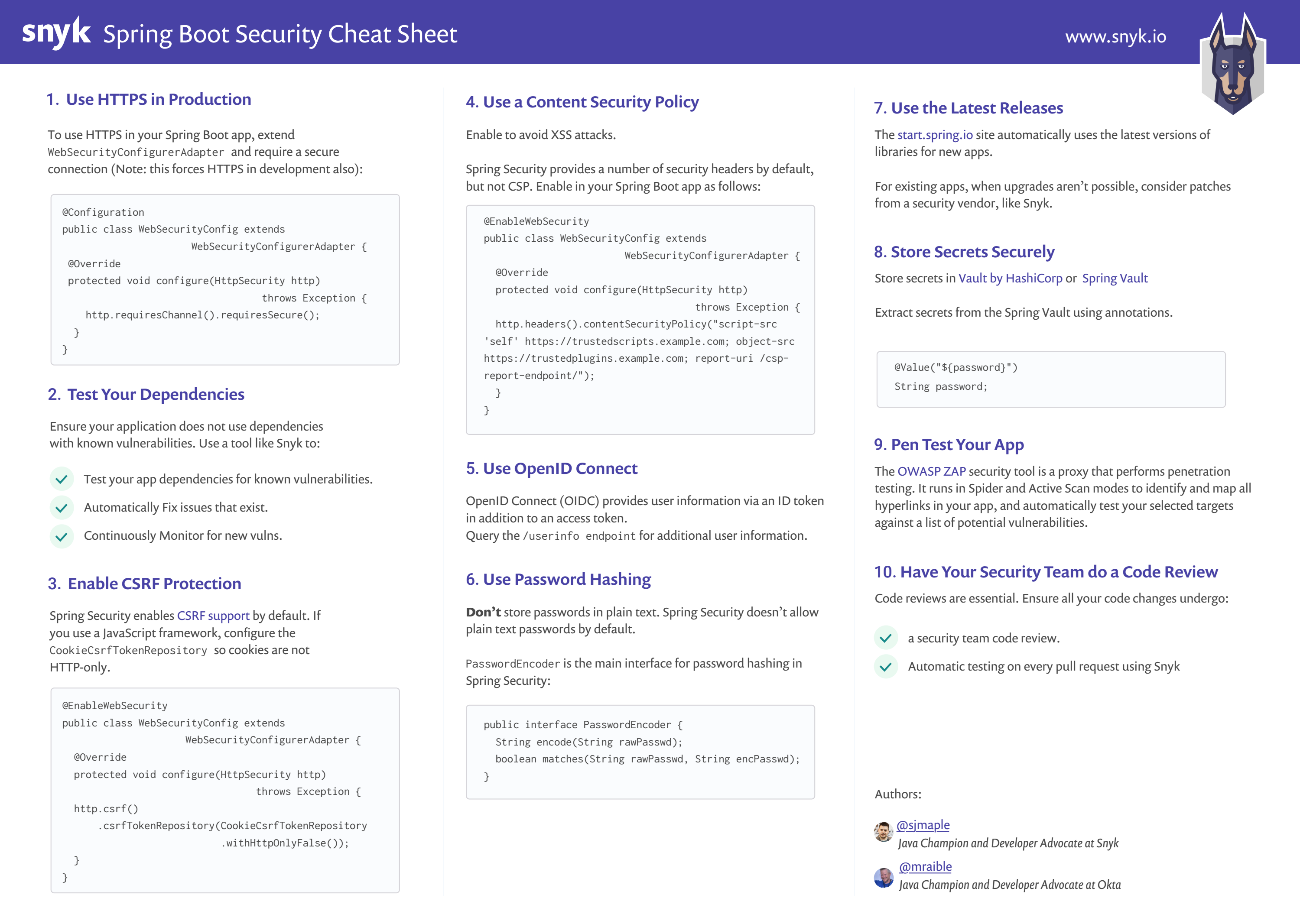
https://snyk.io/blog/spring-boot-security-best-practices/

[**APPLICATION SECURITY**](https://snyk.io/blog/category/application-security/)[**ECOSYSTEMS**](https://snyk.io/blog/category/ecosystems/)

# **10 Spring Boot security best practices**



**Simon Maple, Matt Raible**August 16, 2018

[](https://res.cloudinary.com/snyk/image/upload/v1534422834/blog/Spring_Boot_Security_Cheat_Sheet.pdf)

Working with Spring Boot and don’t know where to start when it comes to security? You need to worry no more! This cheatsheet proposes best practices on how developers and maintainers can improve their Spring Boot security.

Spring Boot is one of the most used frameworks in the Java ecosystem because it dramatically simplifies the development of Spring applications. For this reason, it would be wise of you to try and improve the overall security posture of your Spring Boot applications. And we are here to help you do just that.

This post was originally written in a blog post form, and in more detail, with [**Matt Raible**](https://twitter.com/mraible), fellow Java Champion and Developer Advocate at [**Okta**](https://www.okta.com/). We both work for companies in the security industry, love Java, and want to help developers create more secure applications, so what better to produce than a good looking cheatsheet on Spring Boot Security!

Without further ado, let’s jump into our list of Spring Boot security best practices. Note that the original, more detailed post can be found on the [**Okta blog**](https://developer.okta.com/blog/2018/07/30/10-ways-to-secure-spring-boot).

### **What is Spring Boot security?**

[**Spring Boot security**](https://snyk.io/blog/spring-boot-security-best-practices/) can mean different things. In general, it is adding the Spring Security framework to your Spring Boot web application by including the Spring Boot security starter dependency. Spring Security is an authentication and access-control framework and can be easily included in a Spring Boot application. On the other hand, Spring Boot security is more than just including the Spring Security framework. This cheatsheet focuses on the broader topic of Spring Boot security and how to secure your application created with Spring Boot.

## Spring Boot security best practices

The following curated list will go beyond just introducing Spring Security for authentication and authorization in your Spring Boot application. It focuses on the broader Spring Boot security strategy and covers the following topic:

1. Use HTTPS in production
2. Test your dependencies and find Spring Boot vulnerabilities
3. Enable CSRF protection
4. Use a content security policy for Spring Boot XSS protection
5. Use OpenID Connect for authentication
6. Use password hashing
7. Use the latest releases
8. Store secrets securely
9. Pen test your app
10. Have your security team do a code review

## 1. Use HTTPS in production

TLS/SSL certificates used to be expensive, and HTTPS was considered slow. Machines have become much faster, solving the performance problem, and [**Let’s Encrypt**](https://letsencrypt.org/) provides free TLS certificates. These two developments have changed the game and caused TLS to become mainstream.

To force HTTPS in your Spring Boot app, you can extend WebSecurityConfigurerAdapter and require a secure connection.

@Configuration

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.requiresChannel().requiresSecure();

}

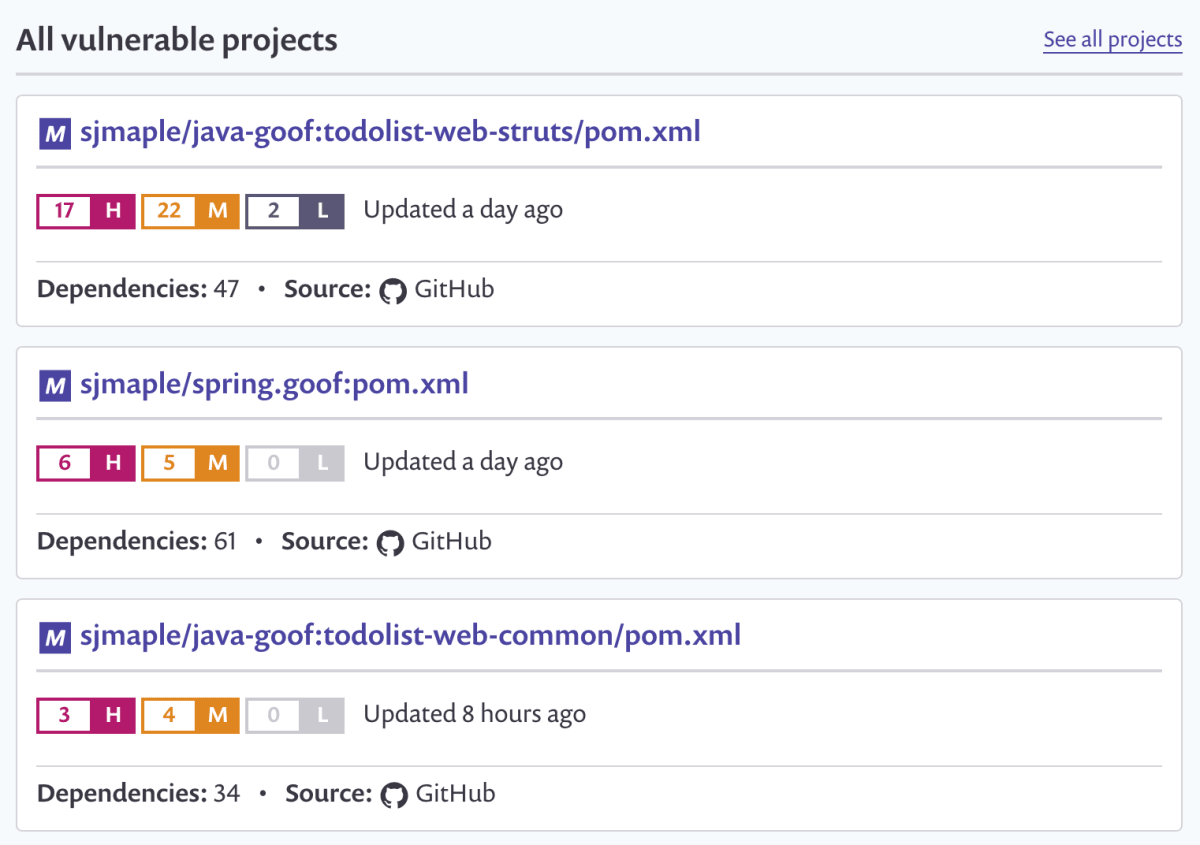
}

Cloud providers can greatly simplify TLS certificates. [**Amazon Certificate Manager**](https://aws.amazon.com/certificate-manager/) is exactly like Let’s Encrypt except it’s built into all of the AWS products/services by default. It lets you provision 100% free SSL certs and handles automatic renewal, etc., with literally zero effort/config. Heroku has [**Automated Certificate Management**](https://devcenter.heroku.com/articles/automated-certificate-management) too.

## 2. Test your dependencies and find Spring Boot vulnerabilities

There’s a good chance you don’t know how many direct dependencies your application uses. It’s extremely likely you don’t know how many transitive dependencies your application uses. This is often true, despite dependencies making up the majority of your overall application and can contain Spring Boot security vulnerabilities. Attackers target open source dependencies more and more, as their reuse provides many victims for a malicious hacker. It’s important to ensure there are no known vulnerabilities in the entire dependency tree of your application.

Snyk tests your application build artifacts, flagging those dependencies that have known Spring Boot security vulnerabilities. It provides you with a list of vulnerabilities that exist in the packages you’re using in your application as a dashboard.



Additionally, it will suggest upgrade versions or provide patches to remediate your security issues, via a pull request against your source code repository. Snyk also protects your environment, by ensuring that any future pull requests raised on your repository are automatically tested (via webhooks) to make sure they do not introduce new known Spring Boot security vulnerabilities.

Snyk is available via a web UI as well as a CLI, so you can easily integrate it with your CI environment, and configure it to break your build when vulnerabilities exist with a severity beyond your set threshold.

## **Find and fix Spring Boot vulnerabilities for free**

Find vulnerabilities in seconds. Fix quickly with an automated pull request.

[**Fix for free**](https://app.snyk.io/login?cta=sign-up&sect=body&page=10-spring-boot-security-best-practices)

## 3. Enable CSRF protection

[**Cross-Site Request Forgery**](https://snyk.io/learn/cross-site-scripting/) is an attack that forces a user to execute unwanted actions in an application they’re currently logged into.

Spring Security has [**excellent CSRF support**](https://docs.spring.io/spring-security/site/docs/current/reference/html5/) that’s on by default. If you’re using Spring MVC’s <form:form> tag or Thymeleaf and @EnableWebSecurity, the CSRF token will automatically be added as a hidden input field.  To make this part of your Spring Boot security strategy, you have to add the Spring Security starter as a dependency

If you’re using a JavaScript framework like Angular or React, you will need to configure the CookieCsrfTokenRepository so JavaScript can read the cookie.

@EnableWebSecurity

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.csrf()

.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());

}

}

Spring Security automatically adds a secure flag to the XSRF-TOKEN cookie when the request happens over HTTPS. Spring Security doesn’t use the SameSite=strict flag for CSRF cookies, but it does when using Spring Session or WebFlux session handling.

## 4. Use a content security policy for Spring Boot XSS protection

[**Content Security Policy**](https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP) (CSP) is an added layer of security that helps mitigate [**XSS (cross-site scripting)**](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) and data injection attacks. To enable it, you need to configure your app to return a Content-Security-Policy header. You can also use a <meta http-equiv="Content-Security-Policy"> tag in your HTML page.

Spring Security provides a number of [**security headers by default**](https://docs.spring.io/spring-security/site/docs/current/reference/html5/). Spring Security does not add a CSP by default. You can enable the CSP header in your Spring Boot app using the configuration below.

@EnableWebSecurity

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.headers()

.contentSecurityPolicy("script-src 'self' https://trustedscripts.example.com; object-src https://trustedplugins.example.com; report-uri /csp-report-endpoint/");

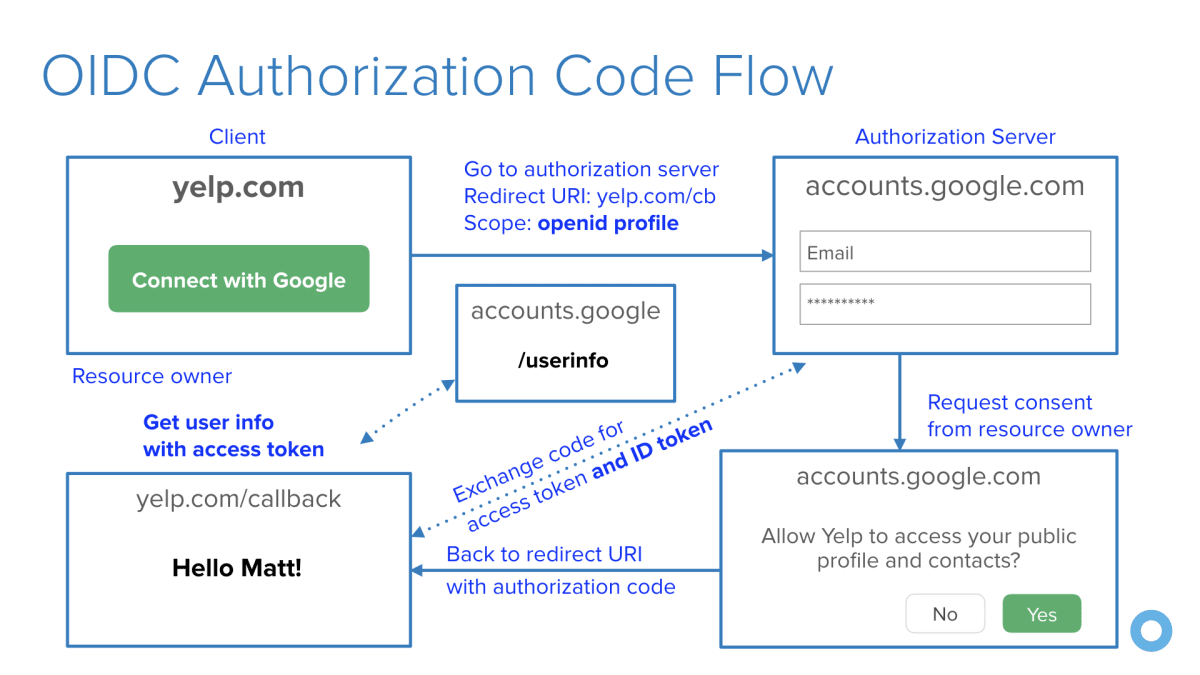
}

}

## 5. Use OpenID Connect for authentication

OpenID Connect (OIDC) is an OAuth 2.0 extension that provides user information. It adds an ID token in addition to an access token, as well as a /userinfo endpoint that you can get additional information from. It also adds an endpoint discovery feature and dynamic client registration.

The diagram below shows how OIDC works for authentication.



## 6. Use password hashing

Storing passwords in plain text is one of the worst things you can do for the security of your app. Luckily, Spring Security doesn’t allow plain text passwords by default. It also ships with a [**crypto module**](https://docs.spring.io/spring-security/site/docs/current/reference/html5/) you can use for symmetric encryption, key generation, and password hashing (a.k.a., password encoding).

PasswordEncoder is the main interface for password hashing in Spring Security and looks as follows:

public interface PasswordEncoder {

String encode(String rawPassword);

boolean matches(String rawPassword, String encodedPassword);

}

Spring Security provides several implementations, the most popular being BCryptPasswordEncoder and Pbkdf2PasswordEncoder.

## 7. Use the latest releases

There are various reasons to regularly upgrade the dependencies in your application. Security is one of the most important reasons that will give you the motivation to upgrade. The [**start.spring.io**](https://start.spring.io/) starter page uses the most recent versions of Spring packages as well as dependencies, where possible.

Infrastructure upgrades are often less disruptive than dependency upgrades, as library authors vary in their sensitivity to backward compatibility and behaviour changes between releases. That being said, you have three options when you find a security vulnerability in your configuration: Upgrade, Patch or Ignore. Upgrades are the safest, in terms of the overall health of your application, but often they’re not always an option. When this is the case, patches can eliminate vulnerabilities from your package, which you can often get from a security specialist, like Snyk. Ignoring a vulnerability is, of course, an option, but not a good one. Perhaps you know of a vulnerability, but do not believe it is directly exploitable. Keep in mind that it might not be in your application flow today, but at some point, a developer might add additional code that uses a vulnerable path.

## 8. Store secrets securely

As part of your Spring Boot security strategy, we also need to think about your data. Sensitive information such as passwords, access tokens, etc., should be handled with care. You cannot leave these around, pass them in plain text, or be predictable if keeping them in your local storage. As (GitHub) [**history has proved**](https://github.com/search?q=removed+password&type=Commits) time and time again, developers do not think carefully enough about how they store their secrets.

A good practice is to store secrets in a vault that can be used to store, provide access to, and even generate credentials to services that your application may use. [**Vault by HashiCorp**](https://www.vaultproject.io/) makes storing secrets trivial, as well as offering a number of additional services. You can also integrate with common authentication mechanisms such as LDAP to obtain tokens.

If this interests you, be sure to invest some time looking at the [**Spring Vault**](https://spring.io/projects/spring-vault) which adds an abstraction over the HashiCorp Vault, providing Spring annotation based access for clients, allowing them to access, store and revoke secrets without getting lost in the infrastructure. The following code snippet shows how easy it is to extract a password from the Spring Vault using an annotation.

@Value("${password}")

String password;

## 9. Pen test your app

The [**OWASP ZAP**](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project) security tool is a proxy that performs penetration testing against your live application at runtime. It’s a popular (over 4k stars) free, open source project that is hosted on [**GitHub**](https://github.com/zaproxy/zaproxy).

Two approaches OWASP ZAP uses to find vulnerabilities are Spider and Active Scan. The Spider tool starts with a seed of URLs, which it will access and parse through each response, identifying hyperlinks and adding them to a list. It will then access these newly found URLs and recursively continue, creating a map of URLs for your web application. The Active Scan tool will automatically test your selected targets against a list of potential vulnerabilities. It provides you with a report that shows where your web application is exploitable, with details about the vulnerability.

## 10. Have your security team do a code review

Code reviews are essential for any high performing software development team. At Okta, all the production code and official open source projects are required to go through an analysis from our expert security team. You might not have security experts at your company, but if you’re dealing with sensitive data, maybe you should!

Be sure to [**download the Spring Boot Security cheat sheet**](https://res.cloudinary.com/snyk/image/upload/v1534422834/blog/Spring_Boot_Security_Cheat_Sheet.pdf) and pin it up somewhere to remind you of some of the best practices you should follow if you’re a developer using Spring Boot in your applications.

## FAQ

### **How do I add security to Spring Boot?**

When [**Adding Spring Security**](https://snyk.io/blog/spring-boot-security-best-practices/) to your Spring Boot application begins with adding the security starter dependency.

<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-security</artifactId>  
</dependency>

By default, authentication is enabled when the starter dependency is included. You can configure the credentials by setting the properties spring.security.user.name and spring.security.user.password

### How do I disable Spring Security in Spring Boot?

By default, [**Spring Security**](https://snyk.io/blog/spring-boot-security-best-practices/) is enabled whenever you include the spring-boot-starter-security package. This can by easily disabled by excluding the SecurityAutoConfiguration in the application.properties file.

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration