Coding Practices and Recommendations of Spring Security for Enterprise Applications

Mazharul Islam*, Sazzadur Rahaman*, Na Meng*, Behnaz Hassanshahi†, Padmanabhan Krishnan†, Danfeng (Daphne) Yao*  
Virginia Tech, Blacksburg, VA*, Oracle Labs, Australia†  
{mazharul, sazzad14, nm8247, danfeng}@vt.edu, {behnaz.hassanshahi, paddy.krishnan}@oracle.com

Abstract—Spring security is tremendously popular among practitioners for its ease of use to secure enterprise applications. In this paper, we study the application framework misconfiguration vulnerabilities in the light of Spring security, which is relatively understudied in the existing literature. Towards that goal, we identify 6 types of security anti-patterns and 4 insecure vulnerable defaults by conducting a measurement-based approach on 28 Spring applications. Our analysis shows that security risks associated with the identified security anti-patterns and insecure defaults can leave the enterprise application vulnerable to a wide range of high-risk attacks. To prevent these high-risk attacks, we also provide recommendations for practitioners. Consequently, our study has contributed one update to the official Spring security documentation while other security issues identified in this study are being considered for future major releases by Spring security community.

I. INTRODUCTION

Application frameworks enable reusing software designs at the architecture level by capturing the common abstractions of an application domain [1]. Spring is the most popular application framework for enterprise Java applications [2], [3]. Spring security [4] offers reusable authentication and authorization modules for enterprise applications that are written in Spring. It also provides default protection against common web-application related vulnerabilities, e.g., CSRF protection, including common security response headers. Spring security is highly customizable by design to enable seamless integration with various use-cases. Unfortunately, the abuse of such customization capabilities can be a great source of application insecurity. For example, in [5], authors observed a common trend of disabling CSRF protection for convenience in StackOverflow posts. Without careful consideration, such customization can render a web application vulnerable to classic CSRF attacks.

Prior research on security issues arising from reusable software components mostly focuses on library API misuse [6]–[12], while framework misconfigurations are largely unexplored. For example, [5], [6], [9], [11] focus on understanding the nature of security API misuses. [6], [8] showed the dangers of misusing application-level SSL/TLS and cryptographic APIs. Researchers extensively studied the role of StackOverflow’s misleading advices [5], [9], [13], poor API designs [14], lack of proper guidelines [15], etc., behind this insecurity. Most of the existing methods to detect security API misuses rely on static code analysis [7], [8], [10], [12], [16]. Although, misconfiguring of security modules in application frameworks has great potential to cause insecurity, their nature, severity, prevalence, and detection feasibility are still mostly unknown.

In this paper, we present a thorough study of framework misconfiguration vulnerabilities in Spring Security. Our goal is to identify various classes of these vulnerabilities (referred to as security anti-patterns), their nature, severity, and prevalence. Specifically, we pose the following research questions. What are the common security anti-patterns in enterprise Spring security applications? How severe are they? Most importantly, how prevalent are they in real-world enterprise software?

To find answers, we took a measurement-based approach. We manually analyzed 28 Spring-based applications hosted on GitHub to observe any insecure customization (i.e., security anti-patterns) of Spring security’s i) authentication and ii) protection against exploits features. We also studied the security of the default configurations of these features. Our analysis discovered 6 types of security anti-patterns. We observed that programmers tend to intentionally disable CSRF protections, store secrets insecurely, use lifelong expiration of access tokens, etc. Our analysis of Spring security’s default configuration revealed 4 major vulnerabilities. Our analysis found that Spring security uses 10 as the default strength (2^10 number of rounds) in Bcrypt during password encoding, while it is recommended to use at least 16 [17] to be secured. We also found that Spring Security uses insecure MD5 hashing to generate the “remember me” cookie. Most importantly, we identified that Spring security does not offer any throttling policy to limit the number of requests by users during API invocation. This insufficiency might lead to denial of service (DoS) attacks to applications using Spring security’s OAuth functionality. Our findings on 4 major vulnerabilities of Spring security’s default configuration resulted in one update to Spring security’s official documentation, while other issues are being considered for future major releases after we disclosed them to the Spring security community.

In summary our contributions are as follows.

• Our analysis of 28 applications identified 6 common Spring security anti-patterns that undermine its security

This work has been supported by the National Science Foundation under Grant No. CNS-1929701.
guarantees. During our analysis, we discovered 17 instances of disabling CSRF tokens, 14 instances of hard coded JWT signing key, 17 instances of storing secrets insecurely. We also analyze the security risk associated with them and highlight recommendations for practitioners on how to avoid the anti-patterns and thus improve security.

- Our analysis of Spring security’s default configuration revealed 4 major vulnerabilities, including the insecure use of Bcrypt for password encoding, the use of MD5 hash to generate “remember-me” cookie. We also identified that the lack of throttling policy per API key is susceptible to denial of service attacks.
- We divided our dataset into two groups, i.e., real-world applications (8) and demo (20) projects and cross-analyzed 6 security anti-patterns across the groups. Our analysis revealed that the anti-pattern’s count ratio is higher in demo projects than the real-world applications. However, the nature of the anti-patterns is similar across the two groups.

II. Threat Model and Methodology

In this section, first, we present the threat model and then, discuss the methodology of our study.

A. Threat model

Existing cryptographic API misuse studies (e.g., [6]–[12]) are not specific for Spring security framework. Enterprise security issues in Spring security, such as the abuse of customization of reusable components or improper security management policies, are not well examined, with a few exceptions. For example, studies in [18], [19] designed authentication and authorization patterns and access control policies of Spring security. In comparison, our paper aims to report security anti-patterns in Spring projects and their security threats. Specifically, we focus on the misconfigurations of two Spring security features, i.e., i) authentication and ii) protection against exploits. We also study the security status of default configurations of these features.

Authentication. Spring security offers 9 types of authentications [20]. We analyze the use of 4 of these authentication mechanisms as follows, i) username-password, ii) “remember me” cookie, iii) OAuth 2.0 and iv) Java authentication and authorization service (JAAS)-based authentication. Username-password based authentication is the most common way to authenticate users while “remember me” cookie facilitates remembering users between sessions. JASS and OAuth2.0 based authentication are a bit different since they delegate the authentication requests to their corresponding JASS server and OAuth 2.0 provider respectively.

Misconfiguring them can lead to a wide range of problems i.e., leaking application secrets (e.g., access tokens, passwords, etc), enabling man-in-the-middle (MitM), denial of service (Dos) attacks, etc.

Protection against exploits. Spring security also provides protections against common exploits. In most of the cases, these protections are enabled by default. To protect against CSRF attacks, Spring security offers the following protections; i) CSRF token and ii) “SameSite Attribute”-based protection. CSRF token-based protection ensures the presence of CSRF token in the HTTP request header to indicate its legitimacy. In “SameSite Attribute”-based protection, the browser only sends the “SameSite” session cookie if and only if both the requested resource and the resource in the top-level browsing context match the cookie. Spring security also enables sending common HTTP security headers by default, including, HTTP Strict Transport Security, X-Frame-Options, X-XSS-Protection, etc. Spring security also enables Strict Transport Security by default to redirect HTTP traffic to HTTPS.

B. Methodology of the study

To systematically discover security anti-patterns, we develop the following methodology. First, we collect a dataset of real-world enterprise application source codes that uses Spring security. Then, we carefully analyze and collect their security configurations by using a descriptive coding technique [21]. After finding the use of a security feature, we extensively match its configuration with the following seven knowledge-base of common security issues; i) Common Weakness Enumeration (CWE) [22], ii) Openstack security anti-patterns alert list [23], iii) Spring security official reference guide [24], iv) Apigee Edge anti-patterns [25], v) Snyk vulnerability database [26], vi) previous research work on security anti-patterns [27]–[31], vii) RFC documents [32]–[35]. If any of these knowledge-base indicates an insecure configuration, we analyze severity in the context of their usage. If an insecure configuration is the result of a customization, we mark it as a security anti-pattern. Otherwise, we mark it as an insecure default configuration.

III. Security Anti-patterns in Spring Security

In this section, first, we will present our analysis result on the collected data and then briefly illustrate each common security misuse and their severity.

Data collection. We collected the source code of 28 applications hosted on GitHub that uses Spring security. 8 of the selected projects are real-world enterprise applications and 20 of them are demo projects with example use of Spring security framework. We considered the following three criteria to filter them [36]:

- #Forks. The number of times the project has been forked.
  This gives an indication that these repositories have been adopted widely by other developers in their own code base [37].
- #Stars. The number of times the project has been starred by other developers which ensures that the curated repository is popular [38] among other developers.
- Originality. The project is not a clone or copy of another existing project.
A. Analysis Result

After analyzing the usage of Spring security framework of the selected projects, we identified 6 Spring security anti-patterns and 4 insecure default behaviors. Table I presents these security misuses, with their reference knowledge-base, affecting features, threat, severity, and counts in 28 projects. After that, we divided the dataset into two groups i.e., i) 8 real-world applications and ii) 20 demo projects. Then we cross- checked the security misuse instances across them. Table II, presents the results of our analysis. Although the misuse count ratio is higher in demo projects than the real-world projects, the nature of the misuse cases are vastly overlapped (Column 3 in Table II). It will be interesting to see whether (and how) developers are being influenced by these insecure demo codebases that can be directly copied.

Next, we describe each of them, their severity, and recommend suggestions on how developers can properly resolve them. After that, we present several interesting case studies.

<table>
<thead>
<tr>
<th>Anti-patterns</th>
<th>Real-world projects (8)</th>
<th>Demo projects (20)</th>
<th>Common cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) lifelong valid access tokens</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>(2) Absence of state param</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>(3) Fixed secrets to sign JWT tokens</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>(4) Storing secrets in insecure places</td>
<td>5</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>(5) Disabling CSRF protection</td>
<td>6</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>(6) Not using TLS</td>
<td>4</td>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>

B. Common Spring security anti-patterns

1) Using lifelong valid access tokens: Spring security allows the developers to specify an expiration time for each randomly generated access tokens. Developers want access token with a long lifetime as they are easier to manage. However, on the other hand, long lifetime increases the risk of reply attacks if any access token gets leaked. The general advise from [23], [25], [32] is to keep the life time just a bit longer than a normal user session time which can be generalized to a period of 15 minutes to 2 hours depending on different use cases. However, we have noticed a security anti-pattern among developers of setting lifetime of access token primarily arbitrary long in the range of 10-20 days as shown on listing 1.

```java
// setting unnecessary long lifetime of 10 days
```

To avoid this security anti-pattern, we suggest the developers to minimize the lifetime of access token as much as possible so that whenever an attacker tries to reply the previously leaked secret access token it would already pass the expiration period. We also suggest to leverage refresh token to facilitate the user to provide a new access token effectively each time when the lifetime of an access token expires.

2) Absence of state parameter in OAuth 2.0 Redirect URLs: The continuous influx of increasing popularity of OAuth 2.0 [39] among developers motivated the active Spring security community to introduce the support for OAuth 2 in its lastest release [40]. One of the most crucial parts among many of OAuth 2.0 authorization framework is sending the auth_code generated by authorization server to client applications via redirection URLs [32]. Interestingly redirect URL has a well define structure with guessable query parameters (1st redirect URL on Fig. 1). This can enable the attacker to construct a similar but malicious redirect URL by replacing user’s auth code with their own auth code (2nd redirect URL on Fig. 1). By making the victim user clicking on this malicious redirect URL, attacker can perform a forced login CSRF attack [41].

To prevent this, RFC-6819 [33] has recommended a strict guideline to add an additional state parameter - the value of which is randomly generated (as shown on the 3rd redirect URL in Fig. 1). In this way the attacker won’t be able to guess the value of state parameter and construct a malicious redirect URL. However, we have noticed in contrast to this strict recommendation, developers tend not to use the additional state parameter in redirect URL rendering the client application vulnerable to previously mentioned forced login attack CSRF attack. Hence we consider the missing of state parameter in redirect URL as a security anti-pattern.
The proper way to handle this security anti-pattern is to randomly generate a value and add this value to the state parameter in the redirect URL as shown on listing 2.

```java
Listing 2. Adding state param in redirect_URL
public String getTOKEN1(@RequestParameter String code)
{
    ... params.add("grant_type","authorization_code");
    params.add("code",code);
    params.add("client_id","aiqiyi");
    params.add("client_secret","secret");
    // Randomly generated value of state param
    params.add("state","127621437303857");
    ... return token;
}
```

3) Using fixed secrets to sign JWT tokens: Spring security facilities the use of JSON Web Tokens (JWT) [42] to authenticate users by adding JwtTokenFilter to the DefaultSecurityFilterChain with only a few lines of additional code. The way JWT works is that there a set of claims embedded inside JWT and the server signs these claims cryptographically using secret key(s). The user must present these cryptographically signed claims to the server and then the server verifies them to check the authentication of these presented claims. This design allows the server to be stateless and consequently scalable which is one of the major reasons behind the emerging popularity of JWT.

However, we noticed that developers tend to sign the claims of JWT cryptographically by predictable fixed secret key which makes it inherently vulnerable [8] and relatively easy for attackers to crack the fixed secret key by brute force attacks.

To circumvent this security anti-pattern and to handle application specific secrets securely, Spring vault facilitates a succinct solution. Spring vault which is an abstraction layer on top of HashiCorp vault [43] providing annotation based access for clients to store and retrieve secrets in a secure way as shown in listing 4.

```java
Listing 4. Retrieving secrets from Spring vault
@Value("${clientPassword}")
String client_password;
```

4) Storing secrets in insecure places: To avoid application specific secrets e.g., password, DB-root password from being leaked developers need to avoid keeping them in unsafe places (e.g., plain-text, local storage). We have noticed a security anti-pattern among developers to write the application specific secrets to application.yml configuration file in plain text as shown in listing 3.

```java
Listing 3. Stroing secrets insecurely in application.yml
spring:
    datasource:
        username: root
        password: uDtmALPgsfKgYsZGlulXL13O
```

5) Disabling CSRF protection: Spring security by default secures the web applications by defining a method csrf() and implicitly enabling this function invocation for each state-changing request (e.g., PATCH, POST, PUT, DELETE) [44]. We have observed a recurring insecure practice among developers to manually disabling the default CSRF protection as shown on listing 5.

```java
Listing 5. Manually disabling default CSRF protection
@override
protected void configure(HttpSecurity hs) throws Exception {
    hs.csrf().disable();
}
```

Spring security’s inherently different CSRF protection mechanism is a plausible answer to the prominence of this insecure coding practice. Spring security makes the CSRF token inaccessible to the front-end of the application by setting httpOnly to true. Being unable to access the CSRF token, front-end JavaScript frameworks (e.g., Angular JS, Vue JS, Laravel etc.) will throw configuration errors. To avoid these errors developers primarily tend to turn off the default CSRF protection the developers seeking a short workaroud way without understanding the insecurity associate with doing so. The proper way to circumvent this error is to
set `httpOnly` to false as shown on listing 6. That being said, we want to point out that even though setting the `httpOnly` to false can reveal the CSRF token to JavaScript loaded from the same domain, third party JavaScript loaded from different domain will not be able to access the access token because of same-origin policy; thereby defeating Cross Site Scripting attacks.

We further want to emphasize that disabling CSRF protection itself is not always a security problem when authentication is done via bearer tokens. However, it can be a severe security problem when Spring security will perform authentication based on the authentication cookie (i.e., JESSIONID) stored on the user’s browser.

6) Not using TLS for HTTP communications: The security of many components of authentication and authorization packages in Spring security (e.g., oauth 2.0, SAML 2.0, CAS, OpenID connect) recommend, and in some cases mandate the use of TLS. However, our analysis found that developers tend to avoid the use of TLS in many places and show a similar trend highlighted in previous study [5], [12], [45].

Although, it is difficult to create, install, find, and validate TLS certificates in development environment, we suggest the developers to enable TLS in the production environment as suggested in [6].

C. Insecure default configuration

1) Using BCrypt with insecure params: Spring security supports multiple ways to implement a PasswordEncoder. A brief summary of them is presented in section A. Here, we focus on BCryptPasswordEncoder, which is popularly used to encode passwords. We found that the default configuration of BCryptPasswordEncoder is vulnerable to brute-force attacks.

BCryptPasswordEncoder is based a deliberately slow hashing function Bcrypt [46]. This slowness or number of rounds in Bcrypt is one of the key factors which makes it resistant to do feasible brute-force attack. The security strength (i.e., slowness or number of rounds) of BCryptPasswordEncoder can be specified by the developers as a parameter to the constructor. However we noticed that the default strength of BCryptPasswordEncoder is 10 (\(2^{10}\) number of rounds) [47]. However as mentioned previously [17] and confirmed by our own experiments presented in Appendix B, this default strength does not have enough slowness and essentially lack the security strength needed to prevent brute-force attack.

As developers tend not to pass any security strength parameter to the constructor assuming the default strength is secure enough, this tendency (as shown on listing 7) leaves an exploitable opportunity for the attackers.

Hence, we consider using the default insecure strength for BCryptPasswordEncoder as an insecure default configuration and recommend the developers to override the default strength by adjusting it according to their own system.

2) Using weak hash algorithm MD5 in remember-me cookie: Spring security provides remember-me which stored in the browser, allows the browser to remember the user for future sessions and causing automated login to take place. This remember-me cookie is constructed from the MD5 hashing of the username, expiration time of the cookie, and password and secret key as shown in listing 8.

Listing 8. Construction of remember-me cookie
```java
Listing 9. Example of adding content security policy headers
```
4) Absence of content security policy: Unexpectedly, unlike other protection mechanisms, Spring security does not add content security policy (CSP) HTTP headers by default. CSP helps the developers to enforce a fine-grained security policy easily to prevent code injection attacks e.g., cross site scripting, clickjacking, and data injection, etc. For example, if a developer perceives JavaScript from all external sources as untrusted then the developer needs to set the CSP header to self to prevent the browser from loading unsafe JavaScript from any untrusted external sources as shown on listing 9.

We have noticed a prominent trend among developers of not adding the CSP headers manually even though adding the headers CSP to prevent varieties of code injection attacks is important. Especially, because these code injection attacks are not trivial to prevent from security stand point. Developers might perceive that just like other protection mechanisms CSP headers are provided by default on in Spring security. Hence, we consider the case of not setting the CSP HTTP headers as an insecure default.

D. Severity of the misconfigurations

Each security misuses discussed in the previous section III-B, III-C has specific attack vectors presented in the literature. To prioritize them based on their security severity, we group them into three categories high, medium, and low. In this regard we consider two criteria i) attack difficulty, and ii) attacker’s gain as inspired from the Common Vulnerability Scoring System (CVSS) calculator [51]. The assignments of these severities to identified security misuses are highlighted in Table 1 and described as follows.

1) High: Anti-patterns causing CSRF and man-in-the-middle attacks are easy to construct and provide large gain for the attackers. Insecure default MD5 in remember me cookie and BCrypt with insecure param can be brute-force easily given the high number of available cracking tools. Attacks originating from exploiting secrets stored in insecure places are easy to construct as well.

2) Medium: Attackers can try to do offline brute force attack to guess the fixed secret key used to sign the JWT token which can be time consuming depending on the entropy/randomness level and length of the fixed secret key. Moreover lifelong access token can be reused/replayed by attackers if and only if it is leaked.

3) Low: We place the two insecure defaults which depend on the presence of another vulnerability for the attacker to take advantage. For example absence of content security policy can be leveraged if and only if there is Cross site vulnerability already present. In the same way lack of required throttling policy per API key can cause DoS/DDoS attack if attacker can bypass the network level protection to mitigate DoS/DDoS attack in the first place.

E. Case Study

Simplicity and performance over insecure practices. In one real-world application, we saw the following comments before one of the security anti-pattern.

“THIS IS NOT A SECURE PRACTICE! For simplicity, we are storing a static key here.”

In another case, a developer responded to us for a potential CSRF attack due to the absence of state parameter in redirect URL as following.

“Increasing the state parameter can effectively prevent CSRF attacks. But my demo is just a simple sso demonstration. The simplest way to demonstrate the entire sso interaction process does not need to consider CSRF attacks.”

In another project, we noticed that the developer intentionally downgraded the default strength 10 (2^10 rounds) of BCryptPasswordEncoder to 8 to increase the performance. This illustrates that developers more often prefer simplicity and performance over secure coding practices.

Storing application secrets in config file. We observed that in 3 applications and 8 demo applications, developers stored secrets in the application.yml. Especially, we observed 5 number of applications to store secrets to sign the JWT in application.yml files. It is recommended to store these secrets in the server’s key stores [12].

Separating development and production environments. Sometimes developers avoid configuring TLS in their development environment [45]. A similar configuration will cause a devastating effect on the production environment. Spring security enables separating two different environments by simply using two separate configuration files. However, we observed that in 13 projects, developers used the same configuration of insecure TLS in both development and production.

IV. DISCUSSION

Disclosure of our findings. We made an effort to share our concerns about these 3 major vulnerabilities found with the Spring security community. We created two pull requests and one issue on the master branch of Spring security project about i) replacing weak MD5 with a strong hashing algorithm SHA-256 ii) adding proper guidelines in Spring security official documentation about setting a secure strength for BCryptPasswordEncoder iii) possibility of performing DoS attack exploiting lack of required throttling policy per API key. The second request was already accepted and the community agreed with us on others. However, they expressed to remain passive for now but will consider bringing our suggested changes in the next Spring security major release. We also reported some of the severe cases to the application developers and in the process of disclosing others.

Limitations. The derived security anti-patterns are mainly based on manual inspection and therefore is subjected to human bias. To address this, the first two authors of the paper carefully and independently apply analysis multiple times to verify the security anti-patterns. We also acknowledge that the data-set constructed by popularity and adaption measure is
susceptible to subjectivity, as this filtering measure may incorrectly remove some relevant projects using spring security.

V. CONCLUSION

Without careful considerations, customizing application frameworks can cause critical vulnerabilities in an enterprise application. In this paper, we studied the application framework misconfiguration vulnerabilities in the light of Spring security. First, by analyzing 28 Spring security applications, we identified 6 security anti-patterns and 4 insecure default behaviors representing possible insecure use-cases of Spring security. Our analysis showed that the security anti-patterns are prevalent and similar across the real-world and demo applications, hence, pose a realistic threat.

REFERENCES


BCryptPasswordEncoder's strength
Mean time to verify single password (sec)

The Spring security reference guide mentions that the strength of BCryptPasswordEncoder should be tuned to take about 1 second to verify a password on the developer’s own system. However, according to our experiments the default strength of BCryptPasswordEncoder which is 10 (number of rounds $2^{10}$) takes around 0.1 seconds to verify a password. This is way less than 1 second lower limit. Since developers tend to use the default strength without any consideration assuming that the default implementation should have enough strength we consider this an insecure default configuration.

Appendix B

Finding vulnerabilities in 8 real world projects

We also test the 8 real-world projects using a know vulnerability scanning tool Snyk [26] and then parse the scanning results. Snyk maintains an online database of known vulnerabilities and can automatically build the project to check against these known vulnerabilities. A summary of the results is shown in Table IV.
# OF VULNERABILITY FOUND AND THEIR RISK LEVEL IN 8 REAL WORLD APPLICATIONS USING Snyk. RISK LEVEL HIGH, MEDIUM, AND LOW ARE DENOTED BY H, M, L RESPECTIVELY.

<table>
<thead>
<tr>
<th>No</th>
<th>Project name</th>
<th># forks</th>
<th># stars</th>
<th>LoC</th>
<th>Risk level</th>
<th># of vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>paascloud-master</td>
<td>3.6k</td>
<td>7.7k</td>
<td>55.2k</td>
<td>H</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>xboot</td>
<td>968</td>
<td>2.6k</td>
<td>21.2k</td>
<td>H</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Spring-boot-cloud</td>
<td>1.2k</td>
<td>2k</td>
<td>523</td>
<td>H</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>sso</td>
<td>323</td>
<td>702</td>
<td>3.3k</td>
<td>H</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>FEBS-cloud</td>
<td>326</td>
<td>660</td>
<td>11.9k</td>
<td>H</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>fw-cloud-framework</td>
<td>325</td>
<td>638</td>
<td>13.9k</td>
<td>H</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>cas</td>
<td>3.2k</td>
<td>7.5k</td>
<td>33.5k</td>
<td>No vulnerability found</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>microservices-platform</td>
<td>736</td>
<td>1.6k</td>
<td>25.4k</td>
<td>Can not run Snyk because of build errors</td>
<td></td>
</tr>
</tbody>
</table>