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.

978-1-7281-8388-6/20/$31.00 ©2020 IEEE
DOI 10.1109/SecDev45635.2020.00024

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 group.

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.

TABLE I
IDENTIFIED SECURITY MISUSES ARE PRESENTED WITH THEIR CORRESPONDING KNOWLEDGE-BASE REFERENCES, AFFECTING FEATURES, THREATS, SEVERITY AND COUNTS IN 28 GITHUB PROJECTS. HIGH, MEDIUM, AND LOW SEVERITY LEVELS ARE DENOTED BY H/M/L RESPECTIVELY.

Type	Rule	Reference	Feature	Threat	Severity	Count (28)
Anti-patterns	(1) Using lifelong valid access tokens	[22], [25], [29]	Authentication	Secrets leaking	M	7
	(2) Absence of state param in redirect URL	[29]	Authentication	CSRF attacks	H	11
	(3) Using fixed secret to sign JWT tokens	[22], [28], [34]	Authentication	Brute-force	M	14
	(4) Storing secrets in insecure places	[26]	Authentication	Secrets leaking	H	17
	(5) Disabling CSRF protection	[26], [27]	Exploit protection	CSRF attacks	H	17
	(6) Not using TLS for HTTP communication	[22], [26], [28]	Exploit protection	Man-in-the-middle	H	15
Insecure defaults	(7) Using Bcrypt with insecure params	[24]	Authentication	Brute-force	H	11
	(8) Using MD5 in remember me cookie	[23], [28]	Authentication	Brute-force	H	N/A
	(9) Lack of req. throttling policy per API key	[22], [33]	Exploit protection	DoS attacks	L	N/A
	(10) Absence of content security policy (CSP)	[26]	Exploit protection	Code injection	L	N/A

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 II
SECURITY MISUSE COUNTS FOR 8 REAL-WORLD AND 20 DEMO PROJECTS.

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

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.

Listing 1. Setting lifelong valid access token

```
app:
   auth:
      tokenExpirationMsec: 864000000
      // 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 lasted 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.

1.https://graph.facebook.com/oauth/?grant_type="authorization_code"
&code=users_auth_code&client_id="aiqyi"&client_secret="secret"

2. https://graph.facebook.com/oauth/?grant_type="authorization_code"
&code=attacker's_code&client_id="aiqyi"&client_secret="secret"

3. https://graph.facebook.com/oauth/?grant_type="authorization_code"
&code=users_auth_code&client_id="aiqyi"&client_secret="secret"
&state = random_value

Fig. 1. The 2nd redirect URL constructed from 1st is vulnerable to forced login CSRF attack. The 3rd redirect URL is not vulnerable due to the non guessable state parameter.

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.

Listing 2. Adding `state` param in `redirect_URL`

```
public String getToken(@RequestParam String code){
      ...
      params.add("grant_type","authorization_code");
      params.add("code",code);
      params.add("client_id","aiqiyi");
      params.add("client_secret","secret");
✓     params.add("state",127621437303857);
      // Randomly generated value of state param
      ...
}
```

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.

```
public class TokenProvider {
   public String createToken(Authentication auth) {
     Public String JWT_SIGN_KEY = "123456";
     token = Jwts.builder()
           ...
🐞         .signWith(SignatureAlgorithm.HS512, JWT_SIGN_KEY)
           // signing with a hard coded fixed secret
           .compact();
           ...
      return token;
   }
}
```

This insecure practice of using a fixed secret to sign all JWT can lead to a great inconvenience frequently when any one of the valid JWT gets leaked. Because then invalidating the leaked JWT would require the developer to change the fixed secret manually. This would automatically invalidate all other valid JWT as well which have not been leaked. Perhaps a quick way around instead of changing the fixed secret would be to keep a blacklist for leaked JWT in the database. However keeping, maintaining, and querying to the database if the JWT is blacklisted for each request can be expensive and most importantly introduces scalability problems, which is one of the very basic problems why JWT was introduced and is popular among developers.

RFC-8725 [35] strongly discourages this insecure coding practice and advises the developers to use random cryptographic keys with sufficient entropy. To solve this problem, we suggest to leverage the optionally available JWT key identifier parameter `kid` which can be leveraged for managing multiple randomly generated secret keys [34].

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.

Listing 3. Stroing secrets insecurely in `application.yml`

```
spring:
   datasource:
      username: root
🐞       password: uOtmALFgsfxgYzEgluLXl3O
```

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.

Listing 4. Retrieving secrets from Spring vault

```
    @Value("${clientPassword}")
    String client_password;
```

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.

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 workaround 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.

Listing 6. Proper way to circumvent CSRF misconfiguration errors

```
@Override
protected void configure(HttpSecurity hs) throws Exception {
      hs.csrf(c -> c
         .csrfTokenRepository(CookieCsrfTokenRepository
✓                .withHttpOnlyFalse())
            // Marking the CSRF Token's HttpOnly to False.
 )
}
```

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].

```
  eureka:
     client:
     serviceUrl:
        defaultZone: http:root@paascloud-eureka:8761
        // use of HTTP without TLS
```

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.

Listing 7. Default strength is vulnerable to brute-force attack

```
  @Bean
  public PasswordEncoder passwordEncoder() {
     return new BCryptPasswordEncoder();
     /* using default strength 10 is vulnerable
     to feasible brute-force attacks */
  }
```

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

```
  base64(username + ":" + expirationTime + ":" +
     md5Hex(username + ":" + expirationTime +
     // Use of weak hashing algorithm MD5
        ":" password + ":" + key))
```

The problem with this approach is that MD5 is considered broken, which susceptible to collision attacks [48] and modular differential attacks [49]. Hence, attackers can easily recover sensitive information or temper the `remember-me` cookie. Therefore, we suggest the Spring security maintainers to fix this issue by replacing MD5 with a secure hashing algorithm (e.g., SHA-256).

3) Lack of required throttling policy per API key: One of the most important parts of resource management policy for web API is to set a proper throttling policy per user. This throttling policy places a limit on the number of requests a user can make with a secret API key. Otherwise, an attacker can use use a valid secret API key to generate a massive number of requests than the web service can handle. In this way, the attacker will be able to make a denial of service attack for other users. However, unlike other security framework (e.g., Django [50]), Spring security lacks this as a built-in feature given the prevalence of this DoS attack with custom made bots. An IP-based throttling policy can prevent DoS attack but then attackers can switch to DDoS attack to abuse valid API keys. We suggest a manual solution where the developers build custom filter and register it in the Spring context.

Listing 9. Example of adding content security policy headers

```
@Override
protected void configure(HttpSecurity hs) {
   hs
      ...
      .headers(headers -> headers
      .contentSecurityPolicy(csp -> csp
```

```
7             .policyDirectives("script-src 'self'"))
8     ...
9 }
```

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 tend 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 I 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

[1] D. C. Schmidt and F. Buschmann, "Patterns, frameworks, and middleware: Their synergistic relationships," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pp. 694–704, 2003.

[2] "2020 java technology report." https://www.jrebel.com/blog/2020-java-technology-report. [Online; accessed 30-April-2020].

[3] "Developer survey results 2019: Web frameworks." https://insights.stackoverflow.com/survey/2019#technology-_-web-frameworks. [Online; accessed 30-April-2020].

[4] "Spring Security." https://spring.io/projects/spring-security. [Online; accessed 28-April-2020].

[5] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure Coding Practices in Java: Challenges and Vulnerabilities," in *ACM ICSE'18*, (Gothenburg, Sweden), May 2018.

[6] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *ACM CCS'12*, 2012.

[7] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, "Why Eve and Mallory love Android: an analysis of Android SSL (in)Security," in *ACM CCS'12*, pp. 50–61, 2012.

[8] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 73–84, 2013.

[9] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security," in *IEEE S&P'17*, pp. 121–136, 2017.

[10] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," in *ECOOP'18*, pp. 10:1–10:27, 2018.

[11] R. Paletov, P. Tsankov, V. Raychev, and M. T. Vechev, "Inferring crypto API rules from code changes," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, pp. 450–464, 2018.

[12] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2455–2472, 2019.

[13] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 536–547, 2019.

[14] Y. Acar, M. Backes, S. Fahl, S. L. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pp. 154–171, 2017.

[15] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl, "Developers need support, too: A survey of security advice for software developers," in *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pp. 22–26, 2017.

[16] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S. P. H. Chung, and W. Lee, "Broken fingers: On the usage of the fingerprint API in android," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[17] "Recommended # of rounds for bcrypt." https://security.stackexchange.com/questions/17207/recommended-of-rounds-for-bcrypt. Last accessed: 2020-04-01.

[18] A. Dikanski, R. Steinegger, and S. Abeck, "Identification and implementation of authentication and authorization patterns in the spring security framework," in *The Sixth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2012)*, pp. 14–30, 2012.

[19] A. Armando, R. Carbone, E. G. Chekole, and S. Ranise, "Attribute based access control for apis in spring security," in *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, SACMAT '14, (New York, NY, USA), p. 85–88, Association for Computing Machinery, 2014.

[20] "Authentication Mechanisms." https://docs.spring.io/spring-security/site/docs/current/reference/html5/#servlet-authentication. Last accessed: 2020-04-01.

[21] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.

[22] "MITRE." https://cwe.mitre.org/index.html. Last accessed: 2020-04-01.

[23] "Openstack Security Impact Check." https://wiki.openstack.org/wiki/Security/OpenStack_Security_Impact_Checks#Testing_for_Security_Anti-patterns. [Online; accessed 29-April-2020].

[24] "Spring Security Reference." https://docs.spring.io/spring-security/site/docs/5.3.0.RELEASE/reference/html5. [Online; accessed 29-April-2020].

[25] "Introduction to antipatterns." https://docs.apigee.com/api-platform/antipatterns/intro. Last accessed: 2020-04-01.

[26] "Vulnerability DB — Snyk." https://snyk.io/vuln. Last accessed: 2020-04-01.

[27] T. Nafees, N. Coull, I. Ferguson, and A. Sampson, "Vulnerability anti-patterns: A timeless way to capture poor software practices (vulnerabilities)," in *Proceedings of the 24th Conference on Pattern Languages of Programs*, PLoP '17, (USA), The Hillside Group, 2017.

[28] A. A. U. Rahman *et al.*, "Anti-patterns in infrastructure as code.," 2019.

[29] P. Siriwardena, *Advanced API Security*. Springer, 2014.

[30] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?," *Empirical Softw. Engg.*, vol. 23, p. 384–417, Feb. 2018.

[31] M. del Pilar Salas-Zárate, G. Alor-Hernández, R. Valencia-García, L. Rodríguez-Mazahua, A. Rodríguez-González, and J. L. L. Cuadrado, "Analyzing best practices on web development frameworks: The lift approach," *Science of Computer Programming*, vol. 102, pp. 1–19, 2015.

[32] Internet Engineering Task Force, "The OAuth 2.0 authorization framework." https://tools.ietf.org/html/rfc6749. Last accessed: 2020-04-01.

[33] Internet Engineering Task Force (IETF), "OAuth 2.0 threat model and security considerations." https://tools.ietf.org/html/rfc6819. Last accessed: 2020-04-01.

[34] "Json Web Signature (JWS)." https://tools.ietf.org/html/rfc7515. Last accessed: 2020-04-01.

[35] "Json Web Token Best Current Practices." https://tools.ietf.org/html/rfc8725. Last accessed: 2020-04-01.

[36] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[37] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang, "Why and how developers fork what from whom in github," *Empirical Software Engineering*, vol. 22, no. 1, pp. 547–578, 2017.

[38] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 334–344, IEEE, 2016.

[39] "IETF OAuth working group." https://oauth.net/2. Last accessed: 2020-04-01.

[40] "What's New in Spring Security 5.0." https://docs.spring.io/spring-security/site/docs/5.0.x/reference/htmlsingle/#new-features. Last accessed: 2020-04-01.

[41] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 75–88, 2008.

[42] "Internet engineering task force (IETF), oauth 2.0 threat model and security considerations." https://tools.ietf.org/html/rfc7519. Last accessed: 2020-04-01.
[43] "Vault by HashiCorp." https://www.vaultproject.io. Last accessed: 2020-04-01.
[44] "java - simple example of spring security with thymeleaf." https://stackoverflow.com/questions/25692735/simple-example-of-spring-security-with-thymeleaf. Last accessed: 2020-04-01.
[45] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 164–175, IEEE, 2019.
[46] N. Provos and D. Mazières, "A future-adaptive password scheme," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, (USA), p. 32, USENIX Association, 1999.
[47] "Bcryptpasswordencoder (spring-security-docs-manual 5.3.2.RELEASE API)." https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html. Last accessed: 2020-04-01.
[48] B. Den Boer and A. Bosselaers, "Collisions for the compression function of md5," in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 293–304, Springer, 1993.
[49] X. Wang and H. Yu, "How to break md5 and other hash functions," in *Annual international conference on the theory and applications of cryptographic techniques*, pp. 19–35, Springer, 2005.
[50] "Throttling- Django REST framework." https://www.django-rest-framework.org/api-guide/throttling/. Last accessed: 2020-04-01.
[51] "Common Vulnerability Scoring System." https://www.first.org/cvss/calculator/3.0. Last accessed: 2020-04-01.

APPENDIX A
IMPLEMENTATIONS OF PASSWORD HASHING

Spring security provides an array of `PasswordEncoder` implementation for storing password. We provide here a summary of these implementations. However among them `Md4Password`, `MessageDigest`, `Standard`, and `LdapSha` password encoders use digest based password encoding which is not considered secure. As a result they are deprecated to indicate that they are legacy implementation. `Argon2Password` and `SCryptPassword` password encoder uses Bouncy castle. One problem associated with this is that Bouncy castle does not exploit parallelism/optimizations that other password crackers will. Therefore there is an uneven asymmetry between attacker and defender. The most recommended way to implement the `PassworEncoder` interface is `BCryptPasswordEncoder`. Table III summarizes the 9 option for password encoding offered by Spring security.

TABLE III
SUMMARY OF PASSWORD ENCODING INTERFACE IMPLEMENTATIONS

Implementation Name	Comment
`BCryptPasswordEncoder`	preferred implementation
`NoOpPasswordEncoder`	stores password in plain-text

`Md4PasswordEncoder`
`MessageDigestPasswordEncoder`
`StandardPasswordEncoder`
`LdapShaPasswordEncoder`
	digest based password encoding

`Argon2PasswordEncoder` `SCryptPasswordEncoder`	uses Bouncy castle
`Pbkdf2PasswordEncoder`	uses PBKDF2

APPENDIX B
INSECURE DEFAULT STRENGTH FOR BCRYPTPASSWORD ENCODER

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.

On our system (LENOVO Ideapad- Intel(R) Core(TM) i5 CPU @ 1.60GHz - 7.4 GiB RAM - running Ubuntu 20.04 64 bit), we found out that appropriate secure strength should be 14 as shown on Figure 2. We emphasize that for GPU which can perform orders of magnitude faster than a typical CPU like the one we have used, the appropriate secure strength to prevent feasible brute-force attack should be higher than 14.

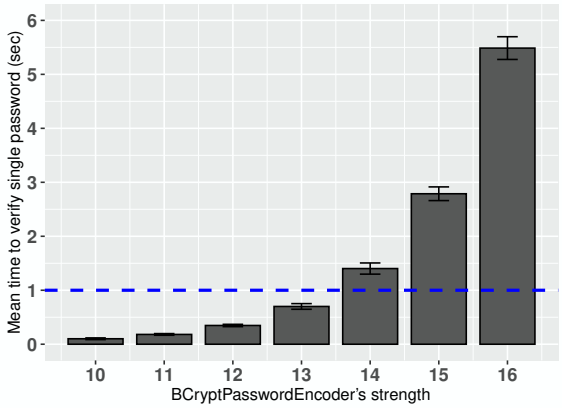

Fig. 2. Mean time to verify a password of size 128 bits on our own system for different strength. Ideally the strength should add enough slowness so that it takes at least 1 sec to verify a password as marked by the horizontal dashed line.

APPENDIX C
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 vulnerability and can automatically build the project to check against these known vulnerabilities. A summary of the results is shown in Table IV.

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.

No	Project name	# forks	# stars	LoC	Risk level	# of vulnerabilities
1	paascloud-master	3.6k	7.7k	55.8k	H	68
					M	18
					L	7
2	xboot	968	2.6k	21.2k	H	3
					M	3
					L	0
3	Spring-boot-cloud	1.2k	2k	523	H	71
					M	18
					L	7
4	sso	323	702	3.3k	H	55
					M	10
					L	5
5	FEBS-cloud	326	660	11.9k	H	5
					M	6
					L	1
6	fw-cloud-framework	325	638	13.9k	H	63
					M	6
					L	3
7	cas	3.2k	7.5k	33.5k	No vulnerability found	
8	microservices-platform	736	1.6k	25.4k	Can not run Snyk because of build errors	