Coding Practices and Recommendations of Spring Security for Enterprise Applications

Mazharul Islam, Sazzadur Rahaman, Na Meng, Behnaz Hassanshahi, Padmanabhan Krishnan, Danfeng (Daphne) Yao
Spring Security in Java Enterprise Applications

How to secure these Java Enterprise applications?

source: https://www.jrebel.com/blog/2020-java-technology-report
Spring Security in Java Enterprise Applications (contd)

- Default protection against common exploits
- Reusable authentication and authorization modules
- Enable seamless integration

“Can improperly configuring Spring security, make the enterprise applications insecure/vulnerable to attacks?”

Enterprise applications
Related work
Source of Vulnerable code

- Library API misuse showed the dangers of misusing application-level SSL/TLS and cryptographic APIs [Georgiev et al ACM CCS ‘12], Egele et al ACM SIGSAC ‘13]
- The role of Stack-Overflow’s misleading advices [Meng et al ICSE’18]
- Poor API designs [Acar et al IEEE S&P 2017]
- Lack of proper guidelines [Acar et al SecDev ‘17]

Security framework misconfigurations can cause insecure/vulnerable code
Research Questions

What are the common security anti-patterns/insecure defaults in enterprise Spring security applications?
Our Contributions

- We performed a measurement based study on 28 Spring-based applications on GitHub.
- We have found
  - 6 types of security anti-patterns
  - 4 types of insecure defaults.
- The frequency of these security anti-patterns/insecure defaults.
- The severity of these security anti-patterns/insecure defaults
Methodology

Filters:
- Forks
- Stars
- Originality

28 Projects

20 Demo Projects

8 Real world Projects

Seven sources

A comprehensive list of insecure configurations
Six Security Anti-patterns
### Six Security Anti-patterns (1)

1. **Life long valid access tokens**  
2. Absence of state param in OAuth 2.0  
3. Fixed secrets to sign JWT tokens.  
4. Disabling CSRF protection  
5. Storing secrets in insecure places.  
6. Not using TLS.

**Listing 1. Setting lifelong valid access token**

```yaml
app:
auth:
  tokenExpirationMsec: 864000000
  // setting unnecessary long lifetime of 10 days
```

Lifelong valid access token is vulnerable to:  
- reply attacks  
- leaks.
Six Security Anti-patterns (2)

1. Life long valid access tokens
2. Absence of state param in OAuth 2.0
3. fixed secrets to sign JWT tokens.
4. Disabling CSRF protection
5. Storing secrets in insecure places.
6. Not using TLS.

Listing 2. Adding state param in redirect_URL

```java
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
    ...
}
```

State-params stops against CSRF attacks
Six Security Anti-patterns (3)

1. Life long valid access tokens
2. Absence of state param in OAuth 2.0
3. Fixed secrets to sign JWT tokens.
4. Disabling CSRF protection
5. Storing secrets in insecure places.
6. Not using TLS.

```
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;
    }
}
```

Hardcoded JWT keys is vulnerable to:
- brute force attacks.
- Leaks
Six Security Anti-patterns (4)

1. Life long valid access tokens
2. Absence of state param in OAuth 2.0
3. fixed secrets to sign JWT tokens.
4. Disabling CSRF protection
5. Storing secrets in insecure places.
6. Not using TLS.

Listing 5. Manually disabling default CSRF protection

```java
@override
protected void configure(HttpSecurity hs) throws Exception {
    hs.csrf().disable();
}
```

Vulnerable to CSRF attacks
Six Security Anti-patterns (5)

1. Life long valid access tokens
2. Absence of state param in OAuth 2.0
3. fixed secrets to sign JWT tokens.
4. Disabling CSRF protection
5. Storing secrets in insecure places.
6. Not using TLS.

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

```yaml
spring:
datasource:
username: root
password: u0tmALFgsfxgYzEgluLX13O
```

Storing secrets insecurely in configuration file.
Six Security Anti-patterns (6)

1. Life long valid access tokens
2. Absence of state param in OAuth 2.0
3. fixed secrets to sign JWT tokens.
4. Disabling CSRF protection
5. Storing secrets in insecure places.
6. Not using TLS.

```
eureka:
  client:
  serviceUrl:

defaultZone: http://root@paascloud-eureka:8761

// use of HTTP without TLS
```

HTTP is vulnerable to
- Man-in-the-middle attack
Four insecure defaults
Four Insecure Defaults (1)

1. Using BCrypt with insecure strength.
2. Using weak hash algo MD5
3. Lack of required throttling policy
4. Absence of CSP

The minimum time bcrypt should take to verify one hash value.
Four Insecure Defaults (2)

1. BCrypt
2. Using weak hash algo MD5
3. Lack of required throttling policy
4. Absence of CSP

MD5 is already a broken hashing algorithm. Vulnerable to
- Collision attacks
- Modular differential attacks.

Listing 8. Construction of remember-me cookie

```plaintext
base64(username + "":" + expirationTime + ":" + md5Hex(username + ":" + expirationTime + 
   // Use of weak hashing algorithm MD5
   ":" password + ":" + key))
```
## Four Insecure Defaults (3)

<table>
<thead>
<tr>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. BCrypt</td>
<td>Resource management policy for web API need to have a proper throttling policy per user to prevent DoS/DDoS attack.</td>
</tr>
<tr>
<td>2. Using weak hash algo MD5</td>
<td></td>
</tr>
<tr>
<td>3. Lack of required throttling policy</td>
<td></td>
</tr>
<tr>
<td>4. Absence of CSP</td>
<td></td>
</tr>
</tbody>
</table>

- Spring security framework lacks throttling policy.
Four Insecure Defaults (4)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BCrypt</td>
</tr>
<tr>
<td>2.</td>
<td>Using weak hash algo MD5</td>
</tr>
<tr>
<td>3.</td>
<td>Lack of required throttling policy</td>
</tr>
<tr>
<td>4.</td>
<td>Absence of CSP</td>
</tr>
</tbody>
</table>

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.

Spring security does not add content security policy (CSP) HTTP headers by default.
Severity of the misconfigurations
### Severity of the misconfigurations

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.

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

High: 6, Medium: 2, Low: 2
### Security Misuse Counts for 8 Real-World and 20 Demo Cases

<table>
<thead>
<tr>
<th>Anti-patterns</th>
<th>Real-world cases</th>
<th>Demo cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) lifelong valid access</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>(2) Absence of access control</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(3) Fixed access to sensitive data</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>(4) Inadequate access controls</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>(5) Privilege escalation</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>(6) Normal access</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

The nature of the misuse cases are vastly overlapped.
Case studies
Case studies (contd)

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

BCryptPasswordEncoder strength downgraded to from default 10 to 8.
Case studies (contd)

Security issue: `state` parameter missing in redirect URL #17

islamzehar opened this issue on May 17 - 2 comments

islamzehar commented on May 17 • edited

zlt2000 commented on May 18

Thank you for your suggestion! 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.

I want to know your view on this security concern and how it can affect the security of my application against CSRF attack as mentioned in the RFC 6749 document? Thanks in advance.

References:
[1] RFC 6749 The OAuth 2.0 Authorization Framework Cross Site Request Forgery
Case studies (contd)

Update BCryptPasswordEncoder documentation with default strength #8542

Islamazhar commented on May 17

Hi,

As mentioned in the Spring security doc on `bcryptpasswordencoder`

rwinch commented on May 21

Thanks for the Pull Request! This is now merged into master 😊

Spring-issuermaster added the `status: waiting for stage` label on May 17

Islamazhar commented on May 18

I searched a bit more and found some interesting links which may be useful for adding a more appropriate context to the issue.

- The python `bcrypt` library mentions about the default strength here in `Adjustable Work Factor` section.
- The NodeJS `bcrypt` documentation mentions about the strength factor here.
- An interesting answer on security stack exchange about the number of recommended rounds for `bcrypt`.
Case studies (contd)

Replacing MD5 hashing for remember me token #8549

Islamazhar opened this issue on May 18 · 4 comments

Rwinch requested changes on May 21

Rwinch left a comment

Thanks for the PR. We need to remain passive, so we would want to allow configuring the algorithm vs changing it. We could consider changing it by default in Spring Security 6 though.

An alternative would be to do something like we did with passwords and have a prefix that specifies the algorithm that is being used. If no algorithm is specified it would default to MD5.


Key Takeaways...

● Misconfigurations in Spring security framework can cause critical vulnerabilities in an enterprise application.
● We identified 6 security anti-patterns and 4 insecure default behaviors
● Our analysis showed that the security anti-patterns are
  ○ prevalent
  ○ similar across the real-world and demo applications
● These misconfigurations hence pose a realistic threat.
Thanks!