IEEE SecDev 2020

IEEE COMPUTER SOCIETY

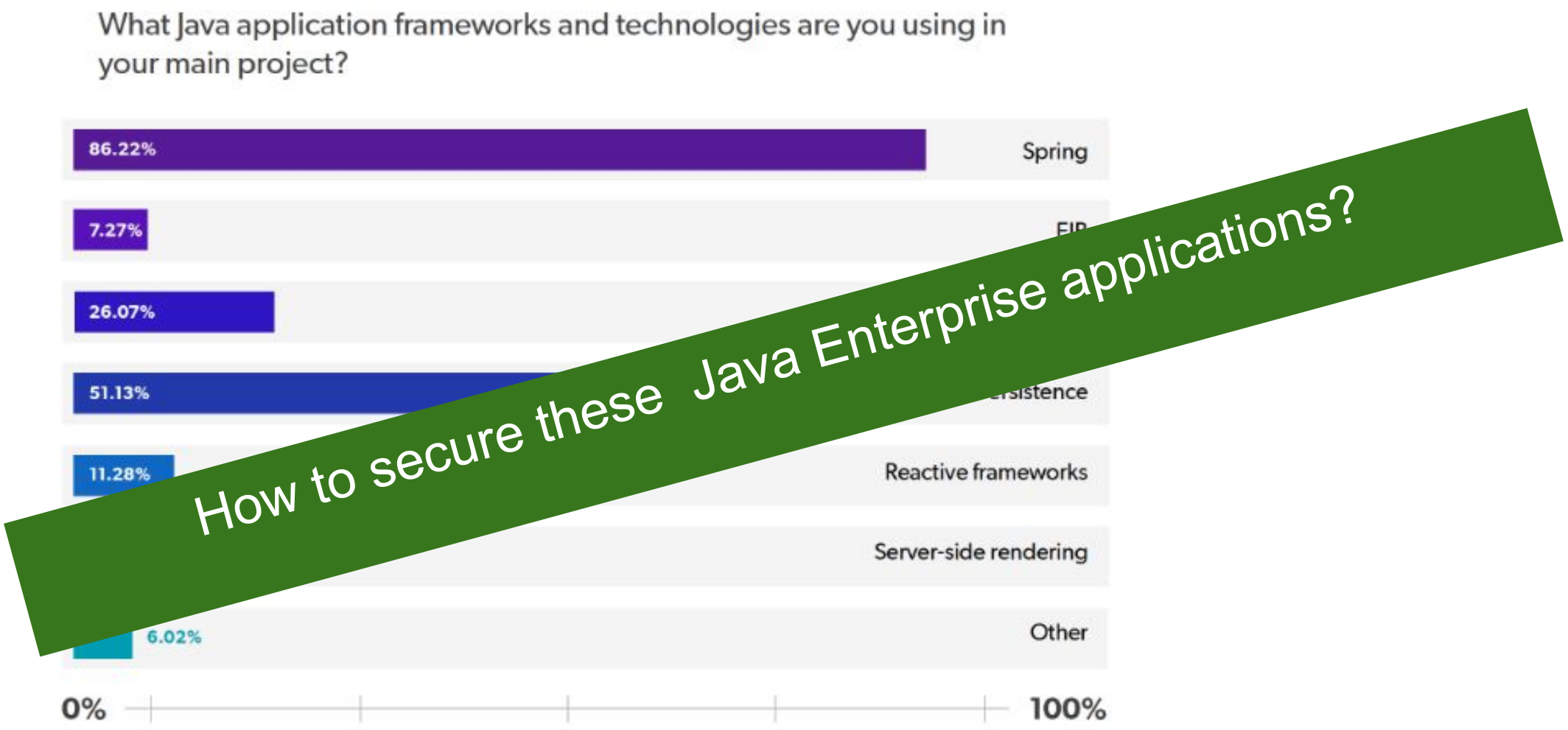
Coding Practices and Recommendations of Spring Security for Enterprise Applications

Mazharul Islam, Sazzadur Rahaman, Na Meng, Behnaz Hassanshahi, Padmanabhan Krishnan, Danfeng (Daphne) Yao

ORACLE LABS

#IEEESecDev https://secdev.ieee.org/2020

Spring Security in Java Enterprise Applications



source: https://www.jrebel.com/blog/2020-java-technology-report

Spring Security in Java Enterprise Applications (contd)

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

- Default protection agai[illegible]ommon
- [illegible]ntication and authorization modules
- Enable seamless integration

Related work

Source of Vulnerable code

- Library API misuse showed the dangers of mis
  SSL/TLS and cryptographic APIs [Ge
  al ACM SIGSAC '13]
- The role of Stack-O
- Poor API de
- L

Security framework misconfigurations can cause insecure/vulnerable code

), Egele et

[Meng et al ICSE'18']

2017]

et al SecDev '17]

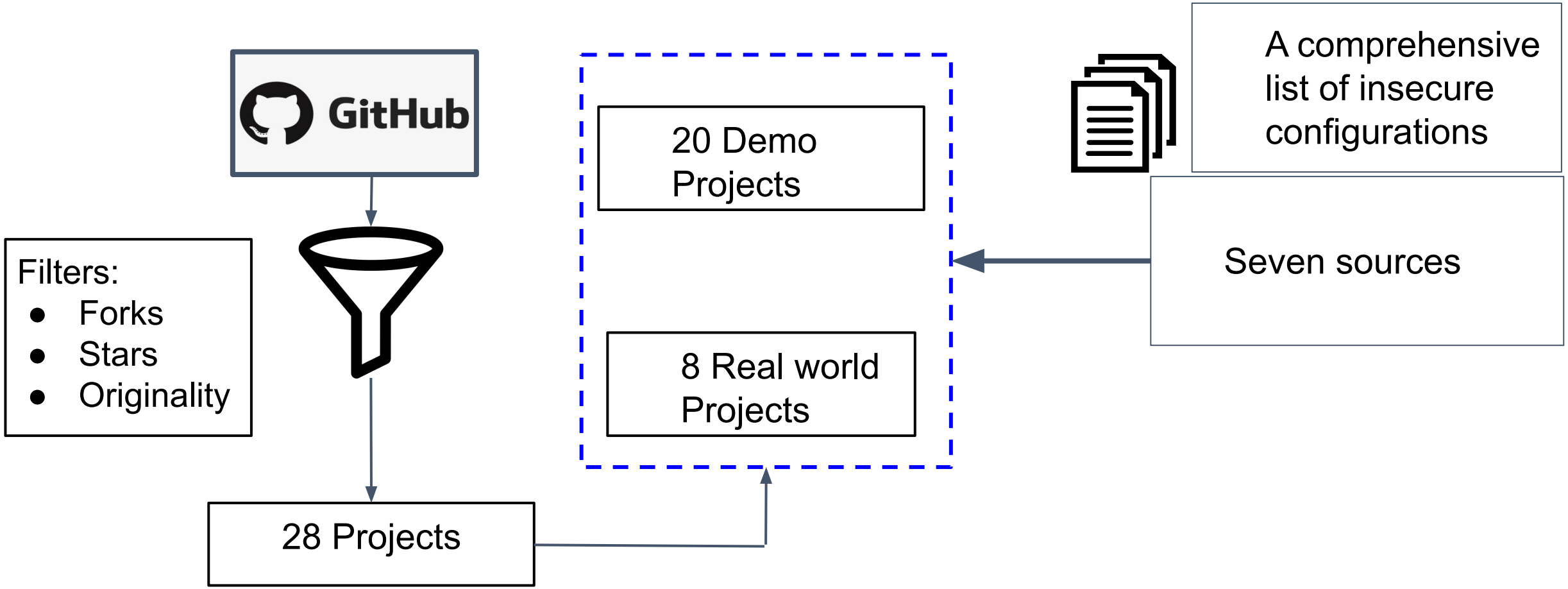
Research Questions

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



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

```
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

```
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

```
@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. Stroing secrets insecurely in `application.yml`

```
spring:
datasource:
username: root
        password: uOtmALFgsfxgYzEg1uLXl3O
```

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.

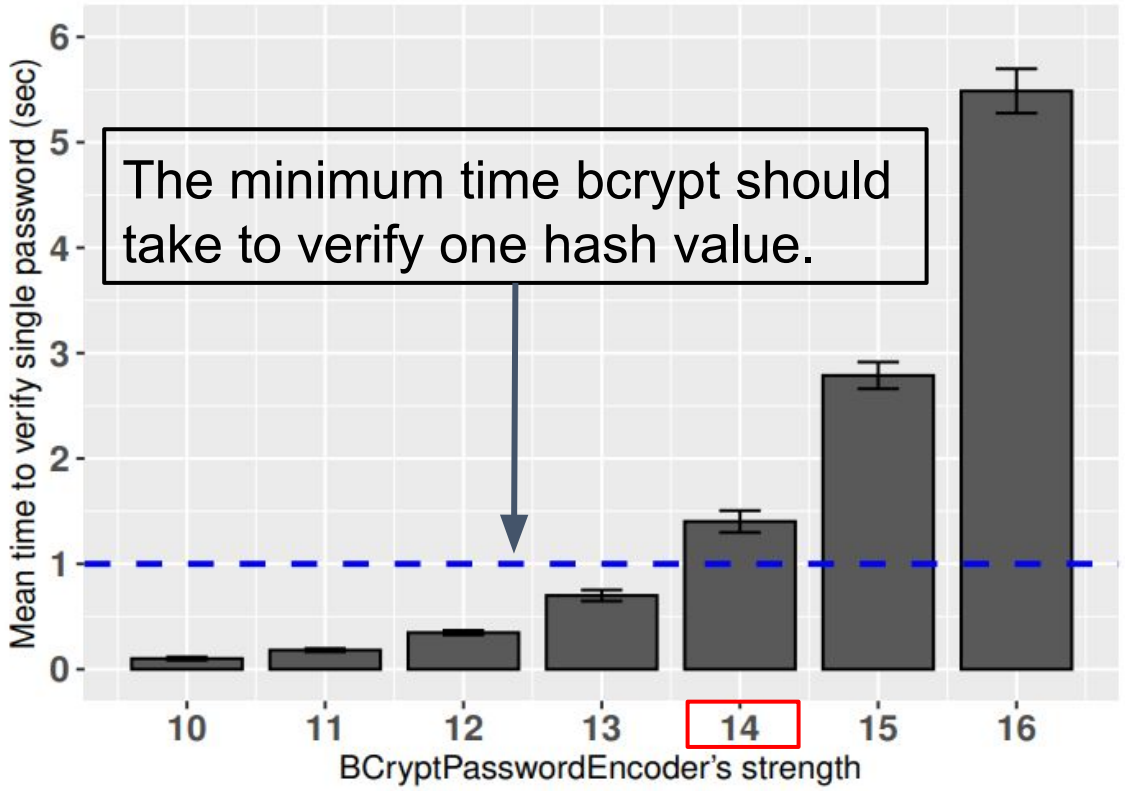
```
eureka:
client:
serviceUrl:
defaultZone: http:root@paascloud-eureka:8761
// use of HTTP without TLS
```

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



Four Insecure Defaults (2)

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

Listing 8. Construction of remember-me cookie

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

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

Four Insecure Defaults (3)

1. BCrypt 2. Using weak hash algo MD5 3. Lack of required throttling policy 4. Absence of CSP	Resource management policy for web API need to have a proper throttling policy per user to prevent DoS/DDoS attack. Spring security framework lacks throttling policy.

Four Insecure Defaults (4)

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

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

High: 6, Medium: 2. Low:2

Characterizing and Scoring Vulnerabilities

Misconfigurations Count

SECURITY MISUSE COUNTS FOR 8 REAL-WORLD AND 20 DEM S.

Anti-patterns	Res		ses
(1) lifelong valid access			0
(2) Absence of		9	2
(3) Fixed		8	4
(4)	5	12	5
(5)	6	11	6
(6) N	4	11	4

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

Closed **islamazhar** opened this issue on May 17 · 2 comments

islamazhar commented on May 17 • edited

zlt2000 commented on May 18 Owner

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

Merged **rwinch** merged 1 commit into `spring-projects:master` from `islamazhar:BcryptPassword-Encoder-default-cost` on May 21

Conversation 2 | Commits 1 | Checks 0 | Files changed 1

islamazhar commented on May 17 • edited — Contributor

Hi,

As mentioned in the Spring security doc on `BCryptPasswordEncoder`

rwinch commented on May 21 — Member

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

👍 1

spring-issuemaster added the status: waiting-for-triage label on May 17

islamazhar commented on May 18 — Author, Contributor

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`.

Reviewers

No reviews

Assignees

Linked issues

Successfully mer
these issues.

None yet

Notifications

You're receiving
the thread.

Case studies (contd)

Replacing MD5 hashing for remember me token #8549

Open **islamazhar** opened this issue on May 18 · 4 comments

rwinch requested changes on May 21

View change

rwinch left a comment

Member

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

CryptographicTechniques, pp. 293–304, Springer, 1993

[2] Wang, Xiaoyun, and Hongbo Yu. "How to break MD5 and other hash functions." Annual international conference on the theory and applications of cryptographic techniques. Springer, Berlin, Heidelberg, 2005.

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!