# Fixing the Fixes: Assessing the Solutions of SAST Tools for Securing Password Storage[⋆]

Harshal Tupsamudre, Monika Sahu, Kumar Vidhani, and Sachin Lodha

TCS Research, Tata Consultancy Services, India
`firstname.lastname@tcs.com, monika.sahu1@tcs.com`

**Abstract.** Text passwords are one of the most widely used authentication mechanisms on the internet. While users are responsible for creating secure passwords, application developers are responsible for writing code to store passwords securely. Despite continued reports of password database breaches, recent research studies reveal that developers continue to employ insecure password storage practices and have several misconceptions regarding secure password storage. Therefore, it is important to detect security issues relating to password storage and fix them in a timely manner before the application is deployed.
In this paper, we survey several open-source (SpotBugs, SonarQube, CryptoGuard, CogniCrypt) Static Application Security Testing (SAST) tools to understand their detection capabilities with respect to password storage vulnerabilities and determine if the remediation fixes suggested by these tools are consistent with the OWASP or NIST recommended password storage guidelines. We found that none of the surveyed tools covers all potential vulnerabilities related to password storage. Further, we found that solutions suggested by the tools are either imprecise or they are not in accordance with the latest password storage guidelines. We conduct a study with 8 developers where each of them attempted to replace insecure SHA-1 based password storage implementation with PBKDF2 solution recommended by the surveyed tools. The study results show that, in the absence of specific examples, developers choose insecure values for PBKDF2 parameters (salt, iteration count, key length). Thus, although the use of PBKDF2 is in adherence with the tool requirements, the resulting password storage code may not be secure in practice.

**Keywords:** Secure Password Storage · Security Testing Tools.

## 1 Introduction

Text based passwords are the most common way of authenticating users on the internet. Plenty of research studies have been conducted to investigate the

[⋆] Harshal Tupsamudre, Monika Sahu, Kumar Vidhani, Sachin Lodha, Fixing the Fixes: Assessing the Solutions of SAST Tools for Securing Password Storage, Proceedings of AsiaUSEC'20, Financial Cryptography and Data Security 2019 (FC). February 14, 2020 Kota Kinabalu, Sabah, Malaysia Springer, 2020 password creation and password management strategies (storage, reuse etc.) of end-users [29][22][39][43][33]. Recently, some efforts have been made to understand the steps taken by developers to protect users' passwords on the server [18][31][32][30]. The use of a weak password could jeopardize the security of the user, but a weaker server-side password implementation could put the security of all application users at risk (including the ones who have put great efforts to create a stronger password). In two recent studies, one involving GitHub developers [18] and the other involving freelance developers [30], researchers found that many developers do not store passwords securely unless prompted to do so. Further, most of the developers who attempt to store passwords securely, use either insecure methods (e.g., base64 encoding, encryption or hashing without using a proper salt) or outdated methods (e.g., MD5 or SHA-1). These results are not surprising, since various password database breaches [12] reveal that even developers from reputed companies are guilty of adopting insecure password storage practices. For instance, a data breach at LinkedIn in 2012 revealed that user passwords were stored using insecure SHA-1 hash function and without salt [27]. A data breach at Adobe in 2013 revealed that user passwords were encrypted instead of hashed [16]. These insecure coding practices by developers are often attributed to usability issues within existing cryptography APIs (e.g., poor documentation and insufficient code examples) [26][17][45], and to their lack of expertise in the security related concepts and technologies [31][32][30].

Several guidelines are available on how to store users' passwords securely. OWASP recommends the use of bcrypt hash function, a unique 16 character long random salt for each password and a common 32 character long random pepper for all passwords [11]. NIST recommends PBKDF2 for password hashing to be used with HMAC-SHA-256 and a work factor of at least 10,000 iterations [25]. Salting provides protection against an attacker pre-computing hashes using rainbow tables. However, for salting to work properly, it should be generated using cryptographically secure pseudo-random number generator (PRNG) [11]. Java provides two PRNGs *java.util.Random* and *java.security.SecureRandom*, of which the latter is cryptographically secure [4]. Encryption is highly discouraged since an attacker who gains access to the decryption key can recover plaintext passwords easily. Additionally, passwords protected with simple hash algorithms such as MD5 and SHA-1 are vulnerable to GPU-based cracking [38].

Security issues related to password storage are so common that they are assigned unique IDs and placed in Common Weakness Enumeration (CWE) which is a community-developed formal list of software weaknesses [3]. This list is intended to serve as a common language for describing software security weaknesses and is referenced by software security tools targeting these vulnerabilities. Further, some of these issues appear consistently in OWASP top 10 critical web application security risks [10]. The description of vulnerabilities pertaining to insecure password storage along with CWE-ID numbers and examples are shown in Table 1. The first part of the table enumerates issues specific to hashing and the second part enumerates issues specific to salting. For ease of reference, we also associate a mnemonic with each CWE-ID.

| CWE | Description | Example | Mnemonic |
|---|---|---|---|
| CWE-327 | Use of a Broken or Risky Cryptographic Algorithm | MD5, SHA-1 | Weak Hash |
| CWE-256 | Unprotected Storage of Credentials | Plaintext | Plaintext |
| CWE-257 | Storing Passwords in a Recoverable Format | AES Encryption | Encryption |
| CWE-261 | Weak Cryptography for Passwords | Base64 encoding | Base64 |
| CWE-916 | Use of Password Hash with Insufficient Computational Effort | 1000 PBKDF2 iterations | Fewer Iterations |
| CWE-759 | Use of a One-Way Hash without a Salt | No Salt | No Salt |
| CWE-760 | Use of a One-Way Hash with a Predictable Salt | Salt based on username | Predictable Salt |
| CWE-330 | Use of Insufficiently Random Values | *java.util.Random* | Insufficient Randomness |
| CWE-338 | Use of Cryptographically Weak PRNG | *java.util.Random* | Weak PRNG |

**Table 1.** Vulnerabilities related to password storage.

If password storage vulnerabilities are expected in the application source code, then it is important to detect and fix them in a timely manner, before the application is deployed. Several open-source and commercial source-code analysis tools are available that analyze applications for security vulnerabilities. Previous research mostly focused on understanding the insecure coding practices of developers [18][31][32][30] and the usability of cryptography APIs [17][45]. However, it is also important to understand the detection capabilities of the existing security testing tools and to check whether they assist developers in eliminating the detected vulnerabilities.

In this paper, we survey four Java source code analysis tools, SpotBugs [13], SonarQube [15], Cryptoguard [5] and CogniCrypt [6], and identify their capabilities in detecting security issues pertaining to password storage. We focus only on vulnerabilities shown in Table 1. Of these four tools, the first two tools are recommended by OWASP and the latter two tools are developed by security researchers [34][28]. A good testing tool not only detects and reports vulnerabilities, but also suggests remediation fixes wherever applicable. Therefore, we also analyze the recommended solutions provided by the surveyed tools and determine if they are consistent with the OWASP (or NIST) recommended password storage guidelines.

– We found that none of the surveyed tools covers all vulnerabilities related to password storage (Table 1).
– Most of the tools detect the use of weak PRNG *java.util.Random* and suggest to replace it with OWASP recommended cryptographically secure PRNG *java.security.SecureRandom* [4].
– All tools detect the use of weak hash functions MD5 and SHA-1 (CWE-327). However, we found multiple problems with the suggested solutions. CogniCrypt recommends the use of fast hash function SHA-256 whereas CryptoGuard does not recommend any solution at all.
– SpotBugs and SonarQube implement NIST recommended PBKDF2 for password hashing using HMAC-SHA-X, where X is 256 or 512, and PBEKeySpec API [2]. Both solutions use a PBEKeySpec constructor that requires user-supplied password along with three parameters, namely *salt*, *iterationCount* and *keyLength*. SpotBugs does not specify a value for salt and uses 4096 iterations which are not enough as per the latest NIST guidelines [25]. Whereas SonarQube does not specify values for any of the three parameters.
– We also conduct a usability study with 8 developers to further assess the recommended PBKDF2 solution of SonarQube. The study results show that since

the parameters of PBEKeySpec constructor are not specified in the solution, 6 developers chose weak values for at least one parameter.

The organization of this paper is as follows. First, we describe work related to passwords. Then, we describe a sample code containing different password storage vulnerabilities. Subsequently, we describe each tool briefly and evaluate it by running on a sample code. We also assess the remediation solutions provided by each tool. Later, we conduct a study to assess the recommended PBKDF2 solution of SonarQube. Finally, we conclude and discuss the future work.

## 2 Related Work

Text password offers several deployment benefits compared to alternative schemes such as graphical passwords and biometrics [19], thereby making it the most dominant authentication method on the internet. However, multiple security studies reveal that passwords suffer from several security and usability issues. For instance, users choose predictable passwords and reuse their passwords across multiple accounts [29][22][20][33]. To improve the security of text passwords, researchers explored diverse composition policies [37][36][35] and developed various interventions [44][42][23][40][12]. Further, to prevent users from choosing leaked passwords, free online services such as haveibeenpwned [12] are available that check user-entered password against millions of passwords in breached databases.

Most of the research studies involving developers were conducted either to understand how well developers implement security-related tasks or to test the usability of the existing cryptography APIs. Storing password data is one of the most common tasks carried out by software developers, however this task is prone to security issues. Acar *et al.* [18] recruited 307 active GitHub users and requested them to implement 3 security related tasks including a credential storage task. The authors found that, of the 307 participants, only 162 (52.8%) stored user credentials in a secure manner. Of the 145 participants who stored password insecurely, 74 (51%) hashed the password without using a proper salt, 45 (31%) participants stored the password in plaintext, 19 (13.1%) participants used a static salt instead of a random salt, 7 (4.8%) participants used MD5, while 6 (4.1%) used SHA-1 family hashes. Similar results were obtained in a recent study conducted by Naiakshina *et al.* [30] involving 43 freelance developers. Of the 43 participants, 10 (23.2%) participants used MD5, 8 (18.6%) participants used Base64 encoding, 7 (16.3%) participants used Bcrypt, 7 (16.3%) participants used SHA-1 family hashes, 6 (13.9%) participants used symmetric encryption and 5 (11.6%) participants used PBKDF2. Further, only 11 participants generated salt using strong PRNG (*java.security.SecureRandom*) whereas 2 participants used static salts, 1 participant used username and 1 generated salt using weak PRNG (*java.util.Random*).

Acar *et al.* [17] conducted an online study with 256 developers to investigate the usability of Python crypto-APIs. They found that poor documentation and missing code examples caused developers to struggle with security. Wijayarathna and Arachchilage [45] conducted a study with 10 developers to evaluate the usability of scrypt password hashing functionality of Bouncycastle API. The authors identified 63 usability issues developers face while using the API for secure password storage. Again, the key factors affecting the usability of API were poor documentation, lack of examples and difficulty in identifying correct parameters to use in API method invocation. Further, if the API is not properly documented, then developers refer to unreliable third party sources and tutorials which could put security of the entire application at risk [21].

Gorski *et al.* [24] conducted a controlled online experiment with 53 participants to study the effectiveness of API-integrated security advice, which informs about an API misuse and places secure programming hints as guidance. They found that 73% of the participants, who received the security warning and advice fixed their insecure code. In this paper, we survey several SAST tools to understand their detection capabilities and analyze their remediation fixes pertaining to insecure password storage. To the best of our knowledge, there has not been any study conducted to evaluate the recommended solutions of security testing tools in the context of password storage.

## 3 Approach

To determine the detection capabilities of each tool in the context of insecure password storage practices, we take the following two-step approach.

1. We refer to the online documentations of SpotBugs [1], SonarQube [9] and CogniCrypt [14] list different security vulnerabilities that they attempt to address along with remediation fixes. We focus only on vulnerabilities related to password storage listed in Table 1. The online documentation of CryptoGuard is not available, hence we refer to its paper [34].
2. To confirm the detection of password storage vulnerabilities as listed in the tool's documentation, we run it on sample code shown in Figure 1. The sample code consists of five methods, each demonstrating different vulnerabilities. The first four methods are derived from the CRYPTOAPI-BENCH created recently in [34]. The first method *hashPassword* returns an insecure SHA-1 hash of the password (CWE-327) and does not use salt (CWE-759). The second method *generateSalt* uses weak PRNG (CWE-330) with static seed and generates salt of insufficient size (4 bytes). The third method *getPBEParameterSpec* derives the values of salt (CWE-760) and number of iterations (CWE-916) required for PBEKeySpec from the user's password. The fourth method *encryptPassword* returns the encrypted version of the password (CWE-257). We added the fifth method *encodePassword* which returns base64 encoding of the password (CWE-261).

### 3.1 Tools

In this section, we describe each tool in more detail. The password storage vulnerabilities covered by each tool as per its online documentation are shown in Table 2. The vulnerabilities detected after executing each tool on sample code are summarized in Table 3. Both online documentation and execution results are in concurrence with each other. We developed sample code using Eclipse IDE (Oxygen 4.7.1a) and JDK 1.8 on Windows 10 (64-bit) machine. Also, we used

```
1  private static final int SALT_SIZE = 4;
2  private static byte[] hashPassword(String password) {
3    MessageDigest md = MessageDigest.getInstance("SHA-1"); //Weak Hash (CWE-327)
4    md.update(password.getBytes()); //No Salt (CWE-759)
5    return md.digest();
6  }
7  private static byte[] generateSalt() {
8    Random r = new Random(0); //Weak PRNG (CWE-330), Constant Seed 0
9    byte[] salt = new byte[SALT_SIZE]; //Insufficient Salt Size
10   r.nextBytes(salt);
11   return salt;
12 }
13 private static PBEKeySpec getPBEParameterSpec(String password) {
14   MessageDigest md = MessageDigest.getInstance("MD5"); //Predictable Salt (CWE-760)
15   byte[] saltGen = md.digest(password.getBytes());
16   byte[] salt = new byte[SALT_SIZE];
17   System.arraycopy(saltGen, 0, salt, 0, SALT_SIZE);
18   int iteration = password.toCharArray().length + 1; //Fewer Iterations (CWE-916)
19   return new PBEKeySpec(password.toCharArray(), salt, iteration, 256);
20 }
21 private static byte[] encryptPassword(String password, String key) {
22   Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
23   SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), "AES");
24   cipher.init(Cipher.ENCRYPT_MODE, secretKey);
25   return cipher.doFinal(password.getBytes()); //Encryption (CWE-257)
26 }
27 private static byte[] encodePassword(String password) {
28   Base64.Encoder encoder = Base64.getEncoder();
29   return encoder.encode(password.getBytes()); //Base64 (CWE-261)
30 }
```

**Fig. 1.** Sample code with password storage related vulnerabilities.

| *CWE-ID* | *Bug* | *SpotBugs* | *SonarQube* | *CryptoGuard* | *CogniCrypt* |
|---|---|---|---|---|---|
| CWE-327 | Weak Hash | ✓ | ✓ | ✓ | ✓ |
| CWE-256 | Plaintext | × | × | × | × |
| CWE-257 | Encryption | × | × | × | × |
| CWE-261 | Base64 | × | × | × | × |
| CWE-916 | Fewer Iterations | × | × | ✓ | × |
| CWE-759 | No Salt | × | × | × | × |
| CWE-760 | Predictable Salt | × | × | ✓ | ✓ |
| CWE-330, CWE-338 | Weak PRNG | ✓ | ✓ | ✓ | × |

**Table 2.** Detection capabilities of different tools as per their online documentation.

the latest versions of all security testing tools.

**SpotBugs (v4.0).** SpotBugs is an open source tool that uses static analysis approach to detect more than 400 vulnerabilities in Java applications. FindSecBugs (v1.10.1) is a plugin of SpotBugs which detects 135 different security vulnerabilities using over 816 unique API signatures [1]. Both SpotBugs and FindSecBugs are available as eclipse plugins. Running FindSecBugs on sample code, revealed three vulnerabilities which are depicted in Table 3. It detects the use of weak hash functions SHA-1 and MD5 at line numbers 3 and 14 respectively. It also detects the usage of weak PRNG *java.util.Random* at line number 8. FindSecBugs displays error markers within Eclipse IDE to highlight the lines of code with vulnerabilities and provides a brief description of each detected vulnerability as shown in Table 3. To view more details about vulnerability and suggested remediation, one can open the SpotBugs explorer and click on the error marker. We note that the detailed description of vulnerabilities given by FindSecBugs eclipse plugin matches exactly with its online documentation [1].

FindSecBugs recommends developers to use PBKDF2 instead of MD5 and SHA-1. It provides two different implementations of PBKDF2, one using bouncy

| *Code Line/Method* | *Mnemonic* | *Description* |
|---|---|---|
| **SpotBugs** | | |
| 3 | Weak Hash | This API SHA1 (SHA-1) is not a recommended cryptographic hash function |
| 8 | Weak PRNG | This random generator (java.util.Random) is predictable |
| 14 | Weak Hash | This API MD5 (MDX) is not a recommended cryptographic hash function |
| **SonarQube** | | |
| 3 | Weak Hash | Make sure that hashing data is safe here. |
| 8 | Weak PRNG | Make sure that using this pseudorandom number generator is safe here. |
| 14 | Weak Hash | Make sure that hashing data is safe here. |
| **CryptoGuard** | | |
| hashPassword | Weak Hash | Violated Rule 2: Found broken hash function ***Constants: [“SHA1” ] |
| generateSalt | Weak PRNG | Violated Rule 13: Untrused PRNG (java.util.Random) |
| getPBEParameterSpec | Weak Hash | Violated Rule 2: Found broken hash function ***Constants: [“MD5”] |
| getPBEParameterSpec | Predictable Salt | Violated Rule 9a: Found constant salts in code |
| getPBEParameterSpec | Fewer Iterations | Violated Rule 8a: Used < 1000 iteration for PBE |
| **CogniCrypt** | | |
| 3 | Weak Hash | First parameter (with value “SHA1”) should be any of SHA-256, SHA-384, SHA-512 |
| 14 | Weak Hash | First parameter (with value “MD5”) should be any of SHA-256, SHA-384, SHA-512 |
| 19 | Predictable Salt | Second parameter was not properly generated as randomized. |

**Table 3.** Vulnerabilities detected by different tools in sample code.

castle API and the other using cryptography API of Java 8 or later (refer to Figure 2). Although, both solutions employ NIST recommended HMAC-SHA-256 for password hashing, we found the following two issues in their implementation:

- They do not specify what the size of salt should be or how it should be generated. OWASP recommends at least 16 bytes unique random salt generated using cryptographically secure PRNG *java.security.SecureRandom*.
- The number of iterations used in both examples is 4096. This was sufficient according to older NIST 2010 guidelines [41], however it is not enough as per the latest NIST 2017 guidelines [25]. The current recommendation is to use at least 10,000 iterations.

```
/*Solution (Using bouncy castle):*/
public static byte[] getEncryptedPassword(String password, byte[] salt) {
    PKCS5S2ParametersGenerator gen = new PKCS5S2ParametersGenerator(new SHA256Digest());
    gen.init(password.getBytes("UTF-8"), salt.getBytes(), 4096);
    return ((KeyParameter) gen.generateDerivedParameters(256)).getKey();
}
/*Solution (Java 8 and later):*/
public static byte[] getEncryptedPassword(String password, byte[] salt) {
    KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 4096, 256 * 8);
    SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
    return f.generateSecret(spec).getEncoded();
}
```

**Fig. 2.** Solutions by FindSecBugs to replace weak hash functions MD5 and SHA-1.

Further, the PBEKeySpec solution uses 256 bytes key length when the recommendation is to use 256 bits [8]. FindSecBugs also recommends the use of cryptographically secure PRNG instead of weak PRNG (line 8). Specifically, its detailed reports says, “A quick fix could be to replace the use of java.util.Random with something stronger, such as java.security.SecureRandom.”

**SonarQube (v8.0).** SonarQube is a more comprehensive code quality and vulnerability detection tool that uses static analysis and supports 27 programming languages. It consists of 554 rules for detecting various security vulnerabilities in Java applications [9]. SonarQube is available as free community edition and three paid editions. It also comes in the form of eclipse plugin SonarLint, however we found that some of the detection rules are not available in SolarLint. Hence, we used its standalone free community edition. The results obtained after running SonarQube on sample code are shown in Table 3.

Similar to FindSecBugs, SonarQube also produces a detailed report along with remediation fixes. It suggests the replacement of weak PRNG *java.util.Random* (line 8) with cryptographically strong PRNG *java.security.SecureRandom*. It also suggests replacing MD5 and SHA-1 with PBKDF2, and provides a list of secure coding practices [7]. It uses NIST recommended HMAC-SHA-512 algorithm for implementing PBKDF2, however we found the following problem with its solution (Figure 3). It also employs PBEKeySpec constructor, but does not specify the values for its parameters (*salt*, *iterationCount* and *keyLength*).

```
void foo(char[] password, byte[] salt, int iterationCount, int keyLength) {
    SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512");
    PBEKeySpec spec = new PBEKeySpec(password, salt, iterationCount, keyLength);
    factory.generateSecret(spec).getEncoded();
}
```

**Fig. 3.** Solution by SonarQube to replace weak hash functions MD5 and SHA-1.

**CryptoGuard.** CryptoGuard is an open source high precision cryptographic vulnerabilities detection tool for Java applications [34]. It uses a set of fast and highly accurate slicing algorithms to detect 16 different cryptographic vulnerabilities such as predictable keys, constant passwords, custom trust manager, insecure random number generators, static salts, insecure cryptographic hash, and so on. It operates on source code, jar file and APK. The vulnerabilities detected after running CryptoGuard on sample code is shown in Table 3. Instead of reporting line numbers, CryptoGuard reports method names, which could be cumbersome for developers to locate the vulnerabilities exactly. Although CryptoGuard has more coverage in terms of detecting security issues pertaining to password storage, *it does not suggest any remediation fixes for them.*

**CogniCrypt (v1.0.0.201905151726).** CogniCrypt is an open source security vulnerabilities detection tool from the CROSSING research center of Technische Universität Darmstadt [28]. It employs static analysis and its scope is limited to the detection of inappropriate use of cryptography in Java applications. It comes with an important code generator feature to help developers in generating the right code for a given security requirement. The static analysis is based on rules developed in a domain-specific CrySL language that specify the correct use of an API [14]. The static analysis reports any deviations from the usage pattern defined within the rules. CogniCrypt is available as eclipse plugin and generates errors markers when it detects incorrect and insecure parts of code. Running CogniCrypt on sample code produced three vulnerabilities as described in Table 3. It detects the use of weak hash functions MD5 and SHA-1. However, *its remediation to use fast hash functions SHA-256, SHA-384 or SHA-512 results in insecure password storage code.* It also flags that salt is not generated randomly for PBEKeySpec, however *it does not specify how the salt should be generated.*

## 4 Study

CryptoGuard has comparatively good detection capabilities, however it does not provide any recommendation to implement secure password storage. CogniCrypt recommends the use of fast SHA-256 hash function, which is an insecure solution in the context of password storage. Therefore, we do not include these two tools in the study. SpotBugs recommends two PBKDF2 solutions, one using Bouncycastle API, and the other using HMAC-SHA-256 and PBEKeySpec API. Recently, researchers found several usability issues with Bouncycastle API [45], hence we do not consider it in our study. SonarQube also recommends a similar solution using HMAC-SHA-512 and PBEKeySpec API. The constructor of PBEKeySpec requires four parameters, namely user-supplied *password*, *salt*, *iterationCount* and *keyLength*. SpotBugs sets the value of *iterationCount* to 4096 and *keyLength* to 256 bytes, however it leaves the choice of salt to the developers (Figure 2). On the other hand, SonarQube does not specify the values of any of these three parameters (Figure 3). Therefore, we decided to evaluate whether SonarQube's detailed vulnerability report is helpful for developers to implement secure password solution. We note that study results pertaining to salt parameter are relevant to SpotBugs as well, since the salt parameter is unspecified in its recommended solutions.

**Methodology.** For the study, we designed a simple password storage task as described in [45]. We provided participants with a simple web application that includes functionalities for registering users and login users. The web application protected passwords using *hashPassword* function (given in Figure 1) which employs insecure SHA-1 hash algorithm (CWE 327). We requested participants to secure passwords using the vulnerability report generated by SonarQube. The report discourages the use of SHA-1 and provides a PBKDF2 implementation using Java PBEKeySpec API (Figure 3). The report is similar to the one available online [7]. Further, participants were allowed to access any resource on the internet in order to implement the recommended solution.

**Setup.** The study was setup on a dedicated Windows 10 (64-bit) machine. We created the web application project in Eclipse Oxygen (4.7.1a) using JDK 1.8. The function *hashPassword* was present in a separate source file, so that participants could focus on the task. Participants were provided the vulnerability report of SonarQube and were allowed to use Chrome browser for implementing the solution. At the end of the task, we stored the browsing history of each participant. For the implementation to work properly, participants were required to choose three parameters (*salt*, *iterationCount* and *keyLength*) of PBEKeySpec.

**Result.** We recruited 8 developers within our organization for the study. The information profile about each participant, their choices for three parameters *salt*, *iterationCount* and *keyLength* of PBEKeySpec and the time required for implementing the solution are shown in Table 4. We found that only two participants (P2, P7) chose the parameters as recommended by NIST as they had relevant experience of password storage task. These participants were aware of PBKDF2 specification and generated a unique 16 bytes salt using cryptographically secure PRNG *java.security.SecureRandom*, set *iterationCount* to 10,000

and *keyLength* to 256 bits. Of the remaining 6 participants, one participant (P5) generated a secure unique random salt for each password, four participants used a constant salt and one participant used userid as salt. Participants P4 and P8 set the value of *iterationCount* to 10,000 whereas the remaining participants 4 participants chose insufficient number of iterations.
Most of the participants chose a correct value of *keyLength* (256 bits). Analysis of browsing history of these participants reveal that they referred to the OWASP web page [8] (link was provided by SonarQube in its vulnerability report) which recommends the value of key length to be 256 bits, however it does not specify the number of iterations. Interestingly, the same page also recommends the size of salt to be 32 bytes. Four participants browsed Oracle's documentation for PBEKeySpec, however it does not recommend any values for the parameters. Three participants searched for the concept of salt (wikipedia). The concept of salt is not widely known, which was also observed in the previous study [31]. We also tested the submitted implementation of each participant using SonarQube. However, *none of the submitted solutions were flagged for vulnerabilities by SonarQube which is a serious concern.* Thus, detecting vulnerabilities just using method signatures is not enough.
**Limitations.** We found that developers chose incorrect parameters and implemented insecure password storage when the security testing tools do not provide specific recommendations. Our observation is based on a convenience sample of 8 developers. However, similar observations were made in a recent study [45] pertaining to usability of cryptography APIs. Their study results [45] also show that developers have difficulty in identifying correct parameters to use in Bouncycastle API method invocation.

| *Participant* | *Development Experience* | *Stored Pass -words Before* | *salt* | *iteration Count* | *keyLength* | *Time* (in min) |
|---|---|---|---|---|---|---|
| P1 | 5.7 years | × | 2 bytes (constant value) | 50 | 256 | 25 min |
| P2 | 7.6 years | √ | 16 bytes (unique SecureRandom) | 10,000 | 256 | 12 min |
| P3 | 6.8 years | √ | Userid | 12 | 256 | 29 min |
| P4 | 2.8 years | × | 11 bytes (constant value) | 10,000 | 256 | 27 min |
| P5 | 3 years | √ | 16 bytes (unique SecureRandom) | 0 | 0 | 32 min |
| P6 | 0.6 years | × | 8 bytes (constant value) | 20 | 222 | 30 min |
| P7 | 15 years | √ | 16 bytes (unique SecureRandom) | 10,000 | 256 | 9 min |
| P8 | 3 years | × | 16 bytes (constant value) | 10,000 | 256 | 35 min |

**Table 4.** Participants information and their choice of parameters for PBEKeySpec.

## 5 Conclusion and Future Work

In this paper, we surveyed four open-source security testing tools (SpotBugs, SonarQube, CryptoGuard and CogniCrypt) to understand their detection capabilities pertaining to password storage vulnerabilities. We found that CryptoGuard has comparatively good coverage, however it does not specify any remediation to fix the insecure password storage code. CogniCrypt detects the use of weak hash functions (MD5 and SHA-1), however it suggests SHA-256 hash function, which is insecure in the context of passwords. Both SpotBugs and

SonarQube recommend the use of PBKDF2 and provide example solutions using PBEKeySpec API. However, SonarQube leaves the equally important choice of PBKDF2 parameters (*salt*, *iterationCount* and *keyLength*) to developers. Further, SpotBugs solution uses 4096 iterations which is insufficient as per the latest NIST 2017 guidelines [25] and leaves the important choice of *salt* parameter to developers. In our study involving 8 developers who were tasked with implementing SonarQube's recommended solution, we found that 6 of them chose insecure values for at least one PBKDF2 parameter. Therefore, it is crucial that security testing tools provide specific password storage solutions to developers.

We note that the insecure password storage is just one of the many implementation issues associated with the code that handles passwords. Other issues include hard-coded password (CWE-259), password in configuration file (CWE-260) and exposure of passwords in log files (CWE-200). Further, several commercial SAST tools such as Synopsys Coverity and HP Fortify are available. In future, we aim to compare the detection capabilities and remediation fixes of open-source as well as commercial tools with regard to insecure password code.

## References

1. Bugs Patterns. `https://find-sec-bugs.github.io/bugs.htm`, Last accessed 19 December 2019
2. Class PBEKeySpec. `https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/PBEKeySpec.html`, Last accessed 19 December 2019
3. Common Weakness Enumeration. `https://cwe.mitre.org/`, Last accessed 19 December 2019
4. Cryptographic Storage. `https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html`, Last accessed 19 December 2019
5. CryptoGuard. `https://github.com/CryptoGuardOSS/cryptoguard`, Last accessed 19 December 2019
6. Eclipse CogniCrypt. `https://www.eclipse.org/cognicrypt/`, Last accessed 19 December 2019
7. Hashing data is security-sensitive. `https://rules.sonarsource.com/java/RSPEC-4790`, Last accessed 19 December 2019
8. Hashing Java. `https://www.owasp.org/index.php/Hashing_Java`, Last accessed 19 December 2019
9. Jave 554 Rules. `https://rules.sonarsource.com/java/`, Last accessed 19 December 2019
10. OWASP Top 10 - 2017. `https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf`, Last accessed 19 December 2019
11. Password Storage. `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html`, Last accessed 19 December 2019
12. Pwned websites- Breached websites that have been loaded into Have I Been Pwned. `https://haveibeenpwned.com/PwnedWebsites`, Last accessed 19 December 2019
13. SpotBugs. `https://spotbugs.github.io/`, Last accessed 19 December 2019
14. The CrySL Language. `https://www.eclipse.org/cognicrypt/documentation/crysl/`, Last accessed 19 December 2019
15. Your teammate for Code Quality and Security. `https://www.sonarqube.org/`, Last accessed 19 December 2019

16. Anatomy of a password disaster – Adobe's giant-sized cryptographic blunder. https://nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic-blunder/ (2019)
17. Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M.L., Stransky, C.: Comparing the Usability of Cryptographic APIs. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 154–171 (May 2017). https://doi.org/10.1109/SP.2017.52
18. Acar, Y., Stransky, C., Wermke, D., Mazurek, M.L., Fahl, S.: Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In: Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017). pp. 81–95. USENIX Association, Santa Clara, CA (Jul 2017), https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar
19. Bonneau, J., Herley, C., v. Oorschot, P.C., Stajano, F.: The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In: 2012 IEEE Symposium on Security and Privacy. pp. 553–567 (May 2012). https://doi.org/10.1109/SP.2012.44
20. Das, A., Bonneau, J., Caesar, M., Borisov, N., Wang, X.: The Tangled Web of Password Reuse. In: NDSS. vol. 14, pp. 23–26 (2014)
21. Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., Fahl, S.: Stack overflow considered harmful? the impact of copy paste on android application security. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 121–136 (May 2017). https://doi.org/10.1109/SP.2017.31
22. Florencio, D., Herley, C.: A Large-scale Study of Web Password Habits. In: Proceedings of the 16th International Conference on World Wide Web. pp. 657–666. WWW '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1242572.1242661, http://doi.acm.org/10.1145/1242572.1242661
23. Forget, A., Chiasson, S., van Oorschot, P.C., Biddle, R.: Improving Text Passwords Through Persuasion. In: Proceedings of the 4th Symposium on Usable Privacy and Security. pp. 1–12. SOUPS '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1408664.1408666, http://doi.acm.org/10.1145/1408664.1408666
24. Gorski, P.L., Iacono, L.L., Wermke, D., Stransky, C., Möller, S., Acar, Y., Fahl, S.: Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In: Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018). pp. 265–281. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/soups2018/presentation/gorski
25. Grassi, P.A., Fenton, J.L., Newton, E.M., Perlner, R.A., Regenscheid, A.R., Burr, W.E., Richer, J.P., Lefkovitz, N.B., Danker, J.M., Choong, Y.Y., Greene, K.K., Theofanos, M.F.: Digital identity guidelines. NIST special publication **800**, 63–3 (2017). https://doi.org/10.6028/NIST.SP.800-63b
26. Green, M., Smith, M.: Developers are Not the Enemy!: The Need for Usable Security APIs. IEEE Security Privacy **14**(5), 40–46 (Sep 2016). https://doi.org/10.1109/MSP.2016.111
27. Kamp, P.H.: LinkedIn Password Leak: Salt Their Hide. Queue **10**(6), 20:20–20:22 (Jun 2012). https://doi.org/10.1145/2246036.2254400, http://doi.acm.org/10.1145/2246036.2254400

28. Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., Kamath, R.: CogniCrypt: Supporting Developers in Using Cryptography. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. pp. 931–936. ASE 2017, IEEE Press, Piscataway, NJ, USA (2017), http://dl.acm.org/citation.cfm?id=3155562.3155681
29. Morris, R., Thompson, K.: Password Security: A Case History. Commun. ACM **22**(11), 594–597 (Nov 1979). https://doi.org/10.1145/359168.359172, http://doi.acm.org/10.1145/359168.359172
30. Naiakshina, A., Danilova, A., Gerlitz, E., von Zezschwitz, E., Smith, M.: &#34;If You Want, I Can Store the Encrypted Password&#34;: A Password-Storage Field Study with Freelance Developers. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. pp. 140:1–140:12. CHI '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3290605.3300370, http://doi.acm.org/10.1145/3290605.3300370
31. Naiakshina, A., Danilova, A., Tiefenau, C., Herzog, M., Dechand, S., Smith, M.: Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 311–328. CCS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134082, http://doi.acm.org/10.1145/3133956.3134082
32. Naiakshina, A., Danilova, A., Tiefenau, C., Smith, M.: Deception Task Design in Developer Password Studies: Exploring a Student Sample. In: Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018). pp. 297–313. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/soups2018/presentation/naiakshina
33. Pearman, S., Thomas, J., Naeini, P.E., Habib, H., Bauer, L., Christin, N., Cranor, L.F., Egelman, S., Forget, A.: Let's Go in for a Closer Look: Observing Passwords in Their Natural Habitat. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 295–310. CCS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3133973, http://doi.acm.org/10.1145/3133956.3133973
34. Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.D.: 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. CCS '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3345659, http://doi.acm.org/10.1145/3319535.3345659
35. Segreti, S.M., Melicher, W., Komanduri, S., Melicher, D., Shay, R., Ur, B., Bauer, L., Christin, N., Cranor, L.F., Mazurek, M.L.: Diversify to Survive: Making Passwords Stronger with Adaptive Policies. In: Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017). pp. 1–12. USENIX Association, Santa Clara, CA (Jul 2017), https://www.usenix.org/conference/soups2017/technical-sessions/presentation/segreti
36. Shay, R., Kelley, P.G., Komanduri, S., Mazurek, M.L., Ur, B., Vidas, T., Bauer, L., Christin, N., Cranor, L.F.: Correct Horse Battery Staple: Exploring the Usability of System-assigned Passphrases. In: Proceedings of the Eighth Symposium on Usable Privacy and Security. pp. 7:1–7:20. SOUPS '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2335356.2335366, http://doi.acm.org/10.1145/2335356.2335366

37. Shay, R., Komanduri, S., Durity, A.L., Huh, P.S., Mazurek, M.L., Segreti, S.M., Ur, B., Bauer, L., Christin, N., Cranor, L.F.: Can Long Passwords Be Secure and Usable? In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 2927–2936. CHI '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2556288.2557377, http://doi.acm.org/10.1145/2556288.2557377
38. Sprengers, M.: GPU-based Password Cracking. Master's thesis, Radboud University Nijmegen Faculty of Science Kerckhoffs Institute (2011)
39. Stobert, E., Biddle, R.: The Password Life Cycle: User Behaviour in Managing Passwords. In: 10th Symposium On Usable Privacy and Security (SOUPS 2014). pp. 243–255. USENIX Association, Menlo Park, CA (Jul 2014), https://www.usenix.org/conference/soups2014/proceedings/presentation/stobert
40. Tupsamudre, H., Dixit, A., Banahatti, V., Lodha, S.: Pass-Roll and Pass-Scroll: New Graphical User Interfaces for Improving Text Passwords. In: EuroUSEC (2017)
41. Turan, M.S., Barker, E., Burr, W., Chen, L.: Recommendation for password-based key derivation. NIST special publication **800**, 132 (2010)
42. Ur, B., Alfieri, F., Aung, M., Bauer, L., Christin, N., Colnago, J., Cranor, L.F., Dixon, H., Emami Naeini, P., Habib, H., Johnson, N., Melicher, W.: Design and Evaluation of a Data-Driven Password Meter. In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. pp. 3775–3786. CHI '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3025453.3026050, http://doi.acm.org/10.1145/3025453.3026050
43. Ur, B., Noma, F., Bees, J., Segreti, S.M., Shay, R., Bauer, L., Christin, N., Cranor, L.F.: "I Added '!' at the End to Make It Secure": Observing Password Creation in the Lab. In: Eleventh Symposium On Usable Privacy and Security (SOUPS 2015). pp. 123–140. USENIX Association, Ottawa (Jul 2015), https://www.usenix.org/conference/soups2015/proceedings/presentation/ur
44. Wheeler, D.L.: zxcvbn: Low-budget password strength estimation. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 157–173. USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler
45. Wijayarathna, C., Arachchilage, N.A.G.: Why Johnny Can't Store Passwords Securely?: A Usability Evaluation of Bouncycastle Password Hashing. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018. pp. 205–210. EASE'18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3210459.3210483, http://doi.acm.org/10.1145/3210459.3210483