Secure Coding Practices in Java: Challenges and Vulnerabilities

Na Meng Stefan Nagy Daphne Yao Wenjie Zhuang Gustavo Arango Argoty
Virginia Tech
Blacksburg, Virginia 24060
{nm8247,snagy2,danfeng,kaito,gustavo1}@vt.edu

ABSTRACT

Java platform and third-party libraries provide functionalities to facilitate secure coding. However, misusing these functionalities can cost developers tremendous time and effort, or introduce security vulnerabilities in software. Prior research focused on the misuse of cryptography and SSL APIs, but did not explore the fundamental research question: what are the biggest challenges and vulnerabilities in secure coding practices? In this paper, we conducted a broader empirical study on StackOverflow posts to understand developers' concerns on Java secure coding, their programming obstacles, and the potential vulnerabilities in their code.

We observed that developers have shifted their effort to the usage of authentication and authorization features provided by Spring Security—a third-party framework designed to secure enterprise applications. The programming challenges are all related to APIs or libraries, including the complicated cross-language data handling of cryptography APIs, and the complex Java-based or XML-based approaches to configure Spring Security. More interestingly, *we identified security vulnerabilities in the suggested code of accepted answers*. The vulnerabilities included using insecure hash functions (e.g., MD5), breaking SSL/TLS security through bypassing certificate validation, and insecurely disabling the default protection against Cross Site Request Forgery attacks. Our findings reveal the insufficiency of secure coding assistance and documentation, as well as the gap between security theory and coding practices.

CCS CONCEPTS

• **General and reference → Empirical studies**;

KEYWORDS

CSRF, SSL/TLS certificate validation, cryptographic hash functions, authentication, authorization, StackOverflow posts

ACM Reference format:
Na Meng Stefan Nagy Daphne Yao Wenjie Zhuang Gustavo Arango Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *Proceedings of ACM conference, Gothenburg, Sweden, May 2018 (ICSE'18),* 13 pages.
https://doi.org/10.475/123_4

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE'18, May 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

Java platform and third-party libraries or frameworks (e.g., BouncyCastle [6]) provide various features to enable secure coding. Misusing these libraries and frameworks not only slows development time, but also leads to security vulnerabilities in the resulting software [14, 98, 99, 103].

Prior research mainly focused on cryptography and SSL API misuse causing security vulnerabilities [80, 82, 85, 88]. Specifically, Lazar et al. manually examined 269 published cryptographic vulnerabilities in the CVE database, and observed 83% of them resulted from cryptography API misuse [88]. Nadi et al. further investigated the obstacles introduced by Java cryptography APIs, developers' usage of the APIs, and desired tool support [94]. Fahl et al. [82] and Georgiev et al. [85] separately implemented a man-in-the-middle attack, and detected vulnerable Android applications and software libraries misusing SSL APIs. Despite these studies, it is still unknown what the major concerns are in secure coding practices, and whether these practices benefitted from security research over time.

In this paper, we conducted *a broader in-depth investigation on the common concerns, programming challenges, and security vulnerabilities in developers' secure coding practices* by inspecting 503 StackOverflow (SO) posts related to Java security. We chose SO [63] because (1) it is a popular online platform for developers to share and discuss programming issues and solutions, and (2) SO plays an important role in educating developers and shaping their daily coding practices. The main challenge of conducting this empirical study is *interpreting security-relevant programming issues or solutions within both program and security contexts.* To comprehend each post within the program context, we manually checked all included information related to the source code, configuration files, and/or execution environments. Then, we identified the root causes and solutions of each problem. To comprehend each post within the security context, we collected information about the developers' implementation intents and the involved security libraries, and determined whether the final implementation fulfilled the intents. Such manual analysis requires so much expertise in both software engineering and security that it is difficult to automate.

In our thorough manual analysis of the 503 posts, we investigated the following three research questions (RQs):

RQ1 *What are the common concerns in Java secure coding?* Although there are various security libraries and frameworks [2, 32, 34, 57, 86, 95], we do not know which libraries and functionalities are most frequently asked about by developers.

RQ2 *What are the common programming challenges?* We aim to identify the common obstacles preventing developers from easily and correctly implementing secure code. Such information will provide SE researchers actionable knowledge to better develop tools, and help close the gap between intended versus actual library usage.

RQ3 *What are the common security vulnerabilities?* We aim to identify security the vulnerabilities spread on SO, since their gaining popularity may cause insecure code to see widespread implementation. This effort will help raise security consciousness among software developers.

In our study, we made three major observations.

- *There were security vulnerabilities in the recommended code of some accepted answers.* For example, usage of MD5 or SHA-1 algorithms was repeatedly suggested, although these algorithms are notoriously insecure and should not be used. Additionally, many developers were advised to trust all incoming SSL/TLS certificates as a temporary fix to certificate verification errors. Such action completely disrupts the security of SSL. Although this bad practice was initially reported by researchers in 2012 [82, 85], developers have still asked for and accepted it until now. Furthermore, when encountering errors in implementing Spring Security authentication, developers were often suggested a workaround to blindly disable the default security protection against Cross Site Request Forgery (CSRF) attacks.
- *Various programming challenges were related to security library usage.* For instance, developers became stuck using cryptography APIs due to clueless error messages, complex cross-language data handling, and delicate implicit API usage constraints. However, when using Spring Security, developers struggled with the two alternative ways of configuring security: Java-based or XML-based.
- *Since 2012, developers have increasingly relied on Spring Security for secure coding.* 267 of the 503 examined posts (53%) were about Spring Security. However, to the best of our knowledge, research has not yet investigated security vulnerabilities related to this framework.

The significance of this work is our empirical evidence for many significant secure coding issues which have not been previously reported on. Compared with OWASP [47] and prior studies (e.g., Nadi et al. [94] and Acar et al. [73]), our research analyzed developers' code to reveal both implementation obstacles and security vulnerabilities. We further investigated the posts' discussion content, and leveraged our security expertise to assess the potential vulnerabilities disseminated. Our findings will motivate new research to help developers overcome the observed issues in the long term.

2 BACKGROUND

The examined posts were mainly about three perspectives of Java security: Java platform security, Java EE security, and other third-party frameworks. This section introduces the key terminologies used throughout the paper.

2.1 Java Platform Security

The platform defines APIs spanning major security areas, including cryptography, access control, and secure communication [41].

The *Java Cryptography Architecture (JCA)* contains APIs for **hashes**, **keys and certificates**, **digital signatures**, and **encryption** [34]. Nine cryptographic engines are defined to provide either cryptographic operations (encryption, digital signatures, hashes), generators or converters of cryptographic material (keys and algorithm parameters), or objects (keystores or certificates) that encapsulate the cryptographic data.

The *access control architecture* protects the access to sensitive resources (e.g., local files) or sensitive application code (e.g., methods in a class). All access control decisions are mediated by a **security manager**. By default, the security manager uses the `AccessController` class for access control operations and decisions.

Secure communication ensures that the data which travels across a network is sent to the appropriate party, without being modified during the transmission. Cryptography forms the basis for secure communication. The Java platform provides API support for standard secure communication protocols like **SSL/TLS**. **HTTPS**, or "HTTP secure", is an application-specific implementation that is a combination of HTTP and SSL/TLS.

2.2 Java EE Security

Java EE is an standard specification for enterprise Java extensions [44]. Various application servers are built to implement this specification, such as **JBoss** or **WildFly** [71], **Glassfish** [18], **WebSphere** [69], and **WebLogic** [3]. A Java EE application consists of components deployed into various containers. The Java EE security specification defines that containers secure components by supporting features like authentication and authorization.

In particular, **authentication** defines how communicating entities, such as a client and a server, prove to each other that they are who they say they are. An authenticated user is issued a *credential*, which includes user information like usernames/passwords or tokens. **Authorization** ensures that users have permissions to perform operations or access data. When accessing a certain resource, a user is authorized if the server can map this user to a security role permitted for the resource.

Java EE applications' security can be implemented in two ways:

- **Declarative Security** expresses an application component's security requirements using either **deployment descriptors** or **annotations**. A deployment descriptor is an XML file external to the application. This XML file expresses an application's security structure, including security roles, access control, and authentication requirements. Annotations are used to specify security information in a class file. They can be either used or overridden by deployment descriptors.
- **Programmatic Security** is embedded in an application and is used to make security decisions, when declarative security alone is not sufficient to express the security model.

2.3 Other Third-Party Security Frameworks

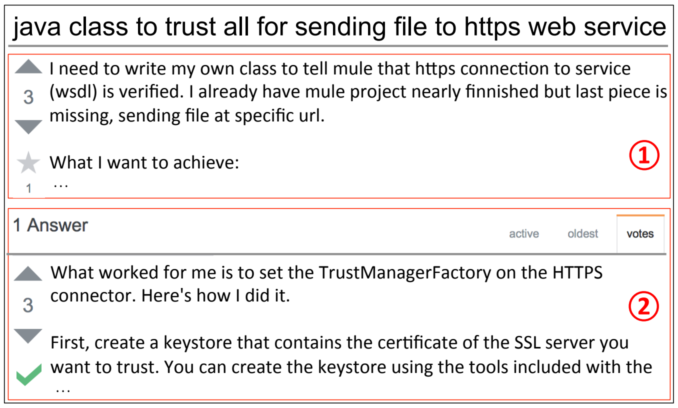
Several frameworks were built to provide authentication, authorization, and other security features for enterprise applications, such as **Spring Security** (SS) [54]. Different from the Java EE security APIs, these frameworks are container independent, meaning that they do not require containers to implement security. For example, SS handles requests as a single **filter** inside a container's filter chain. There can be multiple security filters inside the SS filter. Developers can choose between **XML-based** and **Java-based** security configurations, or a hybrid of the two. Similar to Java EE security, the XML-based configuration implements security requirements with deployment descriptors and source code, while the Java-based expresses security with annotations and code.

3 METHODOLOGY

We leveraged the open source python library Scrapy [51] to crawl posts from the StackOverflow (SO) website. Figure 1 presents the format of a typical SO post. Each post mainly contains two regions: the question and answers.

① **Question region** contains the question description and some metadata. The metadata includes a **vote** for the question (e.g., 3), indicating whether the question is well-defined or well-representative, and a **favorite count** (e.g., 1) showing how many people liked the question.

② **Answer region** contains all answer(s) provided. When one or more answers are provided, the asker decides which answer to **accept**, and marks it with (✓).


java class to trust all for sending file to https web service

3 I need to write my own class to tell mule that https connection to service (wsdl) is verified. I already have mule project nearly finnished but last piece is missing, sending file at specific url.

What I want to achieve:
1 ...

①

1 Answer

3 What worked for me is to set the TrustManagerFactory on the HTTPS connector. Here's how I did it.

②

✓ First, create a keystore that contains the certificate of the SSL server you want to trust. You can create the keystore using the tools included with the
...

Figure 1: A highly viewed post (viewed 556 times) asking about HTTPS workarounds to bypass key checking and allow all host names [33]

We obtained 22,195 posts containing keywords "java" and "security". After extracting the question, answers, and relevant metadata for each post, we refined the data in three ways.

1) Filtering less useful posts. We automatically refined posts by removing duplicated posts, posts without accepted answers, and posts whose questions received negative votes (perhaps because the questions were ill-formed or confusing).

2) Removing posts without code snippets. To better understand the questions within the program context, we only focused on posts containing code snippets. Since our crawled data did not include any metadata describing the existence of code snippets, we developed an intuitive filter to search for keywords "public" and "class" in each post. Based on our observation, a post usually contains these two keywords when it includes a code snippet.

3) Discarding irrelevant posts. After applying the above two filters, we manually examined the remaining posts, and decided whether they were relevant to Java secure coding, or simply contained the checked keywords accidentally.

With the above three filters, we finally included 503 posts in our dataset asked between 2008-2016. We did not include posts asked in 2017, because at the time we conducted experiments, data for only the first several months of 2017 was available. When manually filtering retrieved posts, we also characterized relevant posts based on their *security concerns*, *programming challenges*, and *security vulnerabilities*. Based on this characterization, we classified posts and investigated the following three research questions (RQs):

RQ1: What are the common security concerns of developers? We aimed to investigate: (1) what are the popular security libraries or functionalities that developers frequently asked about, and (2) how have developers' security concerns shifted over the years? Since we had no prior knowledge of developers' security concerns, we adopted an open coding approach to classify posts accordingly. Specifically, Author 4 initially categorized posts based on the software libraries and security concepts discussed. Author 1 (an SE professor) then iteratively reviewed posts to create and adjust the identified security concerns. Next, Author 2 examined around 150 posts suggested by Author 1 to identify security vulnerabilities in their answers. To ensure high quality of findings, the two authors cross checked results, and resolved disagreement when necessary with Author 3 (a cybersecurity professor).

We also classified posts into three categories based on the number of positive votes and favorite counts their questions received:

- Neutral: A question does not have any positive vote or favorite count.
- Positive: A question receives at least one positive vote but zero favorite count.
- Favorite: A question obtains at least one favorite vote.

Thus, the post in Figure 1 is classified as "Favorite", because its favorite count is 1. By combining this categorization with the security concerns, we explored developers' attitudes towards questions related to different concerns. We posit that although some security concerns are common, people view their questions unfavorably, possibly because they are so complicated and project-specific that most developers cannot learn or benefit from them.

RQ2: What are the common programming challenges? For each identified security concern, we further characterized each post with its problem (buggy source code, wrongly implemented configuration files, improperly configured execution environment), the problem's root cause, and the accepted solution. We then clustered posts with similar characterizations. For the post in Figure 1, we identified its problem as being a request for a SSL verification workaround; apparently the developer was unaware SSL should not be bypassed. Its recommended solution was to first create a keystore that contains the certificates of all trusted SSL servers, and then use this keystore to instantiate a `TrustManagerFactory` for establishing (unverified) connections.

RQ3: What are the common security vulnerabilities? To further our understanding of each post's security context, we also inspected unaccepted answers and conversational comments between the question asker and other developers, Based on recommended secure coding practices and the post's security context, we decided whether the accepted solution was security-vulnerable. The post shown in Figure 1 has a secure accepted answer, although the asker originally requested a vulnerable solution as an easy fix.

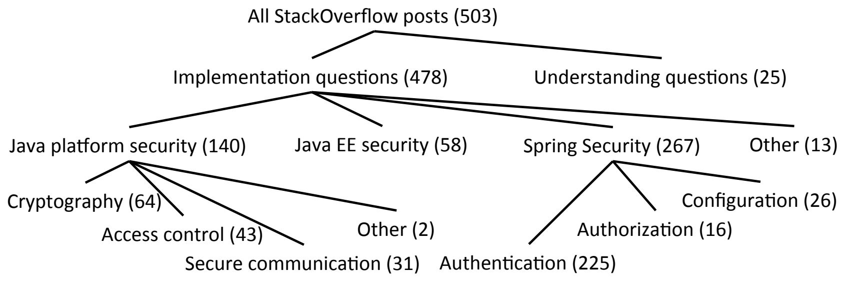


Figure 2: Taxonomy of StackOverflow posts

4 MAJOR FINDINGS

We present our investigation results for the research questions separately in Section 4.1-4.3.

4.1 Common Concerns in Security Coding

Figure 2 presents our classification hierarchy among the 503 posts. At the highest level, we created two categories: **implementation questions** vs. **understanding questions**. The majority (478 posts) were about implementing security functionalities or resolving program errors. Only 25 questions were asked to understand why specific features were designed in certain ways (e.g., "How does Java string being immutable increase security?" [22]) Since our focus is on secure coding practices, our further classification expands on the 478 implementation-relevant posts.

At the second level of the hierarchy, we clustered posts based on the *major security platforms or frameworks* involved in each post. Corresponding to Section 2, we identified posts relevant to **Java platform security**, **Java EE security**, **Spring Security**, and **other** third-party security libraries or frameworks.

At the third level, we classified the posts belonging to either Java platform security or Spring Security, because both categories contained many posts. Among the Java platform security posts, in addition to **cryptography** and **secure communication**, we identified a third major concern—**access control**. Among the Spring Security posts, we found the majority (225) related to **authentication**, with the minority discussing **authorization** and **configuration**.

Finding 1: *56%, 29%, and 12% of the implementation-relevant posts focused on Spring Security, Java platform security, and Java EE security, indicating that developers need more help to secure Java enterprise applications.*

Based on the second- and third-level classifications, we identified seven major security concerns: cryptography, access control, secure communication, Java EE security, authentication, authorization, and configuration. The first three concerns correspond to Java platform security, while the last three correspond to Spring Security. To reveal trends in developers' security concerns over time, we clustered posts based on the year each question was asked.

Figure 3 presents the post distribution among 2008-2016. The total number of posts increased over the years, indicating that more developers became involved in secure coding and faced problems. Specifically, there was only 1 post created in 2008, but 107 posts were created in 2016. During 2009-2011, most posts were about Java

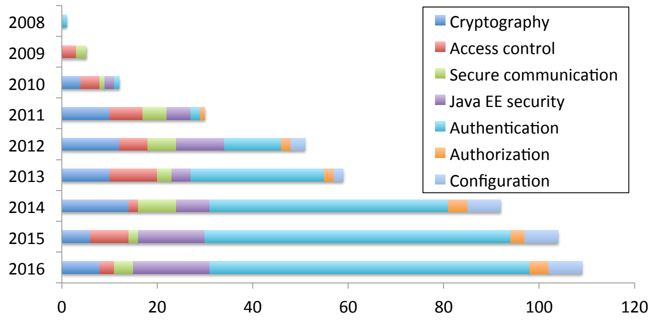


Figure 3: The post distribution during 2008-2016

platform security. However, since 2012, the major security concern has shifted to securing Java enterprise applications (including both Java EE security and Spring Security). Specifically, Spring Security has taken up over 50% of the posts published every year since 2013.

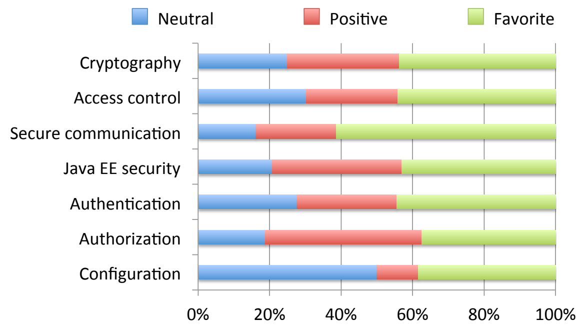


Figure 4: The post distribution among developers' attitudes: neutral, positive, and favorite

As shown in Figure 4, we also clustered posts based on developers' attitudes towards the questions for each security concern. The configuration posts received the highest percentage of neutral opinions (50%). One possible reason is that these posts mainly focused on problems caused by incorrect library versions and library dependency conflicts. Since such problems are usually specific to software development environments, they are not representative or relevant to many developers' security interests. In comparison, secure communication posts received the lowest percentage of neutral opinions (16%), but the highest percentage of favorite (61%), indicating that the questions were more representative, focusing more on security implementation instead of environment configuration.

Finding 2: *Over time, developers' major security concern has shifted from securing Java platform to enterprise applications, especially the Spring Security framework. Secure communication posts received the highest percentage (61%) of favorite votes, indicating that these questions are both important and representative.*

4.2 Common Programming Challenges

To understand the common challenges developers faced, we further examined the posts of the most popular five major categories: authentication (225), cryptography (64), Java EE security (58), access control (43), and secure communication (31). We identified posts with similar questions and related answers, and further investigated why developers asked these common questions. This section presents our key findings for each category.

4.2.1 Authentication. Most posts were related to (1) integrating Spring security with different application servers (e.g., JBoss) [59] or frameworks (e.g., Spring MVC) [55] (35 posts), (2) configuring security in an XML-based [56] or Java-based way [27] (145 posts), or (3) converting XML-based configurations to Java-based ones [10] (18 posts). Specifically, we observed three challenges.

Challenge 1: There is much variation in integrating Spring Security (SS) with different types of applications. Although SS can be used to secure enterprise applications no matter whether the applications are Spring-based or not, the usage varies with the application settings [58]. Even worse is that some SS-relevant implementations may exhibit different dynamic behaviors in different application contexts. As shown in Listing 1, by following a standard tutorial example [68], a developer defined two custom authentication filters—`apiAuthenticationFilter` and `webAuthenticationFilter`—to secure two sets of URLs of his/her Spring Boot web application.

Listing 1: An exemplar implementation working unexpectedly in Spring Boot applications [13]

```
1  @EnableWebSecurity
2  public class SecurityConfiguration {
3    @Configuration @Order(1)
4    public static class ApiConfigurationAdapter
5        extends WebSecurityConfigurerAdapter {
6      @Bean
7      public GenericFilterBean
8          apiAuthenticationFilter() {...}
9      @Override
10     protected void configure(HttpSecurity http)
11         throws Exception {
12       http.antMatcher("/api/**")
13         .addFilterAfter(apiAuthenticationFilter()...)
14         .sessionManagement()...;   } }
15   @Configuration @Order(2)
16   public static class WebSecurityConfiguration
17       extends WebSecurityConfigurerAdapter {
18     @Bean
19     public GenericFilterBean
20         webAuthenticationFilter() {...}
21     @Override
22     protected void configure(HttpSecurity http)
23         throws Exception {
24       http.antMatcher("/")
25           .addFilterAfter(webAuthenticationFilter()...)
26           .authorizeRequests()...; } } }
```

In Listing 1, lines 3-14 correspond to `ApiConfigurationAdapter`, a security configuration class that specifies `apiAuthenticationFilter` to authenticate URLs matching the pattern "`/api/**`". Lines 15-26 correspond to `WebSecurityConfiguration`, which configures `webAuthenticationFilter` to authenticate the other URLs. Ideally, only one filter is invoked given one URL, however in reality, both filters were invoked. The root cause is that each filter is a bean (annotated with @Bean on lines 6 and 18). Spring Boot detects the filters and adds them to a regular filter chain, while SS also adds them to its own filter chain. Consequently, both filters are registered twice and can be invoked twice. To solve the problem, developers need to enforce each bean to be registered *only once* by adding specialized code. Unfortunately, the tutorial example was documented without any clarification about the scenarios where the code does not work.

Challenge 2: The two security configurations (Java-based and XML-based) are hard to implement correctly. Take the Java-based configuration for example. There are lots of annotations and APIs of classes, methods, and fields available to specify different configuration options. Particularly, `HttpSecurity` has 10 methods, each of which can be invoked on an `HttpSecurity` instance and then produces another `HttpSecurity` object. If developers are not careful about the invocation order between these methods, they can get errors [25]. As shown in Listing 1, the method `antMatcher("/api/**'')` must be invoked *before* `addFilterAfter(...)` (lines 12-13), so that the filter is only applied to URLs matching the pattern "`/api/**`". Unfortunately, such implicit constraints are not well documented.

Challenge 3: Converting from XML-based to Java-based configurations is tedious and error-prone. The semantic conflicts between annotations, deployment descriptors, and code implementations are always hard to locate and resolve. Such problems become more serious when developers express security in a hybrid way of Java-based and XML-based. Since Spring Security 3.2, developers are supported to configure SS in a pure Java-based approach, and there is documentation describing how to migrate from XML-based to Java-based configurations [57]. However, manually applying tons of the migration rules is still time-consuming and error-prone.

Finding 3: *Spring Security authentication posts were mainly about configuring security for various enterprise applications in different ways (Java-based or XML-based), and converting between them. The challenges were due to incomplete documentation, as well as missing tool support for automatic configuration checking and converting.*

4.2.2 Cryptography. 45 of the 64 posts were about key generation and usage. For instance, some posts discussed how to create a key from scratch [38], and how to generate or retrieve a key from a random number [26], a byte array [12], a string [30], a certificate [17], BigIntegers [5], a keystore [4], or a file [67]. Some other posts focused on how to compare keys [9], print key information [66], or initialize a cipher for encryption and decryption [39].

Specifically, we observed three common challenges of correctly using the cryptography APIs.

Challenge 1: The error messages did not provide sufficient useful hints about fixes. We found five posts concentrated on the same problem: "get InvalidKeyException: Illegal key size", while the solutions were almost identical: (1) download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files", "local_policy.jar", and "US_export_policy.jar"; and (2) place the policy files in proper folders [1]. Developers got the same exception because of missing either of the two steps. Providing a checklist of these necessary steps in the error message could help developers quickly resolve the problem. However, the existing error messages did not provide any constructive suggestion.

Challenge 2: It is difficult to implement security with multiple programming languages. Three posts were about encrypting data with one language (e.g. PHP or Python) and decrypting data with another language (e.g., Java). Such cross-language data encryption & decryption is challenging, because the format of the generated data by one language requires special handling in another language. Listing 2 is an example to generate an RSA key pair and encrypt data in PHP, and to decrypt data in Java [16].

Listing 2: Encryption in PHP and decryption in Java [16]

```
1  // *****keypair.php *****
2  if(file_exists('private.key')) {
3      echo file_get_contents('private.key'); }
4  else {
5      include('Crypt/RSA.php');
6      $rsa = new Crypt_RSA();
7      $res = $rsa->createKey();
8      $privateKey = $res['privatekey'];
9      $publicKey  = $res['publickey'];
10     file_put_contents('public.key', $publicKey);
11     file_put_contents('private.key', $privateKey); }
12 // *****encrypt.php *****
13 include('Crypt/RSA.php');
14 $rsa = new Crypt_RSA();
15 $rsa->setEncryptionMode(CRYPT_RSA_ENCRYPTION_OAEP);
16 $rsa->loadKey(file_get_contents('public.key'));
17 // *****MainClass.java *****
18 BASE64Decoder decoder=new BASE64Decoder();
19 String b64PrivateKey=getContents(
20   "http://localhost/api/keypair.php").trim();
21 byte[] decodedKey=decoder.decodeBuffer(b64PrivateKey);
22 BufferedReader br=new BufferedReader(
23   new StringReader(new String(decodedKey)));
24 PEMReader pr=new PEMReader(br);
25 KeyPair kp=(KeyPair)pr.readObject();
26 pr.close();
27 PrivateKey privateKey=kp.getPrivate();
28 Cipher cipher=Cipher.getInstance(
29   "RSA/None/OAEPWithSHA1AndMGF1Padding","BC");
30 cipher.init(Cipher.DECRYPT_MODE, privateKey);
31 byte[] plaintext=cipher.doFinal(cipher);
```

In this example, when a key pair is generated in PHP (lines 2-11), the public key is easy to retrieve in PHP (lines 13-16). However, retrieving the private key in Java is more complicated (lines 18-30). After reading in the private key string (lines 19-20), the Java implementation first uses `Base64Decoder` to decode the string into a byte array (line 21), which corresponds to an OpenSSL PEM encoded stream (line 22-23). Because OpenSSL PEM is not a standard data format, the Java code further uses a PEMReader to convert the stream to a PrivateKey instance (lines 24-27) before using the key to initialize a cipher (lines 28-30). Existing documentation seldom describes how the security data format (e.g., key) defined in one language corresponds to that of another language. Unless developers are experts in both languages, it is hard for them to figure out the security data processing across languages.

Challenge 3: Implicit constraints on API usage cause confusion. Two posts were about getting "InvalidKeySpecException: algid parse error, not a sequence", when obtaining a private key from a file [29]. The problem is that the key should be in PKCS#8 format when used to create a `PKCS8EncodedKeySpec` instance, as shown below:

Listing 3: Consistency between the key format and spec [29]

```
//privKey should be in PKCS#8 format
byte[] privKey=...;
PKCS8EncodedKeySpec keySpec=
  new PKCS8EncodedKeySpec(privKey);
```

The tricky part here is that a private key retrieved from a file always has the data type `byte[]` even if it is not in PKCS#8 format. If developers invoke the API `PKCS8EncodedKeySpec(...)` with a non-PKCS#8 formatted key, they will be stuck with the clueless exception. Three solutions were suggested to get a PKCS#8 format key: (1) to implement code to convert the byte array, (2) to use an OpenSSL command to convert the file format, or (3) to use the `PEMReader` class of BouncyCastle to generate a key from the file. Such implicit constraints between an API and its input format are delicate.

Finding 4: *The cryptography posts were majorly about key generation and usage. Developers asked these questions mainly due to clueless error messages, cross-language data handling, and implicit API usage constraints.*

4.2.3 *Java EE security.*

33 of the 58 posts were on authentication and authorization. However, the APIs of these two security features were defined differently on different application servers (e.g., WildFly and Glassfish), and developers might use these servers in combination with diverse third-party libraries [49]. As a result, the posts seldom shared common solutions or code implementation.

One common challenge we identified is the usage of declarative security and programmatic security. When developers misunderstood annotations, they could use incorrect annotations that conflict with other annotations [35], deployment descriptors [72], code implementation [11], or file paths [48]. Nevertheless, existing error reporting systems only throw exceptions. There is no tool helping developers identify or resolve conflicting configurations.

Finding 5: *Java EE security posts were mainly about authentication and authorization. One challenge is the complex usage of declarative security and programmatic security, and any complicated interaction between the two.*

4.2.4 Access Control. 43 posts mainly discussed how to *restrict* or *relax* the access permission(s) of a software application for certain resource(s).

Specifically, 21 questions asked about restricting untrusted code from accessing certain packages [40], classes [42], or class members (i.e., methods and fields) [20]. Two alternative solutions were commonly suggested for these questions: (1) to override the `checkXXX()` methods of `SecurityManager` to disallow invalid accesses, or (2) to define a custom policy file to grant limited permissions. Another nine posts were on how to allow applets to perform privileged operations [53], because applets are executed in a security sandbox by default and can only perform a set of safe operations. One commonly recommended solution was to digitally sign the applet. Although it seems that there exist common solutions to the most frequently asked questions, the access control implementation is not always intuitive. We identified two common challenges of correctly implementing access control.

Challenge 1: The effect of access control varies with the program context. We identified two typical scenarios from the posts. First, the RMI tutorial [28] suggested that a security manager is needed *only* when RMI code downloads code from a remote machine. Including a `SecurityManager` instance in the RMI program which does not download any code can cause an AccessControlException [37]. Second, although a signed applet is allowed to perform sensitive operations, it loses its privileges when being invoked from Javascript [21]. As a result, the invocation to the signed applet should be wrapped with an invocation of `AccessController.doPrivileged(...)`.

Challenge 2: The effect of access control varies with the execution environment. `SecurityManager` can disallow illegal accesses via reflection only when the program is executed in a controlled environment (i.e., on a trusted server) [7]. Nevertheless, if the program is executed in an uncontrolled environment (e.g. on an untrusted client machine) and hackers can control how to run the program or manipulate the jar file, the security mechanisms become voided.

Finding 6: *The access control posts were mainly about* `SecurityManager`*,* `AccessController`*, and the policy file. Configuring and customizing access control policies are challenging.*

4.2.5 Secure Communication. Among the 31 examined posts, 22 posts were about SSL/TLS-related issues, discussing how to create [60], install [64], find [43], or validate an SSL certificate [62], how to establish a secure connection [36], and how to use SSL together with other libraries, such as JNDI [23] and PowerMock [70].

In particular, six posts focused on the problem of unable to find a valid server certificate to establish an SSL connection with a server [43]. Instead of advising to install the required certificates, two accepted answers suggested a highly insecure workaround to disable the SSL verification process, so that any incoming certificate can pass the validation [61]. Although such workarounds can effectively remove the error, they essentially fail the requirement to secure communication with SSL. In Section 4.3, we will further explain the security vulnerability due to such workarounds. Developers likely accepted the vulnerable answers because they found it challenging to implement the whole process of creating, installing, finding, and validating an SSL certificate.

Finding 7: *Security communication posts mainly discussed the process of establishing SSL/TLS connections. This process contains so many steps that developers were tempted to accept a broken solution to simply bypass the security check.*

4.3 Common Security Vulnerabilities

Among the five categories listed in Section 4.2, we identified security vulnerabilities in the accepted answers of three frequently discussed topics: Spring Security's `csrf()`, SSL/TLS, and password hashing.

4.3.1 Spring Security's `csrf()`*.* **Cross-site request forgery (CSRF)** is a serious attack that tricks a web browser into executing an unwanted action (e.g., transfer money to another account) in a web application (e.g., a bank website) for which a user is authenticated [106]. The root cause is that attackers created forged requests and mixed them with legitimate ones. Since the application cannot distinguish between the two types of requests, it normally responds to the forged requests, performing undesired operations.

By default, Spring Security provides CSRF protection by defining a function `csrf()` and implicitly enabling the function invocation. Correspondingly, developers should include the CSRF token in all PATCH, POST, PUT, and DELETE methods to leverage the protection [31]. However, among the 12 examined posts that were relevant to `csrf()`, 5 posts discussed program failures, while all the accepted answers suggested an insecure solution: disabling the CSRF protection by invoking `http.csrf().disable()`. In one instance, after accepting the vulnerable solution, an asker commented "*Adding csrf().disable() solved the issue!!! I have no idea why it was enabled by default*" [45]. Unfortunately, the developer happily disabled the security protection without realizing that such workaround would expose the resulting software to CSRF security exploits.

Finding 8: *In 5 of the 12* `csrf()`*-relevant posts, developers took the suggestion to irresponsibly disable the default CSRF protection. Developers were unaware of the potential vulnerabilities resulting from use of such insecure code.*

4.3.2 SSL/TLS. We examined 10 posts discussing the usage of SSL/TLS, and observed two important security issues.

Problem 1: Many developers opted to trust all SSL certificates and permit all hostnames with the intent of quickly building a prototype in the development environment. SSL is the standard security technology for establishing an encrypted connection between a web server and browser. Figure 5 details the major steps of establishing an SSL connection [50]. To activate SSL on the server, developers need to provide all identity information of the website (e.g., the host name) to a Certification Authority (CA), and request for an SSL certificate (Step ①). After validating the website's information, the CA issues a digitally signed SSL certificate (Step ②). When a client or browser attempts to connect to the website (Step ③), the server sends over its certificate (Step ④). The client then conducts several checks, including (1) whether the certificate is issued by a CA the browser trusts, and (2) whether the requested hostname matches the hostname associated with the certificate (Step ⑤). If all these checks are passed, the SSL connection can be established successfully.

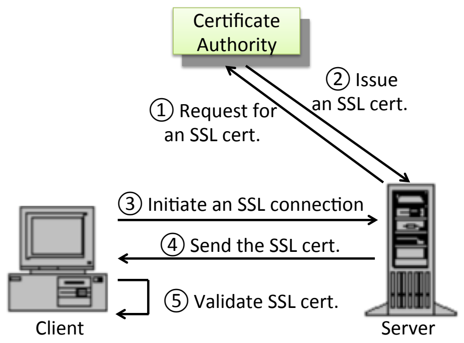


Figure 5: Simplified overview of creating an SSL connection

Though safest practice is to enable SSL only after obtaining a signed certificate from a CA, many developers implement and test certificate verification code before obtaining one. A common workaround without CA-signed certificates is to create a local *self-signed* certificate for use in implementing certificate verification [60]. However, 9 of the 10 examined posts accepted an insecure solution to bypass security checks entirely by trusting *all* certificates and/or allowing *all* hostnames, as demonstrated by Listing 4.

Listing 4: A typical implementation to disable SSL certificate validation [52]

```
// Create a trust manager that does not validate certificate chains
TrustManager[] trustAllCerts = new TrustManager[]{
  new X509TrustManager() {
     public java.security.cert.X509Certificate[]
         getAcceptedIssuers() {return null;}
     public void checkClientTrusted(...) {}
     public void checkServerTrusted(...) {} }};
// Install the all-trusting trust manager
try {
  SSLContext sc = SSLContext.getInstance("SSL");
  sc.init(null, trustAllCerts,
      new java.security.SecureRandom());
  HttpsURLConnection.setDefaultSSLSocketFactory(
      sc.getSocketFactory());
} catch (Exception e) {}
// Access an https URL without any certificate
try {
  URL url=new URL("https://hostname/index.html");
} catch (MalformedURLException e) {}
```

Disabling the SSL certificate validation process completely invalidates the secure communication protocol, leaving clients susceptible to **man-in-the-middle (MITM) attacks** [85]; Namely, by secretly relaying and possibly altering communication (e.g., through DNS poisoning) between client and server, attackers can fool the SSL-client to instead connect to an attacker-controlled server [85]. Although the insecurity of this coding practice was highlighted in 2012 [85], three examined posts created since then still discussed the bad practice [8, 33, 61]. This indicates a significant gap between security theory and coding practices. Some developers justified their verification-bypassing logic by saying "I want my client to accept any certificate (because I'm only ever pointing to one server)", or "*Because I needed a quick solution for debugging purposes only. I would not use this in production due to the security concerns ...*" [65]. However, as pointed by another SO user [65] and demonstrated by prior research [82, 85], ***many of these implementations find their way into production software, and have yielded radically insecure systems as a result***.

Problem 2: Developers were unaware of the best usage of SSL/TLS. TLS is SSL's successor. It is so different from SSL that the two protocols do not interoperate. To maintain the backward compatibility with SSL 3.0 and interoperate with systems supporting SSL, most SSL/TLS implementations allow for protocol version negotiation: if a client and a server cannot connect via TLS, they will fall back to using the older protocol SSL 3.0. In 2014, Möller et al. reported the **POODLE attack** which exploits the SSL 3.0 fallback [93]. Specifically, there is a design vulnerability in the way SSL 3.0 handles block cipher mode padding, which can be exploited by attackers to decrypt encrypted messages. With the POODLE attack, a hacker can intentionally trigger a TLS connection failure and force usage of SSL 3.0, allowing decryption of encrypted messages.

Ever since 2014, researchers have recommended developers to disable SSL 3.0 support and configure systems to present the SSL 3.0 fallback. The US government (NIST) mandates ceasing SSL usage in the protection of Federal information [19]. Nevertheless in reality, none of the 10 posts mentioned the POODLE attack. The single post we examined created in 2016 [61] relied on SSL.

Finding 9: *9 of 10 SSL/TLS-relevant posts discussed insecure code to bypass security checks. We observed two important security threats: (1) StackOverflow contains a lot of obsolete and insecure coding practices; and (2) secure programmers are unaware of the state-of-the-art security knowledge.*

4.3.3 Password Hashing. We found 6 posts related to hashing passwords with MD5 or SHA-1 to store user credentials in databases. However, these hashing functions were found insecure [102, 104]. They are vulnerable to offline **dictionary attacks** [15]; After obtaining a password hash H from a compromised database, a hacker can use brute-force methods to enumerate a list of password guesses, until finding the password P whose hash value matches H. Impersonating a valid user at login allows an attacker to conduct malicious behavior. Researchers recommended **key-stretching algorithms** (e.g., PBKDF2, bcrypt, and scrypt) as the best practice for secure password hashing, as these algorithms are specially crafted to slow down hash computation by orders of magnitude [75, 84, 101], which substantially increases the difficulty of dictionary attacks.

Unfortunately, only 3 of the 6 posts (50%) mentioned the best practice in their accepted answers, indicating that many posts on secure hashing recommended insecure hash functions. We found one post which asked about using MD5 hashing in Android [46]. Although subsequent discussion between developers revealed some recommendations of avoiding MD5, the asker kept justifying his/her choice of MD5. The asker even shared a completely wrong understanding of secure hashing: "*The security of hash algorithms **really** is MD5 (strongest) > SHA-1 > SHA-256 > SHA-512 (weakest)*", although the opposite is true, which is MD5 < SHA-1 < SHA-256 < SHA-512.

Among these posts, some developers misunderstood security APIs and ignored the potential consequences of their API choices. Such posts can have profound negative impact, because they may mislead people by conveying incorrect information on an otherwise popular website.

Finding 10: *3 of 6 hashing-relevant posts accepted vulnerable solutions as correct answers, indicating that developers were unaware of best secure programming practices. Incorrect security information may propagate among StackOverflow users and negatively influence software development.*

4.3.4 Potential social impacts of insecure coding practices. Among the 17 SO posts that either discussed or recommended insecure coding practices relevant to CSRF, SSL/TLS, and password hashing, we observed two phenomena. First, the total view count of these posts is 622,922. Such a large viewcount means many developers have read these posts, while some have perhaps already heeded their erroneous advice and incorporated vulnerable code in their own projects. Second, the influential answers are not necessarily secure. In one post [24], the insecure suggestion by a user with higher reputation (i.e., 55.6K reputation score) was selected as the accepted answer, as opposed to the correct fix by a user with lower reputation (29K). In another post [52], one insecure "quick fix" answer received 5 votes probably because it indeed eliminated the error messages. The positive indicators for insecure solutions (e.g., high reputation and positive votes) can mislead developers to implement insecure practices.

Finding 11: *Highly viewed posts may influentially promote insecure coding practices. This problem may be further aggravated by misleading indicators such as accepted answers, answers' positive votes, and responders' high reputation.*

5 RELATED WORK

This section describes related work on analyzing, detecting, and preventing security vulnerabilities due to library API misuse.

5.1 Analyzing Security Vulnerabilities

Prior studies showed API misuse caused many security vulnerabilities [88, 90, 103, 105]. For instance, Long identified several Java features (e.g., the reflection API) whose misuse or improper implementation can compromise security [90]. Lazar et al. manually examined 269 published cryptographic vulnerabilities in the CVE database, and observed 83% of them were caused by the misuse of cryptographic libraries [88]. Veracode reported that 39% of all applications used broken or risky cryptographic algorithms [103].

Barua et al. automatically extracted latent topics in SO posts [74], but not specific to security. Nadi et al. reported the obstacles of using cryptography APIs by examining 100 SO posts and 48 developers' survey inputs [94]. Acar et al. focused on the vulnerabilities in Android code [73]. The studies by Yang et al. [105] and Rahman [97] are the most relevant to our research. They automatically extracted security-relevant topics from SO questions, and identified high-frequency keywords like "Password" and "Hash" for post categorization. However, neither study focused on developers' programming challenges and security vulnerabilities in posts as we did. Our SE and security findings have more technical depth.

5.2 Detecting Security Vulnerabilities

Approaches were built to detect security vulnerabilities caused by API misuse [77, 80, 82, 83, 85, 87, 89, 96]. For instance, Egele et al. implemented a static checker for six well defined Android cryptographic API usage rules, such as "Do not use ECB mode for encryption", and analyzed 11,748 Android applications for any rule violation [80]. They found 88% of the applications violated at least one checked rule. Fischer et al. extracted Android security-related code snippets from SO, and manually labeled a subset of the data as "secure" or "insecure" [83]. The labeled data allowed them to train a classifier and efficiently judge whether a code snippet is secure or not for the whole data set. Next, they searched for code clones of the snippets in 1.3 million Android apps, and found many clones of the insecure code. Fahl et al. [82] and Georgiev et al. [85] separately implemented an attack model: man-in-the-middle attack, and detected vulnerable Android applications and software libraries which misused SSL APIs. Both research groups observed that developers disabled certification validation for testing with self-signed and/or trusted certificates. He et al. developed SSLINT, an automatic static analysis tool, to identify the misuse of SSL/TLS APIs in client-side applications [87].

Compared with prior research, our study has two new contributions. First, *our scope is broader.* We report new challenges on secure coding practices, such as complex security configurations in Spring Security and developers' outdated security knowledge. Second, *our investigation of an online forum provides a new social perspective about secure coding.* These unique insights cannot be discovered through analyzing code.

5.3 Preventing Security Vulnerabilities

Researchers proposed approaches to prevent developers from implementing vulnerable code and misusing APIs [78, 79, 81, 91, 92, 100]. For example, Mettler et al. designed Joe-E—a security-oriented subset of Java—to support secure coding by removing any encapsulation-breaking features from Java (e.g., reflection), and by enforcing the least privilege principle [91]. Keyczar is a library designed to simplify the cryptography usage, and thus to prevent API misuse [79]. Below shows how to decrypt data with Keyczar:

Listing 5: Simple decryption with Keyczar APIs

```
Crypter crypter=new Crypter("/rsakeys");
String plaintext=crypter.decrypt(ciphertext);
```

Compared with the decryption code shown in Listing 2 (lines 18-31), this implementation is much simpler and more intuitive. All details about data format conversion and cipher initialization are hidden, while a default strong block cipher is used to properly decrypt data.

Some approaches were developed to apply formal verification techniques and analyze the security properties of cryptographic protocol specifications [78, 92] and cryptographic API implementations [81, 100]. For instance, Protocol Composition Logic (PCL) is a logic for proving security properties, like network protocols that use public and symmetric key cryptography [78]. The logic is designed around a process calculus with actions for possible protocol steps, including generating new random numbers and sending and receiving messages The proof system consists of axioms about individual protocol actions, and inference rules that yield assertions about protocols composed of multiple steps.

6 OUR RECOMMENDATIONS

By analyzing the SO posts relevant to Java security from both software engineering and security perspectives, we observed the gap between the intended usage of APIs and the actual problematic API usage by developers, sensed developers' frustration when they spent tremendous effort identifying correct API usage (e.g., two weeks as mentioned in [55]), and observed potential security consequences from library misuses. Below are our recommendations based on the analysis.

For Security Developers. Conduct security testing to check whether the implemented features work as expected. Do not disable security checks (e.g., CSRF check) to implement a temporary fix in the testing or development environment. Be cautious when following SO accepted or reputable answers to implement secure code, because these solutions may be unsafe and outdated. For SO administrators, we recommend them to specially handle the posts with vulnerable code, because these posts may influentially mislead security developers.

For Library Designers. Remove or deprecate the APIs whose security guarantees are broken (e.g., MD5). Design clean and helpful error reporting interfaces which show not only the error, but also possible root causes and solutions. Design simplified APIs with strong security defenses implemented by default.

For Tool Builders. Develop automatic tools to diagnose security errors, locate buggy code, and suggest security patches or solutions. Build vulnerability prevention techniques, which compare peer applications that use the same set of APIs to infer and warn potential misuses. Explore approaches that check and enforce the semantic consistency between security-relevant annotations, code, and configurations. Build new approaches to transform between the implementations of declarative security and programmatic security.

7 THREATS TO VALIDITY

This study is mainly based on our manual inspection of Java security-relevant posts, so the observations may be subject to human bias. To alleviate the problem, the first author of the paper conducted multiple careful inspections of all posts relevant to implementation questions, while the second author also examined the posts related to security vulnerabilities (mentioned in Section 4.3) multiple times.

To remove posts without code snippets, we defined a filter to search for keywords "public" and "class". If a post does not contain both words, the filter automatically removes the post from our data set. This filter may incorrectly remove some relevant posts that contain code. In the future, we will improve our crawling technique to keep the <code> tags around code snippets in the raw data, and then rely on these tags to filter posts more precisely. We will also leverage Cerulo et al.'s approach [76] to automatically extract source code from free text.

We conservatively mentioned posts whose accepted answers *will* cause security vulnerabilities, although there might be more accepted answers that suffer from known security attacks. Due to the limited available program and environment information in each post, and our limited knowledge about frameworks and potential security attacks, we decided not to mention suspicious posts whose accepted answers *might* lead to security vulnerabilities.

8 CONCLUSION

Our work aimed at assessing the current secure coding practices, and identifying potential gaps between theory and practice, and between specification and implementation. Our analysis of hundreds of posts on the popular developer forum (StackOverflow) revealed a worrisome reality in the software development industry.

- A substantial number of developers do not appear to understand the security implications of coding options, showing a lack of cybersecurity training. This situation creates frustration in developers, who sometimes end up choosing completely insecure-but-easy fixes. Examples of such easy fixes include using obsolete cryptographic hash functions, disabling CSRF protection, trusting all certificates to enable SSL/TLS, or using obsolete communication protocols. These poor coding practices, if used in production code, will seriously compromise the security of software products.
- We provided substantial empirical evidence showing that (1) Spring Security usage is overly complicated and poorly documented; (2) the error reporting systems of Java platform security APIs cause confusion; and (3) the multi-language support for securing data is rather weak. These issues can seriously hinder developers' productivity, resulting in great frustration and confusion.
- Interestingly, we found that the social dynamics among askers and responders can impact people's security choices. Highly viewed posts may wrongly promote vulnerable code. Metadata like accepted answers, responders' reputation scores, and answers' positive vote counts can further mislead developers to take insecure advices.
- Developers' security concerns have shifted from cryptography APIs to Spring Security over time, although researchers have not provided scientific solutions to resolve the vulnerabilities in such new context.

We described several possible solutions to improve secure coding practices in the paper. However, efforts (e.g., workforce retraining) to correct these alarming security issues may take a while to take effect. Our future work is on building automatic or semi-automatic security bug detection and fixing tools.

ACKNOWLEDGMENT

We thank anonymous reviewers for their thorough comments on our earlier version of the paper. This work was supported by NSF Grant No. CCF-1565827 and ONR-450487.

REFERENCES

[1] 2017. AES-256 implementation in GAE. https://stackoverflow.com/questions/12833826/aes-256-implementation-in-gae. (2017).

[2] 2017. Apache Shiro Documentation. https://shiro.apache.org/documentation.html. (2017).

[3] 2017. Application Server - Oracle WebLogic Server. https://www.oracle.com/middleware/weblogic/index.html. (2017).

[4] 2017. Basic Program for encrypt/Decrypt : javax.crypto.BadPaddingException: Decryption error. https://stackoverflow.com/questions/39518979/basic-program-for-encrypt-decrypt-javax-crypto-badpaddingexception-decryption. (2017).

[5] 2017. BigInteger to Key. https://stackoverflow.com/questions/10271164/biginteger-to-key. (2017).

[6] 2017. Bouncy Castle. https://www.bouncycastle.org. (2017).

[7] 2017. Can a secret be hidden in a 'safe' java class offering access credentials? https://stackoverflow.com/questions/5761519/can-a-secret-be-hidden-in-a-safe-java-class-offering-access-credentials. (2017).

[8] 2017. Communication with server that support ssl in java. (2017). https://stackoverflow.com/questions/21156929/java-class-to-trust-all-for-sending-file-to-https-web-service

[9] 2017. Compare two Public Key values in java [duplicate]. https://stackoverflow.com/questions/37439695/compare-two-public-key-values-in-java. (2017).

[10] 2017. Configure Spring Security without XML in Spring 4. https://stackoverflow.com/questions/20961600/configure-spring-security-without-xml-in-spring-4. (2017).

[11] 2017. @Context injection in Stateless EJB used by JAX-RS. https://stackoverflow.com/questions/29132547/context-injection-in-stateless-ejb-used-by-jax-rs. (2017).

[12] 2017. Converted secret key into bytes, how to convert it back to secrect key? https://stackoverflow.com/questions/5364338/converted-secret-key-into-bytes-how-to-convert-it-back-to-secrect-key. (2017).

[13] 2017. Custom Authentication Filters in multiple HttpSecurity objects using Java Config. https://stackoverflow.com/questions/37304211/custom-authentication-filters-in-multiple-httpsecurity-objects-using-java-config. (2017).

[14] 2017. CWE-227: Improper Fulfillment of API Contract ('API Abuse'). https://cwe.mitre.org/data/definitions/227.html. (2017).

[15] 2017. Dictionary Attacks 101. https://blog.codinghorror.com/dictionary-attacks-101/. (2017).

[16] 2017. Encryption PHP, Decryption Java. https://stackoverflow.com/questions/15639442/encryption-php-decryption-java. (2017).

[17] 2017. Get public and private key from ASN1 encrypted pem certificate. https://stackoverflow.com/questions/30392114/get-public-and-private-key-from-asn1-encrypted-pem-certificate. (2017).

[18] 2017. GlassFish. https://javaee.github.io/glassfish/. (2017).

[19] 2017. Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf. (2017).

[20] 2017. Hiding my security key from java reflection. https://stackoverflow.com/questions/14903318/hiding-my-security-key-from-java-reflection. (2017).

[21] 2017. How can I get a signed Java Applet to perform privileged operations when called from unsigned Javascript? https://stackoverflow.com/questions/1006674/how-can-i-get-a-signed-java-applet-to-perform-privileged-operations-when-called. (2017).

[22] 2017. How does Java string being immutable increase security? https://stackoverflow.com/questions/15274874/how-does-java-string-being-immutable-increase-security. (2017).

[23] 2017. how to accept self-signed certificates for JNDI/LDAP connections? https://stackoverflow.com/questions/4615163/how-to-accept-self-signed-certificates-for-jndi-ldap-connections. (2017).

[24] 2017. How to add MD5 or SHA hash to spring security? https://stackoverflow.com/questions/18581463/how-to-add-md5-or-sha-hash-to-spring-security. (2017).

[25] 2017. How to apply spring security filter only on secured endpoints? https://stackoverflow.com/questions/36795894/how-to-apply-spring-security-filter-only-on-secured-endpoints. (2017).

[26] 2017. How to generate secret key using SecureRandom.getInstanceStrong()? https://stackoverflow.com/questions/37244064/how-to-generate-secret-key-using-securerandom-getinstancestrong. (2017).

[27] 2017. How to override Spring Security default configuration in Spring Boot. https://stackoverflow.com/questions/35600488/how-to-override-spring-security-default-configuration-in-spring-boot. (2017).

[28] 2017. Implementing a Remote Interface. http://docs.oracle.com/javase/tutorial/rmi/implementing.html. (2017).

[29] 2017. InvalidKeySpecException : algid parse error, not a sequence. https://stackoverflow.com/questions/31941413/invalidkeyspecexception-algid-parse-error-not-a-sequence. (2017).

[30] 2017. java - Edit code sample to specify DES key value. https://stackoverflow.com/questions/22858497/edit-code-sample-to-specify-des-key-value. (2017).

[31] 2017. java - Simple example of Spring Security with Thymeleaf. https://stackoverflow.com/questions/25692735/simple-example-of-spring-security-with-thymeleaf. (2017).

[32] 2017. Java Authentication and Authorization Service (JAAS) Reference Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html. (2017).

[33] 2017. java class to trust all for sending file to https web service. https://stackoverflow.com/questions/21156929/java-class-to-trust-all-for-sending-file-to-https-web-service. (2017).

[34] 2017. Java Cryptography Architecture. http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html. (2017).

[35] 2017. Java EE 7 EJB Security not working. https://stackoverflow.com/questions/30504131/java-ee-7-ejb-security-not-working. (2017).

[36] 2017. Java Mail get mails with pop3 from exchange server => Exception in thread "main" javax.mail.MessagingException. https://stackoverflow.com/questions/25017050/java-mail-get-mails-with-pop3-from-exchange-server-exception-in-thread-main. (2017).

[37] 2017. Java RMI / access denied. https://stackoverflow.com/questions/36570012/java-rmi-access-denied. (2017).

[38] 2017. Java Security - RSA Public Key & Private Key Code Issue. https://stackoverflow.com/questions/18757114/java-security-rsa-public-key-private-key-code-issue. (2017).

[39] 2017. Java security init Cipher from SecretKeySpec properly. https://stackoverflow.com/questions/14230096/java-security-init-cipher-from-secretkeyspec-properly. (2017).

[40] 2017. Java Security Manager completely disable reflection. https://stackoverflow.com/questions/40218973/java-security-manager-completely-disable-reflection. (2017).

[41] 2017. Java Security Overview. http://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html. (2017).

[42] 2017. Java security: Sandboxing plugins loaded via URLClassLoader. https://stackoverflow.com/questions/3947558/java-security-sandboxing-plugins-loaded-via-urlclassloader. (2017).

[43] 2017. Java SSL - InstallCert recognizes certificate, but still "unable to find valid certification path" error? https://stackoverflow.com/questions/11087121/java-ssl-installcert-recognizes-certificate-but-still-unable-to-find-valid-c. (2017).

[44] 2017. JSR-000366 Java Platform, Enterprise Edition 8 Public Review Specification. http://download.oracle.com/otndocs/jcp/java_ee-8-pr-spec/. (2017).

[45] 2017. logout call - Spring security logout call. https://stackoverflow.com/questions/24530603/spring-security-logout-call. (2017).

[46] 2017. MD5 hashing in Android. https://stackoverflow.com/questions/4846484/md5-hashing-in-android. (2017).

[47] 2017. OWASP. https://www.owasp.org/index.php/Main_Page. (2017).

[48] 2017. PicketLink / Deltaspike security does not work in SOAP (JAX-WS) layer (CDI vs EJB?). https://stackoverflow.com/questions/32392702/picketlink-deltaspike-security-does-not-work-in-soap-jax-ws-layer-cdi-vs-ej. (2017).

[49] 2017. Resteasy Authorization design - check a user owns a resource. https://stackoverflow.com/questions/34315838/resteasy-authorization-design-check-a-user-owns-a-resource. (2017).

[50] 2017. RF 6101 - The Secure Sockets Layer (SSL) Protocol Version 3.0. https://tools.ietf.org/html/rfc6101. (2017).

[51] 2017. Scrapy | A Fast and Powerful Scraping and Web Crawling Framework. https://scrapy.org. (2017).

[52] 2017. security - Allowing Java to use an untrusted certificate for SSL/HTTPS connection. https://stackoverflow.com/questions/1201048/allowing-java-to-use-an-untrusted-certificate-for-ssl-https-connection. (2017).

[53] 2017. security exception when loading web image in jar. https://stackoverflow.com/questions/2011407/security-exception-when-loading-web-image-in-jar. (2017).

[54] 2017. Spring Security. https://projects.spring.io/spring-security/. (2017).

[55] 2017. Spring Security 4 xml configuration UserDetailsService authentication not working. https://stackoverflow.com/questions/41321176/spring-security-4-xml-configuration-userdetailsservice-authentication-not-workin. (2017).

[56] 2017. Spring security JDK based proxy issue while using @Secured annotation on Controller method. https://stackoverflow.com/questions/35860442/spring-security-jdk-based-proxy-issue-while-using-secured-annotation-on-control. (2017).

[57] 2017. Spring Security Reference. http://docs.spring.io/spring-security/site/docs/3.2.4.RELEASE/reference/htmlsingle/#jc-httpsecurity. (2017).

[58] 2017. Spring Security Tutorial. http://www.mkyong.com/tutorials/spring-security-tutorials/. (2017).

[59] 2017. Spring Security using JBoss <security-domain>. https://stackoverflow.com/questions/28172056/spring-security-using-jboss-security-domain. (2017).

[60] 2017. SSL Certificate Verification : javax.net.ssl.SSLHandshakeException. https://stackoverflow.com/questions/25079751/ssl-certificate-verification-javax-net-ssl-sslhandshakeexception. (2017).

[61] 2017. Ssl handshake fails with unable to find valid certification path to requested target. https://stackoverflow.com/questions/40977556/ssl-handshake-fails-with-unable-to-find-valid-certification-path-to-requested-ta. (2017).

[62] 2017. SSL Socket Connection working even though client is not sending certificate? https://stackoverflow.com/questions/26761966/ssl-socket-connection-working-even-though-client-is-not-sending-certificate. (2017).

[63] 2017. StackOverflow. https://stackoverflow.com. (2017).

[64] 2017. The Webserver I talk to updated its SSL cert and now my app can't talk to it. https://stackoverflow.com/questions/5758812/the-webserver-i-talk-to-updated-its-ssl-cert-and-now-my-app-cant-talk-to-it. (2017).

[65] 2017. Trusting all certificates using HttpClient over HTTPS. https://stackoverflow.com/questions/2642777/trusting-all-certificates-using-httpclient-over-https. (2017).

[66] 2017. Use of ECC in Java SE 1.7. https://stackoverflow.com/questions/24383637/use-of-ecc-in-java-se-1-7. (2017).

[67] 2017. Using public key from authorized_keys with Java security. https://stackoverflow.com/questions/3531506/using-public-key-from-authorized-keys-with-java-security. (2017).

[68] 2017. Web Security Samples. https://github.com/spring-projects/spring-security-javaconfig/blob/master/samples-web.md#sample-multi-http-web-configuration. (2017).

[69] 2017. WebSphere Application Server - IBM. http://www-03.ibm.com/software/products/en/appserv-was. (2017).

[70] 2017. When a TrustManagerFactory is not a TrustManagerFactory (Java). https://stackoverflow.com/questions/14654639/when-a-trustmanagerfactory-is-not-a-trustmanagerfactory-java. (2017).

[71] 2017. WildFly. http://wildfly.org. (2017).

[72] 2017. Wildfly 9 security domains won't work. https://stackoverflow.com/questions/37425056/wildfly-9-security-domains-wont-work. (2017).

[73] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. 289–305. https://doi.org/10.1109/SP.2016.25

[74] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. 2014. What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empirical Software Engineering* 19, 3 (01 Jun 2014), 619–654. https://doi.org/10.1007/s10664-012-9231-y

[75] Sirapat Boonkrong. 2012. Security of passwords. *Information Technology Journal* 8, 2 (2012), 112–117.

[76] Luigi Cerulo, Massimiliano Di Penta, Alberto Bacchelli, Michele Ceccarelli, and Gerardo Canfora. 2015. Irish: A Hidden Markov Model to detect coded information islands in free text. *Science of Computer Programming* 105, Supplement C (2015), 26 – 43. https://doi.org/10.1016/j.scico.2014.11.017

[77] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2016. Evaluation of Cryptography Usage in Android Applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS) (BICT'15)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 83–90. https://doi.org/10.4108/eai.3-12-2015.2262471

[78] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. 2007. Protocol Composition Logic (PCL). *Electronic Notes in Theoretical Computer Science* 172 (2007), 311 – 358. https://doi.org/10.1016/j.entcs.2007.02.012 Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

[79] Arkajit Dey and Stephen Weis. 2017. *Keyczar: A Cryptographic Toolkit.*

[80] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS '13)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2508859.2516693

[81] Levent Erkök and John Matthews. 2008. Pragmatic Equivalence and Safety Checking in Cryptol. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification (PLPV '09)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/1481848.1481860

[82] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2382196.2382205

[83] Felix Fischer, Konstantin BÂĺottinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *38th IEEE Symposium on Security and Privacy (S&P '17)* (2017-05-22).

[84] Cory Gackenheimer. 2013. Implementing Security and Cryptography. In *Node. js Recipes*. Springer, 133–160.

[85] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2382196.2382204

[86] Li Gong and Gary Ellison. 2003. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation* (2nd ed.). Pearson Education.

[87] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang. 2015. Vetting SSL Usage in Applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy*. 519–534. https://doi.org/10.1109/SP.2015.38

[88] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys '14)*. ACM, New York, NY, USA, Article 7, 7 pages. https://doi.org/10.1145/2637166.2637237

[89] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2014. *iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications*. Springer International Publishing, Cham, 349–362. https://doi.org/10.1007/978-3-319-11698-3_27

[90] Fred Long. 2005. *Software Vulnerabilities in Java*. Technical Report CMU/SEI-2005-TN-044. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7573

[91] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed Systems Symposium*. Internet Society. http://www.truststc.org/pubs/652.html

[92] J. C. Mitchell, M. Mitchell, and U. Stern. 1997. Automated Analysis of Cryptographic Protocols Using Mur/Spl Phi/. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy (SP '97)*. IEEE Computer Society, Washington, DC, USA, 141–. http://dl.acm.org/citation.cfm?id=882493.884384

[93] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE bites: exploiting the SSL 3.0 fallback. *PDF online* (2014), 1–4.

[94] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[95] Scott Oaks. 1998. *Java Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[96] Lucky Onwuzurike and Emiliano De Cristofaro. 2015. Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '15)*. ACM, New York, NY, USA, Article 15, 6 pages. https://doi.org/10.1145/2766498.2766522

[97] Muhammad Sajidur Rahman. 2016. *An empirical case study on Stack Overflow to explore developers' security challenges*. Master's thesis. Kansas State University.

[98] Fahmida Y. Rashid. 2017. Library misuse exposes leading Java platforms to attack. http://www.infoworld.com/article/3003197/security/library-misuse-exposes-leading-java-platforms-to-attack.html. (2017).

[99] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. 2014. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *Proceedings of the 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC '14)*. IEEE Computer Society, Washington, DC, USA, 75–80. https://doi.org/10.1109/DASC.2014.22

[100] E. Smith and D. L. Dill. 2008. Automatic Formal Verification of Block Cipher Implementations. In *2008 Formal Methods in Computer-Aided Design*. 1–7. https://doi.org/10.1109/FMCAD.2008.ECP.10

[101] J Steven. 2017. Password storage cheat sheet. (2017). https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet

[102] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. *IACR Cryptology ePrint Archive* 2017 (2017), 190.

[103] Veracode. 2017. STATE OF SOFTWARE SECURITY. https://www.veracode.com/sites/default/files/Resources/Reports/state-of-software-security-volume-7-veracode-report.pdf. (2017).

[104] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. 2004. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *IACR Cryptology ePrint Archive* 2004 (2004), 199.

[105] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. *Journal of Computer Science and Technology* 31, 5 (01 Sep 2016), 910–924.

https://doi.org/10.1007/s11390-016-1672-0
[106] William Zeller and Edward W Felten. 2008. Cross-Site Request Forgeries: Exploitation and prevention. https://www.cs.utexas.edu/~shmat/courses/library/zeller.pdf. (2008).