Towards Best Secure Coding Practice for Implementing SSL/TLS

Mohannad Alhanahnah, Qiben Yan
Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE, USA (Email: yan@unl.edu)

***Abstract*—Developers often make mistakes while incorporating SSL/TLS functionality in their applications due to the complication in implementing SSL/TLS and their fast prototyping requirement. Insecure implementations of SSL/TLS are subject to different types of Man in The Middle (MiTM) attacks, which ultimately makes the communication between the two parties vulnerable to eavesdropping and hijacking attacks, thereby violating confidentiality and integrity of the exchanged information. This paper aims to support developers in detecting insecure SSL/TLS implementation in their codes by utilizing a *low-cost cross-language* static analysis tool called PMD. In the end, two insecure implementations of SSL/TLS have been identified, and subsequently a new PMD rule set is created. This rule set consists of three rules for addressing *hostname validation vulnerability* and *certificate validation vulnerability*. The rules have been evaluated over $1,517$ code snippets obtained from Stack Overflow, and the results show that 71% of the code snippets contain insecure SSL/TLS patterns. The detection rate of our approach is 100%, while it detects 165 violations inside the vulnerable code snippets in total.**

***Index Terms*—SSL/TLS, certificate validation, static analysis.**

I. Introduction

Developers, even experienced developers, are prone to make mistakes and not follow the best secure coding practices in implementing cryptographic APIs in their applications [1], [2]. According to a recent survey [3], 83% of the investigated 269 cryptography related vulnerabilities are caused due to the misuse of cryptographic APIs by the developers, which brings serious concerns on the security of the applications.

Secure Socket Layer (SSL) and Transport Layer Security (TLS, successor of SSL) are cryptographic mechanisms for securing the communications between two parties, and vulnerable implementations of these essential security protocols can be subject to Man in The Middle Attack (MiTM), which ultimately makes the communications between the two parties vulnerable to eavesdropping and hijacking attacks, thereby violating the confidentiality and integrity of the exchanged information. Developers also make mistakes while incorporating off-the-shelf SSL/TLS implmentations in their applications [4], [5]. Recently, Meng et al. [1] analyze the posts from Stack Overflow website, and find that developers have the tendency to integrate vulnerable answers related to SSL implementation in their applications, because SSL implementation is sometimes difficult to comprehend without sufficient security background, and/or they want to quickly build a prototype in the development environment to catch up with the stringent project deadline.

Moreover, even though bug finding tools such as PMD [6] define rules to detect weak implementations and potential bugs based on predefined rules, some of which check security properties, e.g., MethodReturnsInternalArray[1] and ArrayIsStoredDirectly[2], these tools including PMD do not contain rules for detecting weak/vulnerable SSL/TLS implementation patterns.

Two insecure SSL/TLS implementation patterns have been identified, including *hostname validation vulnerability* and *certificate validation vulnerability* [5]. Section IV provides more details about these vulnerable patterns. Meng et al. [1] further show that 81.8% of examined posts from Stack Overflow have endorsed an insecure solution to bypass security checks by trusting all certificates and/or allowing all hostnames, corresponding to the above two vulnerable patterns.

Addressing the insecure implementation of cryptographic mechanisms, particularly SSL/TLS implementation, is a pressing need. Apparently, allowing developers to detect insecure implementations in their codes can significantly reduce the number of insecure applications. Therefore, it is imperative to provide a tool for developers to produce more secure code, and thus avoid the potential security incidents such as Heartbleed [7] and being vulnerable to MiTM attacks.

This paper contributes towards establishing secure coding practice for developers by *developing practical and ready to use rule sets (consisting of three rules) using PMD to detect vulnerable SSL/TLS implementations.* These rules can accurately and efficiently identify potential SSL/TLS vulnerabilities, and help raise developers' awareness of insecure SSL/TLS implementation patterns.

II. Background

This section describes background knowledge about static analysis tools for detecting bugs, and introduces our criteria for selecting the tool to implement our detection rules. Static analysis and dynamic analysis solutions have been introduced for detecting vulnerability in the applications. Since this work aims at assisting developers in detecting insecure implementation of SSL/TLS, we will review and compare some state-of-the-art solutions proposed for detecting programming bugs using static analysis techniques.

[1]Exposing internal arrays directly allows the user to modify some code that could be critical.

[2]Constructors and methods receiving arrays should clone objects and store the copy.

Several open source static analysis tools are presented for detecting bugs in Java programs, including:

1) FindBugs [8]: is an open source tool for detecting bugs in Java code. It is a static analysis tool on Java bytecode, and can be used via command line and integrated into different IDEs. FindBugs can discover various types of bugs including problematic coding practice and vulnerabilities. FindBugs rules can be created using Visitor pattern (Java API). However, this tool does not detect insecure SSL/TLS implementation patterns.
2) Hammurapi [9]: is an open source tool for analyzing Java source code. It can be integrated to IDEs, and is developed with scalability in mind. Hammurapi employs Abstract Syntax Tree (AST), where new rules can be added to this tool, using java code or XML rules. However, this tool is rather complicated [10], and does not focus on detecting security vulnerabilities and insecure implementation patterns.
3) Jlint [11]: is written in C++ for detecting common programming errors in Java (e.g., race condition). Jlint performs semantic and syntax analysis on Java bytecode for accomplishing its duties. Therefore, Jlint is not intended for the check and validation of insecure SSL/TLS implementation, nor any kind of other security checks. Although new rules can be integrated into Jlint, it will require modifying Jlint's source code [10], which makes Jlint difficult to expand.
4) PMD: is an open source tool, which is written in Java and it checks Java source code for a set of predefined bugs. PMD can be used through command line, and graphical user interface via the available plugins for various IDEs. PMD constructs Abstract Syntax Tree (AST), and then examines the constructed AST for detecting bugs. As mentioned in Section I, PMD checks for some security bugs, but it neither checks insecure cryptographic mechanisms, nor examines SSL/TLS implementations. PMD rules can be defined using Java code (Visitor pattern) or XPath queries. This provides more flexibility and makes it easier for extension.

We define two selection criteria for identifying the optimal tool to develop our detection rules. The tool should be:

1) open source and is still actively supported by the community.
2) easy-to-use and facilitating the integration of new rules.

Accordingly, PMD has been selected for implementing our new rules to detect insecure SSL/TLS implementations, because PMD is an open source tool, and can be easily expanded with new rule sets. Unlike other tools that require changing the source codes of the tools, or are limited to a specific method for adding new rules, PMD is flexible, easy-to use, and deemed as a cross-architectural analysis tool, as it can analyze different programming languages.

III. Insecure SSL/TLS Patterns

This section explains the commonly identified SSL/TLS vulnerabilities, and describes the justification behind selecting those vulnerabilities. Then, we present the code snippets that represent each vulnerability. Two insecure implementations of SSL/TLS have been widely discussed and reported in the literature [1], [4], [5], [12], [13]. Figure 1 summarizes these insecure patterns and a detailed description about each pattern is as follows:

1) Certificate validation vulnerability: the certificate and all Certificate Authorities (CAs) in the certificate chain of CAs are trusted and not being verified. As illustrated in Listing 1, method (*checkServerTrusted*) does not perform any verification. To fix the vulnerability, it should go over the chain of CAs that are included in the certificate, and verifies the validity of each CA in the chain until reaching the root CA. Otherwise, an attacker can replace the original certificate of the server with a self-signed certificate to be accepted by the client, since the certificate chain is not verified. As a result, MiTM attack can be established.
2) Hostname validation vulnerability: two insecure patterns have been identified under this vulnerability. The last line in Listing 2 shows the case when the developer not only fails to validate the hostname, but he/she also allows trusting all hostnames. In Listing 3, host verification is not performed at all, because the method (*verify*) does nothing and always returns true. These are two most commonly observed vulnerabilities related to hostname validation, the existence of which allows MiTM attackers to eavesdrop and hijack the communications, by allowing an attacker to impersonate the host.

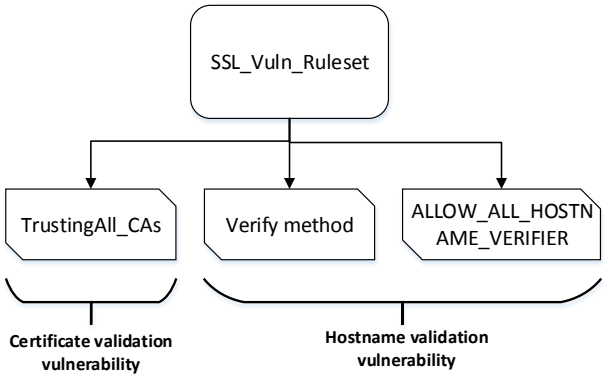

Fig. 1. Insecure SSL/TLS implementation patterns

Listing 1
TRUSTING ALL CERTIFICATES PATTERN

```
public void
    checkServerTrusted(X509Certificate[]
    chain, String authType) throws
    CertificateException {
//do nothing
}
```

Listing 2
ALLOWING ALL HOSTNAMES PATTERN #1

```
SSLSocketFactory sf = new
    MySSLSocketFactory(trustStore);
sf.setHostnameVerifier(SSLSocketFactory.
ALLOW_ALL_HOSTNAME_VERIFIER);
```

Listing 3
ALLOWING ALL HOSTNAMES PATTERN #2

```
HostnameVerifier hostnameVerifier = new
    HostnameVerifier() {
  @Override
  public boolean verify(String hostname,
      SSLSession session) {
    return true;
  }
}
```

IV. PROPOSED PMD RULESETS

The goal of this work is to create new rules for detecting insecure SSL/TLS implementation patterns. In this section, we describe the architecture of PMD and the supported methods for creating new PMD rules, introduce our assumption, and demonstrate the proposed PMD rulesets for accurately detecting the insecure SSL/TLS implementation patterns.

A. PMD Rulesets and Rules

Figure 2 illustrates the architecture of PMD, which includes the newly proposed ruleset (described in Section III). A Java class is analyzed by generating its Abstract Syntax Tree. The analyzer then examines the generated AST against a set of predefined rules, and finally a report will be generated that displays the detected bugs. Even though Data Flow Analysis (DFA) has been integrated into PMD, PMD has not supported creating rules based on DFA yet. PMD rules are organized based on different categories (formally known as Rulesets), while each ruleset contains several rules that address a single bug. Therefore, each rule possesses several properties like a description of the bug, the priority, and the detection rule.

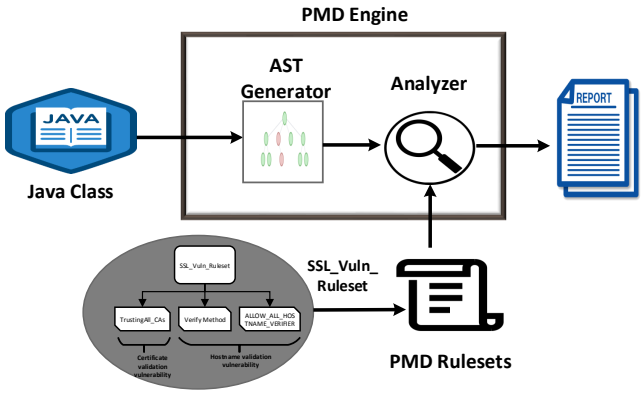

Fig. 2. PMD architecture and the proposed PMD rulesets (*SSL_Vuln_Ruleset*)

We extend the ***PMD rules*** by adding a novel ruleset, which consists of three rules for detecting the selected insecure SSL/TLS patterns. Figure 3 depicts the structure of PMD rules.

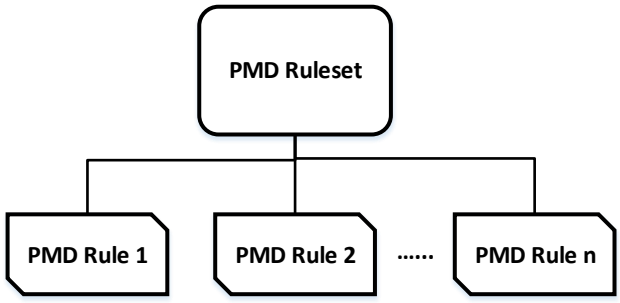

Fig. 3. PMD ruleset structure

As mentioned earlier, one of the main advantages of using PMD is the support for different methods to create new rules. In this paper, two methods can be used for creating new PMD rule set and rules:

1- Java class: PMD rule can be written as a Java class that extends *AbstractJavaRule*, and Visitor API can then be used for inspecting some properties in the generated AST of the class under analysis. Then, this rule class can be declared under a specific PMD ruleset.

2- XPath quires: this method treats the generated AST as an XML file, then we can write XPath queries to find specific patterns. PMD provides a handy tool for designing XPath queries called *PMD rule designer*,which can be used for generating the AST for the targeted pattern and creating the XPath query. Listing 4 presents an XPath that have been created using *PMD rule designer* for the insecure pattern presented in Listing 1.

Listing 4
ALLOWING ALL HOSTNAMES PATTERN #2

```
1  //MethodDeclaration[@Name='checkServerTrusted'
2  and
3  Block[count(*) = 0]]
```

In this paper, XPath method is utilized for creating the rules, and PMD rule designer has facilitated and simplified the rule creation process. The designer tool contains four windows, namely the source code (top-left), XPath query (top-right), AST & DFA (bottom-left), and the result of XPath query (bottom-right). *Our main assumption here is that the developers strive to detect any insecure implementations, and develop more secure applications.* This is a reasonable assumption as most developers have already recognized the importance of the security of their applications. Therefore, the developers apply PMD and the corresponding rulesets to detect the security vulnerabilities in their applications.

Here are the detailed steps for the developers to construct the *SSL_Vuln_Ruleset* to detect SSL/TLS implementation vulnerabilities:

1) Obtaining the source code of PMD, as we want to add new rules, it should be rebuilt again using Maven after adding new rules.
2) Using the aforementioned insecure patterns (Listings1-3) in the source code window of PMD rule designer to

generate new rules.

3) Generating the AST for the provided source code. As depicted in the AST window, AST is treated as XML file, which consists of nodes and each node owns specific properties. Accordingly, the XPath query can be created.
4) The XPath query is then generated (buttom-right window), which relates to the matched pattern in the source code XPath query in this example (Listing 4) is simplified for clarity, but more involved matching criteria can be integrated for deriving more accurate results and avoid false positive results. For instance, *checkServerTrusted()* contains two parameters, and the data type of each parameter needs to be identified. The developer can definitely fool this XPath query by adding useless statements (e.g., print statements) within the body of *checkServerTrusted()*. However, this contradicts our assumption that developers have the intention to identify insecure implementations (i.e., the developer has no malicious intent).
5) After making sure the XPath query works as intended, a new rule can be added to the *SSL_Vuln_Ruleset*. The ruleset is included in an XML file that contains the definition of a set of rules. The default location of all Java rulesets is under the following directory `PMD-java/src/main/resources/rulesets/java`.
6) The new *SSL_Vuln_Ruleset* location should be declared in the text file `rulesets.properties`, which instructs PMD about the location of all existing rulesets.

Eclipse PMD plug-in is another easier way for creating the ruleset and its rules. But this approach limits the usage of the rule into a dedicated machine, and reduces automation capabilities for running the evaluation, especially over a large number of Java classes. In our experiment, we add a single rule using Eclipse plug-in, and the detection result is presented in Figure 4. The error message displays "Consider verifying the intended certificates and not allowing all certificates by updating checkServerTrusted() method", which is in fact the suggestion for resolving the SSL/TLS vulnerability. PMD has successfully detected *checkServerTrusted()* is implemented in an insecure manner. PMD also shows other details about the detected violations, such as the line number, the name of the violated rule, etc. Suggestions for fixing this error can be also incorporated within the details of this alert, which would greatly assist the developers not only in detecting insecure patterns, but also in resolving them.

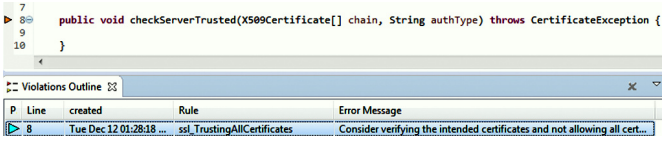

Fig. 4. PMD analysis results on Eclipse after adding a new rule

Listing 5 presents the definition of one of the detection rules in our *SSL_Vuln_Ruleset*. This rule detects insecure implementations of *checkServerTrusted()*. Line 3 shows the definition of the ruleset, which includes the the name. The actual rule is defined between Lines 9-29, and Line 15 states the priority of this rule. Finally, (Lines 18-22) contain the location where the XPath query (generated using PMD designer) is defined. To this end, we prove that PMD can tremendously facilitate the process of creating new rules for detecting new bugs, including SSL/TLS implementation bugs.

Listing 5
DEFINITION OF SSL_VULN_RULESET

```
1  <?xml version="1.0"?>
2
3  <ruleset name="SSL_Vuln_Ruleset">
4
5    <description>
6      This ruleset detects insecure implementation
           of SSL/TLS
7    </description>
8
9    <rule name="TrustingAllCAs"
10     language="java"
11     message="Consider verifying the intended
           certificates and not allowing all
           certificates"
12   <description>
13     This is an insecure implementation of SSL/
           TLS, which trusts ALL certificates.
14   </description>
15   <priority>3</priority>
16   <properties>
17   <property name="xpath">
18   <value>
19     <![CDATA[
20     //MethodDeclaration[@Name='
           checkServerTrusted'
21     and Block[count(*) = 0]] ]]>
22   </value>
23   </property>
24   </properties>
25   <example>
26     <![CDATA[
27     ]]>
28   </example>
29   </rule>
30 </ruleset>
```

V. EVALUATION

This section describes our evaluation approach, including two research questions, and the results we get to answer each research question.

We conducted the evaluation over a dataset obtained from [13]. This dataset consists of 1,517 code snippets extracted from Stack Overflow website. However, these codes cover all cryptographic implementations and are not only limited to SSL/TLS implementations. Hence, we conducted data filtration over two phases. In the first phase, codes that contain these keywords (SSL, TLS, ssl, tls, X509 and x509) are shortlisted. In the end, 597 code snippets have be shortlisted after this phase. This phase provides us all code snippets that contain SSL and TLS implementation. In the second phase, 263 files are obtained, which contain the following keywords (verify, checkServerTrusted, ALLOW_ALL_HOSTNAME_VERIFIER). The purpose of this filtration phase is shortlisting the code snippets that are related to the insecure patterns. We focus on answering the following two research questions:

- **RQ1:** How well do our detection rules perform in practice, and can they effectively detect the identified insecure patterns in real-world applications?
- **RQ2:** What is the runtime performance of PMD after using our rules?

All experiments have been performed on Ubuntu 16.04 virtual machine and 4GB memory. The modifications have been performed on the source code of PMD version 5.8.1.

1) Results for RQ1: The total code snippets that have been analyzed are 263, but 76 snippets could not be parsed correctly by PMD (will be discussed in Sec VII). For the rest of the code snippets ($263 - 76 = 187$), 54 files do not contain any insecure patterns. We manually investigate some of these files, and find that they are correctly bypassing our detection rules. This means the number of True Negative is 54. Therefore, the total number of the detected (True Positive) insecure SSL/TLS implementation patterns is ($187 - 54 = 133$), which reflects that $\%71.12$ of the code snippets in our dataset contain insecure patterns. Figure 5 shows the number of vulnerable snippets and non-vulnerable snippets.

We also have randomly investigated several code snippets that have been detected by one of our rules to verify if they really contain insecure patterns.

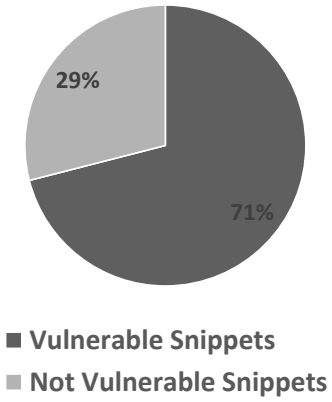

Fig. 5. Distribution of vulnerability detection using proposed PMD rulesets

Table I lists the detection results of the proposed *SSL_Vuln_Ruleset* in Figure 1. The most common insecure patterns are *"Trusting All CAs"* and *"Allowing All Hostname Verifier"*. We discover that several code snippets even contain more than one insecure patterns.

Figure 6 presents the detection results based on the identified two categories of SSL/TLS vulnerabilities. The number of detection alarms does not match the number of vulnerable code snippets, because as mentioned earlier a single vulnerable code snippets can contain more than one insecure patterns.

The results show the proposed detection rules have correctly identified the insecure code snippets, and no code snippets have been misclassified (no False Positive or False Negative). Therefore, the both detection precision and recall of our approach are 100%.

TABLE I
DETECTION RESULTS

Matching Insecure Patterns	Matched Code Snippets
TrustingAll_CAs	43
Verify Method	18
ALLOW_ALL_HOSTNAME_VERIFIER	40
TrustingAll_CAs & Verify Method	24
TrustingAll_CAs & ALLOW_ALL_HOSTNAME_VERIFIER	8
Verify Method & ALLOW_ALL_HOSTNAME_VERIFIER	0

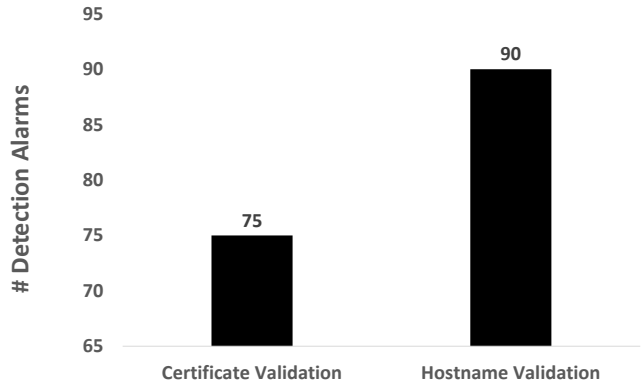

Fig. 6. Detection results according to the types of SSL/TLS Vulnerabilities

2) Results for RQ2: Measuring the overhead that might be introduced after using the new rules is crucial. Therefore, we compute the required time for analyzing the code snippets, which includes the required time for identifying which rule has been violated, and the time for parsing the generated XML report for each code snippet. The total analysis time is 144 seconds for the dataset generated after the two phases of filtration (263 code snippets). On average, the required time for analyzing each code snippet against our three rules and parsing its XML report is 0.55 second, which shows the efficiency of the proposed method.

VI. Related Work

In this section, we review additional related work. FixDroid plug-in for Android studio has been developed in [12], which addresses several limitations in Android Lint tool. It is used for helping App developers in improving the quality of their code including insecure implementations. FixDroid attempts to address the insecure implementations of SSL/TLS. However, FixDroid only considers a single pattern, which is Improper HostNameVerifier, while in our solution we consider three most commonly observed patterns.

Another plug-in called CogniCrypt is developed for assisting developers in generating secure implementation of crypto APIs [14]. This plug-in automatically generates secure implementation instead of detecting insecure patterns using static analysis technique. Although SSL API implementation

is covered by the plug-in, it does not show details about the type of SSL implementations that have been covered.

HVLearn is a blackbox testing tool for verifying hostname ins SSL/TLS implementations based on automata learning algorithms [15]. However, developers do not actually need blackbox testing techniques for detecting insecure implementation, as the source code is available. Also, HVLearn focuses only on detecting one aspect of insecure SSL/TLS patterns.

Other solutions have been developed to detected insecure implementation of SSL/TLS [5], [16]. However, these solutions intended to analyze released applications and not to assist developers in detecting insecure patterns while implementing SSL/TLS APIs.

VII. Discussions

In this section, we provide a discussion on three limitation of our approach for detecting the insecure patterns. First, after the filtration, we have 187 code snippets, while some of the snippets cannot be analyzed. Although the current dataset is sufficient for validating our new rules, in future, we need a larger dataset for drawing more affirmed conclusions. Also, we observe the duplications in the code snippets while performing the manual investigation. Furthermore, we performed a quick validation over the code snippets that have not been parsed, and our preliminary analysis shows that the AST of those file cannot be generated.

Second, even though the discussed tools in Section II does not consider the particular problem that have been addressed in this work, we need to adapt and then evaluate these other tools to compare their performance, efficiency and usability against our proposed approach.

Finally, as discussed in Section IV, we assume the developers have the motivation to find any bugs in his/her code, which is a valid assumption. But there is a possibility that the developers unintentionally inserts meaningless or debugging statements, which invalidates our rules. However, this situation can be handled by adding more conditions to the XPath query to avoid being inappropriately bypassed. There are also some cases such as the one presented in Listing 6, where a boolean variable holding a *"true"* value is returned rather than an explicit *"true"* value. In this case, PMD Data Flow Analysis should be explored to handle such cases.

Listing 6
Our rule fails to detect this insecure pattern that is similar to Listing 3

```
boolean isTesting = true;
HostnameVerifier hostnameVerifier = new
    HostnameVerifier() {
  @Override
  public boolean verify(String hostname,
      SSLSession session) {
    return isTesting;
  }
}
```

For future work, we are planning to consider the insecure implementations of other cryptographic libraries. Also, we will consider some cases that can not be detected through our current rules (e.g., Listing 6).

VIII. Conclusion

This paper sheds light on a vital implementation issue: insecure coding practices while implementing SSL/TLS APIs in Java applications. Two common vulnerabilities have been identified, while three insecure patterns that represent each vulnerability have been defined. We employ PMD static analysis tool for implementing our detection rules. After comparing it with other existing open source tools, we adopt the XPath approach for creating the new rules. In our evaluation with 187 code snippets from Stack Overflow website, we show that 71% of these code snippets are vulnerable, as they are discovered to contain various insecure patterns, which validates the effectiveness of our detection rules.

References

[1] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," *arXiv preprint arXiv:1709.09970*, 2017.

[2] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "Cognicrypt: Supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17) – Tool Demo Track*, 2017.

[3] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does cryptographic software fail?: A case study and open problems," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, ser. APSys '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:7.

[4] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "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*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 50–61.

[5] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting ssl usage in applications with sslint," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 519–534.

[6] "PMD Tool," https://pmd.github.io/pmd-5.8.1/index.html, accessed at Nov 17, 2017.

[7] "Heartbleed Bug," http://heartbleed.com/, accessed at Dec 17, 2017.

[8] "Jlint - find bugs in java programs," http://findbugs.sourceforge.net/, accessed at Dec. 2017.

[9] "Jlint - find bugs in java programs," http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/products/hammurapi/index.html, accessed at Dec. 2017.

[10] M. Aderhold and A. Kochtchi, "Tailoring pmd to secure coding," Tech. Rep., 2013.

[11] "Jlint - find bugs in java programs," http://jlint.sourceforge.net, accessed at Dec. 2017.

[12] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A stitch in time: Supporting android developers in writing secure code," 2017.

[13] F. Fischer, K. Bttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy paste on android application security," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 121–136.

[14] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "Cognicrypt: supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 931–936.

[15] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *Proceedings of the 38th IEEE Symposium on Security & Privacy*, 2017.

[16] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.