Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Why Johnny can't develop a secure application? A usability analysis of Java Secure Socket Extension API

Chamila Wijayarathna[a,*], Nalin Asanka Gamagedara Arachchilage[b]

[a] School of Engineering and Information Technology, University of New South Wales, Canberra, Australia
[b] University of New South Wales, Canberra, Australia

ARTICLE INFO

Article history:
Received 7 May 2018
Revised 19 September 2018
Accepted 21 September 2018
Available online 29 September 2018

Keywords:
Security APIs
Transport Layer Security
Programmer experience
API usability
Java Secure Socket Extension

ABSTRACT

Lack of usability of security Application Programming Interfaces (APIs) is one of the main reasons for mistakes that programmers make that result in security vulnerabilities in software applications they develop. Especially, APIs that provide Transport Layer Security (TLS) related functionalities are sometimes too complex for programmers to learn and use. Therefore, applications are often diagnosed with vulnerable TLS implementations due to mistakes made by programmers. In this work, we evaluated the usability of Java Secure Socket Extension (JSSE) API to identify usability issues in it that persuade programmers to make mistakes while developing applications that would result in security vulnerabilities. We conducted a study with 11 programmers where each of them spent around 2 hours and attempted to develop a secure programming solution using JSSE API. From data we collected, we identified 59 usability issues that exist in JSSE API. Then we divided those usability issues into 15 cognitive dimensions and analyzed how those issues affected the experience of participant programmers. Results of our study provided useful insights about how TLS APIs and similar security APIs should be designed, developed and improved to provide a better experience for programmers who use them.

1. Introduction

Nowadays, almost all aspects of our lives are associated with the use of internet and information technology. Banking, entertainment, shopping and social networking are only a few things that people do everyday with the use of internet and information technology. Not only in people's day to day life, most organizations, ranging from small businesses to governments, use internet to carry out various operations. Most of the online activities that are carried out using the internet involve storing or transferring sensitive data of users and organizations, which represents a key target to hackers.

Despite the continuous evolution of security technologies, it appears that hackers are still capable of identifying security vulnerabilities of software applications to make their attacks successful. Vulnerabilities get introduced to applications because programmers are making mistakes while developing those applications (Fahl et al., 2012; 2013; Georgiev et al., 2012). Lack of usability in security Application Programming Interfaces (APIs) that programmers use to develop applications is one of the main factors behind mistakes that programmers make, which result in security vulnerabilities (Fahl et al., 2012; 2013; Georgiev et al., 2012; Wurster and van Oorschot, 2009). For example, less usable encryption APIs can cause program-

* Corresponding author.
E-mail addresses: z5122098@student.unsw.edu.au (C. Wijayarathna), nalin.asanka@adfa.edu.au (N.A.G. Arachchilage).

https://doi.org/10.1016/j.cose.2018.09.007
0167-4048/© 2018 Elsevier Ltd. All rights reserved.

mers to make mistakes while using them, which can result in insecure password storages that let attackers to steal user passwords.

Usability issues in Secure Sockets Layer (SSL) and Transport Layer Security (TLS) related security APIs have been identified as a major cause for security vulnerabilities that exist in applications which use those APIs (Fahl et al., 2012; 2013; Georgiev et al., 2012). When APIs are not usable, programmers find it difficult to learn and understand those APIs, and hence they use APIs erroneously. To improve the usability of APIs, it is important to understand what programmers expect from APIs and why API implementations have failed to meet programmer expectations. In this work, by observing programmers who use Java Secure Socket Extension (JSSE) API (JSSE reference guide, 2017b), which is the standard implementation of TLS for Java platform and one of the most commonly used TLS APIs for Java, we attempt to identify usability issues that exist in JSSE API and how those issues affect programmers who use the API. We use JSSE API for this study since it is one of the most popular and widely used APIs that provide TLS related functionalities for Java applications (Meyer et al., 2014). The findings of this research would help in improving the usability of JSSE API as well as other APIs that provide TLS functionalities.

Following is the research question that we are trying to answer in this work

- What are the usability issues exist in JSSE API and how those issues affect programmer experience and security of applications they develop?

To answer this research question, we conducted a qualitative experimental study with 11 participants where each participant spent about 2 hours for the study. In this experiment, we used cognitive dimensions questionnaire based methodology (Blackwell and Green, 2000; Clarke, 2004; Wijayarathna et al., 2017a) and think aloud method (Van Someren et al., 1994) to identify usability issues of the JSSE API that participants experienced. From the data we gathered, we identified useful insights about usability issues exist in JSSE API. We believe the knowledge obtained in this study will help to develop more usable security APIs in the future and will guide developers to improve the usability of existing security APIs.

The paper is organized as follows. Section 2 provides an overview about SSL/TLS protocols and JSSE API. Section 3 reviews previous related research. Section 4 describes the experiment methodology and Section 5 presents the results of the study. Then we discuss the impact of usability issues we identified in this study to the security of applications that make use of JSSE API. Then we present suggestions to address identified usability issues, discuss limitations of the study and then conclude the paper.

2. TLS protocol and JSSE API

SSL and its successor TLS are cryptographic protocols that were introduced to protect network communications from eavesdropping and tampering (Freier et al., 2011; JSSE reference guide, 2017b). To establish a secure connection, the two parties that are involved in the communication must agree on cryptographic algorithms and keys to use in the communication and do client/server authentication if required. Once this process, which is known as SSL handshake, is completed, they can securely communicate by encrypting messages using exchanged cryptographic keys (Freier et al., 2011). Since, only the two parties involved in the communication know the keys to decrypt the messages exchanged, any other party that will listen to the communication can neither understand nor can change data that is being communicated. Currently, SSL protocol has been deprecated and it's predecessor TLS is being used (Barnes et al., 2015). Therefore, in this work, we are focusing on TLS protocol.

Implementing SSL/TLS communication is a complex task, which can be difficult for programmers who are not experts of security (Georgiev et al., 2012). Therefore, APIs have been developed to encapsulate the underlying complexity of SSL/TLS functionalities and to provide simple and easy to understand interfaces. Programmers use these APIs to integrate SSL/TLS into applications they develop without implementing them from scratch (Georgiev et al., 2012). OpenSSL (OpenSSL, 2016), JSSE (JSSE reference guide, 2017b), CryptoAPI (Coleridge, 1996), NSS (NSS, 2017), yaSSL (yaSSL, 2017), GnuTLS (GnuTLS, 2017) and Bouncycastle (Bouncycastle, 2017) are some of the popular APIs that provide SSL/TLS related functionalities. JSSE is one such API which provides numerous interfaces that Java applications can use to establish TLS connections and transfer data securely.

Fig. 1 shows main classes of JSSE that are used to implement a TLS communication. SSLSocket, SSLServerSocket, SSLSocketFactory, SSLServerSocketFactory and SSLEngine are the core classes of the API that provide functionalities related to creating secure sockets and communications (JSSE reference guide, 2017b). JSSE API also contains classes such as KeyManager, TrustManager, KeyManagerFactory and TrustManagerFactory that provide supporting functionalities such as setting up key materials (JSSE reference guide, 2017b).

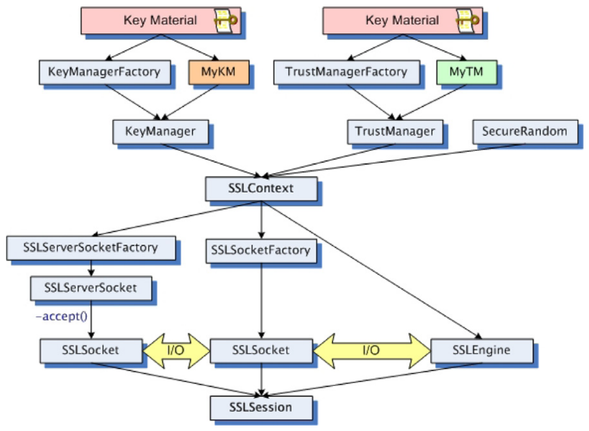

Fig. 1 – Main classes of JSSE used for TLS implementation (source : https://docs.oracle.com).

3. Related work

The trade-off between usability of APIs and security of applications that make use of those APIs is a long discussed topic

(Acar et al., 2017a; Myers and Stylos, 2016; Wurster and van Oorschot, 2009). Less usability of APIs, especially of security APIs, has been identified as one of the main reasons behind security vulnerabilities exist in applications that make use of those APIs (Acar et al., 2017a; Fahl et al., 2012; 2013; Georgiev et al., 2012).

Most programmers who are involved in the software development process are not security experts (Wurster and van Oorschot, 2009). Furthermore, programmers believe that programmes they develop are not security critical even when those are (Wurster and van Oorschot, 2009). This makes programmers less concerned about the security of applications they develop, which eventually results in those programmers developing vulnerable applications. Wurster and van Oorschot (2009) suggest two potential solutions for this problem, which are,

- Educating programmers about security
- Improving usability of programming tools and security APIs

However, they argue that educating all programmers is not practically achievable and therefore, most feasible solution is improving the usability of tools and APIs (Wurster and van Oorschot, 2009).

Many researchers discuss the importance of the usability of security APIs for developing secure software. Mindermann (2016) argues that security of an application will be far better if the libraries used to develop that application are more usable. He stresses the importance of applying usability research for security APIs to deliver more usable security APIs. Acar et al. (2017a) also highlight the importance of the usability of security APIs by pointing the fact that programmers who make use of security APIs are not experts of security.

Previous research has examined how lack of usability in SSL/TLS related APIs has resulted in introducing security vulnerabilities to applications that make use of them. Georgiev et al. (2012) identified that many security critical applications such as Amazon EC2 Java library, Amazon's and Paypal's merchant SDKs, osCommerce, ZenCart and UberCart do not implement SSL/TLS correctly and therefore, vulnerable to attacks such as the Man-In-The-Middle (MITM) attack. They identified the bad design and the lack of usability in APIs that implement SSL/TLS such as JSSE, OpenSSL and GnuTLS as one of the root causes for these vulnerabilities. By examining 13500 popular free Android applications, Fahl et al. (2012) identified that 1074 (8%) of those applications contained incorrect SSL/TLS code that makes the application vulnerable to MITM attacks. They also blamed less usable SSL/TLS APIs as the reason for these vulnerabilities. In a different study, Fahl et al. (2013) examined 1009 iOS applications and identified that 9.7% of them are vulnerable to MITM attacks and leaked sensitive information. They also claimed that root cause for this is not only the carelessness of the programmers, but also limitations and usability issues in SSL/TLS APIs.

Even though the importance of the usability of SSL/TLS APIs has been identified and discussed (Fahl et al., 2012; 2013; Georgiev et al., 2012), not much work has been done to identify usability issues exist in SSL/TLS APIs. Gorski and Iacono (2016) present 11 characteristics that need to be considered when evaluating the usability of security APIs. Green and Smith (2016) introduce 10 rules for developing usable security APIs. By considering these 2 sets of guidelines and by referring to previous work done on usability evaluation of general APIs, Wijayarathna et al. (2017a) present a cognitive dimensions framework, which consists of 15 dimensions to be used in the usability evaluation of security APIs.

There are only a few studies that involve empirical evaluations of security APIs. Acar et al. (2017a) evaluated and compared usability of 5 cryptographic APIs for python and they identified that security of applications that use security APIs is significantly related to the usability of security APIs used for developing the applications. Naiakshina et al. (2017) conducted a study where they qualitatively analyzed why developers get password storage wrong. Wijayarathna and Arachchilage (2018) used think-aloud approach and cognitive dimensions questionnaire method to identify usability issues of Bouncycastle API, an API that provide cryptographic functionalities.

Not much work has been done on evaluating or improving the usability of SSL/TLS APIs either. Parallel to our work, Ukrop and Matyas (2018) conducted a study to evaluate the usability of OpenSSL command line interface. However, they have not taken OpenSSL API into consideration. By analyzing the SSL/TLS related vulnerabilities in Android and iOS applications, Fahl et al. (2013) proposed a new framework with following characteristics to address the usability issues in existing frameworks.

- Programmers do not develop code for SSL/TLS, they just do configurations.
- Properly handle identified problematic scenarios.
- Support to differentiate between developer device and end-user device.
- Inform end-user reliably with any SSL/TLS related problems that occur.

They have examined applications that use SSL/TLS and interviewed developers who developed vulnerable code. Our results will enhance Fahl et al.'s (2013) results by adding more new findings with the observation of programmers and their thought process while they are using JSSE API to develop applications.

4. Study design

The study was designed to identify issues that programmers face as well as to observe how those issues affect them while they are using JSSE API for application development. Conducting a user study is a widely known method for identifying usability issues of APIs (Acar et al., 2017a; Clarke, 2004; McLellan et al., 1998; Piccioni et al., 2013). Therefore, we designed a user study to identify usability issues of JSSE API. In a user study for API usability, participants will complete some tasks using the API which is being evaluated (Acar et al., 2017a; Clarke, 2004; McLellan et al., 1998; Piccioni et al., 2013). The objective is to identify usability issues that participants experience while doing such tasks. We employed two techniques to identify usability issues that users encounter while using the API, which are popular in API usability community. Those are:

- Cognitive dimensions questionnaire based method (Clarke, 2004; Piccioni et al., 2013; Wijayarathna et al., 2017a).
- Think-aloud method (McLellan et al., 1998).

Cognitive dimensions questionnaire based methodology (Blackwell and Green, 2000; Clarke, 2004; Wijayarathna et al., 2017a) is the only methodology that has been proposed as a systematic approach to evaluate the usability and identify usability issues of security APIs (Wijayarathna et al., 2017a). The Cognitive Dimensions Framework presents a set of dimensions that describe aspects of a tool or an API that impact its usability (Blackwell and Green, 2000; Clarke, 2004; Wijayarathna et al., 2017a). We used the version of Cognitive Dimensions Framework proposed by Wijayarathna et al. (2017a) in this study, which consists of 15 cognitive dimensions. This framework is embedded to the usability evaluation process through the cognitive dimensions questionnaire (Blackwell and Green, 2000; Clarke, 2004; Wijayarathna et al., 2017a). In the evaluation process, few users of the API (i.e. programmers) have to complete some tasks using the API individually and then answer the questionnaire based on their experience (Blackwell and Green, 2000; Clarke, 2004; Wijayarathna et al., 2017a). In this way, evaluators can get an idea about each aspect of the API that is covered by the Cognitive Dimensions Framework from questionnaire responses and identify usability issues of the API. Even though the use of this framework and methodology is not validated through empirical evidence yet, it seems to provide good results for security API usability evaluations (Wijayarathna and Arachchilage, 2018; Wijayarathna et al., 2017b). We decided to use it as it is the only methodology that has been proposed so far to evaluate the usability and identify usability issues of security APIs.

We used think-aloud method (Van Someren et al., 1994) to get more insights in to issues that were identified by the cognitive dimensions questionnaire method and how those issues affected programmers. Using two different techniques helps to improve the reliability of data we collect (Golafshani, 2003). Furthermore, we expected that it would help us to identify a broad range of usability issues exist in JSSE API.

Even though researchers such as Acar et al. (2017a) used System Usability Scale (SUS) (Brooke et al., 1996) as the measurement of user satisfaction in their usability evaluations, we did not use it because of its non-diagnostic nature. We intended to identify usability issues of JSSE API and using non-diagnostic methods such as SUS does not provide any help in this case.

We decided to conduct the study as a remote study, since conducting a lab study with professional software developers often encounter obstacles, such as high cost and low response rate. We requested participants to record their computer screens and think aloud output while completing the task, so we can observe how they completed the task by looking at the recordings.

This study was approved by the Human Research Ethic Committee of our university.

4.1. *Task design*

Even though a wide range of user studies have been discussed and conducted to evaluate usability of APIs (Acar et al., 2017a; 2016; Clarke, 2004; Naiakshina et al., 2017; Wijayarathna and Arachchilage, 2018; Wijayarathna et al., 2017a), still there is no proper discussion on how programming tasks should be designed to use in the evaluation. In a study conducted by Naiakshina et al. (2017), they had to conduct 3 pilot studies to design a task due to the lack of guidance available on that. Therefore, we designed a programming task to closely match with the real world use cases of JSSE API. We identified the basic but common use cases of the API by referring to JSSE reference guide (2017b). We designed the task based on use cases that were available there, but also tried to get participants to use most components/classes of the API. Furthermore, we tried to make it less complex and less time consuming, so participants are likely to complete the task before they lose their interest, but still complex enough to be interesting and to allow some mistakes.

In the task, we provided participants with Java source codes for a socket client (Appendix D) and a server (Appendix C) that communicate with each other insecurely. Participants were asked to secure the communication between the server and the client by integrating TLS using JSSE API. Task description we provided for participants is available at Appendix B.

We also provided them resources that are required to complete the task such as keystore and truststore files. We requested participants to refer any learning resources they want while completing the task, so their experience would be similar to what they would do in their real life work as a software developer. Knowledge sources they used were also observed in screen recordings.

4.2. *Participants*

We recruited programmers with Java experience from Github to participate in the study. We used Github to recruit participants rather than recruiting participants from our university or local software development firms to get a more diverse sample of programmers. Furthermore, recruiting participants from Github helps to get participants with more experience in software development, which improves the ecological validity of the study (Acar et al., 2017b). We used GitHub API[1] to get a list of GitHub repositories and then filtered repositories that use Java as the main language. We used a script[2] written in Selenium[3] for this. We extracted publicly available email addresses of contributors to these repositories and sent emails inviting them to participate in our study. We offered them a $15 Amazon gift voucher as a token of appreciation for the participation. Furthermore, we offered them the opportunity to subscribe to get notified about outcomes of our research. In the invitation email, we included a link to sign up for the study with a filtering questionnaire (Appendix A). Furthermore, we informed them that participation is voluntary and participants can withdraw from the study at any time.

[1] https://developer.github.com/v3/
[2] https://github.com/cdwijayarathna/emailCrawler
[3] https://www.seleniumhq.org/

Sign up form required participants to enter their name and email address, which were required to send study material to them. However, such personally identifiable information of the participants were removed from the final data set which we used for the analysis.

We conducted 4 usability studies parallelly and recruited participants to all 4 studies together. We sent 13000 invitations to Java developers and 347 developers signed up by completing the sign up form. Some emails we sent were bounced and 7 developers requested to be removed from our list, a request we honored. Furthermore, few developers replied back to us saying that they are unable to participate in the study. Once people signed up, we filtered out those who did not have any software development experience since our target sample for the study was software developers. Furthermore, we filtered out participants with no experience in using Java (Since we invited contributors of repositories that used Java as the main language, there can be some contributors who had only contributed to them with one of the other used languages such as HTML, CSS, etc.) because if a participant faces issues with programming language while completing the task, we may not be able to clearly identify usability issues of the API they encountered. Then we divided participants who signed up into four studies we conducted based on their demographics. We selected 70 programmers for the JSSE API study and sent study material to them. Total of 12 participants completed the study. However, we had to remove 1 participant as they had not followed guidelines properly (The particular participant had not recorded their screen at the initial learning stage, but referred to resources they read in their think-aloud data while completing the task. We omit their response from final data as we could not get the complete information about their experience with the API). From the rest of the programmers, 7 informed us that they are unable to complete the experiment in the time period we expected to collect results. Rest of the programmers did not responded to our email that we sent them the study material with, and therefore, we considered them as dropouts. We did not send them any reminders, because in our initial email we informed them that if they do not respond to any of our emails, we will not email them again. We did this because we cared about not spamming people.

Table 1 summarizes demographics of the participants that took part in the study.

Table 1 – Participant demographics summary.

Demographic	Number	Percent
Software development experience		
Less than 1 year	1	9.1
1–3 years	5	45.5
3–5 years	2	18.2
5–10 years	2	18.2
More than 10 years	1	9.1
Java experience		
Less than 1 year	1	9.1
1–3 years	2	18.2
3–5 years	5	45.5
More than 5 years	3	27.3
Number of hours spending for Java programming		
Currently not using Java	4	36.4
1–10 hours per week	3	27.3
21–30 hours per week	2	18.2
31–40 hours per week	2	18.2
Participant has previously used JSSE or not		
Yes	5	45.5
No	6	54.5

4.3. *Study procedure*

Before conducting the main study, we requested 3 participants, who are known to the first author and not related to the study, to complete the task by following the guidelines to verify whether guidelines are clear and convey the expected meaning. They also answered the cognitive dimensions questionnaire proposed by Wijayarathna et al. (2017a) after completing the task. We did minor modifications to task guidelines and the questionnaire based on their results.

We used the modified task guidelines for the main study (Available in Appendix B). Once each participant signed up by completing the sign up form and consented to participate in the study, we sent them details of the programming task to do, code artefacts and resources (keystore and truststore files) to use. Participants completed the task remotely on their own computers and we suggested them to complete the task in a time comfortable to them. We requested them to think-aloud and record their screens with voice (so think-aloud results will be recorded) while completing the task.

Once a participant completed the task, they were asked to send their source code with the screen recording to us through email. Then they had to complete the cognitive dimensions questionnaire (Appendix E) which we shared via Google forms. Questionnaire given to participants was divided in to sections as in the Appendix E, but the section headers that show the cognitive dimension corresponding to each question set were not provided.

Participants spent an average of 57.67 min (Median : 47.57 minutes, sd=33.34) doing the programming task. Quickest participant to complete the task completed it in 5.78 min while slowest participant spent 104.62 min for the programming task. Participants who used JSSE API for the first time spent an average of 65.57 min (Median : 59.18 min, sd=31.8) while participants who had previous experience working with JSSE API spent an average of 48.06 min (Median : 35.85 min, sd=36.25) working on the programming task.

4 participants used Intellij Idea as the programming environment for completing the programming task, 3 participants used Eclipse and 2 participants used Netbeans. 2 participants did not use any Integrated Development Environment (IDE) for completing the Task.

4.4. *Data analysis*

Two types of data were collected during the data collection step, which are,

- Questionnaire answers
- Videos with screen recordings and think aloud results

These two types of data were analyzed separately to identify usability issues of JSSE API that each participant experienced.

The analysis was done manually by two analysts independently. We used manual analysis since our data set was small (Basit, 2003; Welsh, 2002). Questionnaire answers were analyzed prior to analyzing videos. Cognitive dimensions questionnaire (Wijayarathna et al., 2017a) contains open ended questions as well as close ended questions. Analysts evaluated and qualitatively analyzed questionnaire responses to identify usability issues encountered by each participant. Analysis was done inductively where analysts went through each participant's response, one question at a time, and noted issues of the API that the participant had mentioned. Cognitive Dimensions Framework (Wijayarathna et al., 2017a) was used as a guidance in this process. Analysts mapped specific components (quotes) of the responses with the issues they identified.

After analyzing questionnaire answers, recordings were analyzed to identify the usability issues that each participant encountered. For identifying usability issues from the screen recording data, programmer's experience was evaluated by tracking resources used by the participant, events where the participant showed surprise, events where participant had to make difficult choices, context switches, misconceptions, difficulties faced, mistakes made, requested features and time taken for tasks. Special attention was given to decisions made by participants that caused to reduce the security of programmes they developed. It took around 2–3 times the length of the video to analyze each video.

Once both analysts came up with a set of issues identified from questionnaire responses and screen recordings, the agreement of the issues were calculated using Cohen's Kappa coefficient (Cohen, 1960; Lombard et al., 2002). Cohen's kappa calculation for inter coder agreement was 0.818. Two coders resolved conflicting issues they identified through discussion.

5. Study results

5.1. Usability issues

From the study, we could identify a total of 59 usability issues that exist in JSSE API. Questionnaire responses revealed a total of 51 issues while video recording analysis revealed 24 issues of the API.

Each participant had reported an average of approximately 14 usability issues. However, if we consider the participants who used JSSE API for the first time in the study, they reported approximately 15 usability issues in average, while participants who had previous experience with the API had reported approximately 12 issues in average.

Table 2 shows the number of usability issues identified corresponding to each cognitive dimension. Highest number of issues were observed under the penetrability dimension.

Out of 59 issues we observed, 30 issues were experienced by only 1 participant. From this point onwards, we are only presenting issues that were experienced by at least 2 participants as we believe those issues are more impactful than the rest. In next sub sections, we extensively discuss each cognitive dimension (Wijayarathna et al., 2017a) of JSSE API with the comments made by participants and what we observed.

Table 2 – Number of issues identified for each cognitive dimension.

Cognitive dimension	Number of issues
Abstraction level	5
Learning style	4
Working framework	1
Work step unit	1
Progressive evaluation	2
Premature commitment	1
Penetrability	25
API elaboration	2
API viscosity	1
Consistency	1
Role expressiveness	2
Domain correspondence	5
Hard to misuse	3
End user protection	1
Testability	1
Other	4

For the ease of the presentation, we labeled participants with labels P1, P2,..., P11. They will be referred with this label from here onward. Statements made by participants that are presented in this section were not corrected for any grammatical errors and are presented as those were stated.

5.2. Abstraction level of JSSE API

Abstraction level describes "The minimum and maximum levels of abstraction exposed by the API, and the minimum and maximum levels usable by a targeted developer" (Clarke, 2004; Wijayarathna et al., 2017a). Table 3 shows issues related to the abstraction level we discuss in this section and participants who experienced each issue.

P2 and P9 in their answers to the questionnaire reported that JSSE API contains too many classes. P9 reported that *"There are lot of factories and stuff, I wish I could rather just specify the basic information (key file path and password) and get a socket pair right away"*. Furthermore, P9 suggested that *"They could get away with some of the intermediary classes. Just create the keystore and new socket. Less hastle to deal with"*.

P9, P10 and P11 mentioned in their questionnaire responses that the API's abstraction level is too low. P9 further elaborated saying *"They [classes] could have been simpler"*. P11 said in their questionnaire response that *"Lot is to be implemented by hand using the API"*. We could also observe this in P11's screen recording. P11 struggled while using *SSLEngine.wrap()* and *SSLEngine.unwrap()* methods because those methods require input parameters to be in the *ByteBuffer* format. Programmer had to convert input fields to *ByteBuffers* where they had to spend a significant time for that. If that conversion was embedded in to the *wrap* and *unwrap* functions and provided with a more higher level interface for the programmer, they would have found it easier to use. We observed a similar situation in P6's screen recording while setting up server keystore and client truststore using *KeyManageFactory* and *TrustManagerFactory* classes. It would have been easier to

Table 3 – Issues related to abstraction level. Q - Observed in the questionnaire response. V - Observed in the screen recording.

Issue	Participants who experienced the issue (Letter/s within bracket shows the method/s that indicated the issue)
API contains too many classes.	P2(Q), P9(Q)
Some classes of the API are too high level.	P4(Q), P10(Q)
Some classes of the API are too low level.	P6(V), P9(Q), P10(Q), P11(Q,V)
The API requires programmers to use class casting.	P4(V), P6(Q), P11(V)

use if there was a single function to set up keystore/truststore by giving keystore/truststore file path and password.

However, P4 and P10 found the API's abstraction level to be higher than their expectations. In their response to the questionnaire, P10 mentioned that "*The working of SSLSocket is hidden from the user. There is no way for me to know whether the API implements precisely what I am thinking*". Nevertheless, other 6 participants mentioned in their responses to the questionnaire that the abstraction level of the API is neither high nor low level.

Another issue related to the abstraction level that was reported is the use of class casting. To implement the given task, participants had to use class casting in several places as shown below.

```
SSLServerSocketFactory factory =
     sslContext.getServerSocketFactory();

//createServerSocket() method returns a ServerSocket
     object, it required to be casted to a
     SSLServerSocket object to use it in TLS
     communication
serverSocket = (SSLServerSocket)factory.
     createServerSocket(3000);
```

P6 mentioned this in their questionnaire response saying "*API does not use the powerful Java type system at all. I.e. the casts to SSLSocket and SSLServerSocket are crimes against good API. And not necessary either*". In the screen recordings, P4 showed frustration about having to cast from *Socket* to *SSLSocket* while completing the task. While going through the *SSLSocketClient* sample at JSSE reference guide (2017b), P4 mentioned that "*It is getting cast to a SSLSocket. I don't know what it is actually returning*". Especially, we observed that the use of class casting makes provided example codes difficult to read and follow.

5.3. *Learning style of JSSE API*

Learning style describes "the knowledge about the API and its security background that the programmer needs to have before starting to use the API and how the programmer would gain the knowledge about the API and its security background" (Clarke, 2004; Wijayarathna et al., 2017a). Cognitive Dimensions Framework describes 3 main approaches that a programmer can follow to learn an API (Wijayarathna et al., 2017a) which are,

- Write a couple of lines of code to try to get something working and then build up an understanding from that.
- Copy sample code provided with the API.
- Read a high level overview of the API first and only start writing code once they have an idea about the architecture of the API and how each class in the API relates to other classes in the API.

In their responses to the questionnaire, each participant selected the learning style they followed while using the API. From the 11 participants, 8 went on to learn the API by copying a sample code provided with the API. 2 participants started by reading an overview of the API first and started writing code only after getting an idea about the architecture of the API. 1 participant wrote a few lines of code and built up an understanding from that. Table 4 shows the issues related to learning style we discuss in this section and participants who experienced each issue.

P1, P4 and P11 who went on learning the API by copying a sample code mentioned in their questionnaire responses that they were not successful in that way of learning. We could identify this while observing these users' screen recordings as well. There were several reasons they suggested as possible reasons for failing to learn the API sufficiently to complete the task. P1 and P4 said in their questionnaire responses that there were too much information to read and learn. P1 elaborated on this saying "*API is too big to learn within a short period*". P2, P5, P6, P7 and P10 who thought that they were successful in learning the API also mentioned in their questionnaire responses that there were too much to read and learn. P2 further elaborated on this saying "*It would be better if the API documentation is labeled as 'essentials' and 'optional'*". It was observed in video recordings that P3 and P4 struggled and showed frustration about the amount of reading and learning that was required to use the API.

Another cause that made the API difficult to learn is that the API requires programmers to have pre-requisite security knowledge. We identified that some programmers fail to gain a sufficient understanding about the API due to their lack of expertise in security. Especially, participants mentioned that lack of knowledge in SSL/TLS, network security and keystore/truststore concepts resulted in failing them to learn and use the API properly. 4 participants mentioned in their questionnaire responses that lack of knowledge in SSL and TLS made it difficult to complete the task using JSSE API. This was apparent in their screen recordings also where we identified that 5 participants struggled to complete the task due to their lack of knowledge and understanding of SSL and TLS. They spent a considerable amount of time reading about SSL handshake and SSL/TLS protocols, but still failed to gain a sufficient knowledge to carry on with the task.

Table 4 – Issues related to learning style. Q - Observed in the questionnaire response. V - Observed in the screen recording.

Issue	Participants who experienced the issue(Letter/s within bracket shows the method/s that indicated the issue)
Difficult to learn the API by following sample codes.	P1(Q,V), P4(Q,V), P11(Q)
Too much information to read and learn.	P1(Q), P2(Q), P4(V,Q), P5(Q), P6(Q), P7(Q), P10(Q)
Difficult to use the API without previous security knowledge.	P1(Q,V), P2(Q,V), P3(Q,V), P4(Q,V), P5(V), P6(Q), p7(q) P8(Q,v), P9(V), P10(Q,V)

We also observed that participants struggled to complete the task due to their lack of knowledge on keystore/truststore concepts. P1, P4, P7 and P10 mentioned in their questionnaire responses that it would have been easier to complete the task if they had previous knowledge on keystore/truststore concepts. P4 mentioned that if they knew "*how the keystores and certificates work and why I need them*" it would have been helpful in completing the task. P10 also clarified that having a previous knowledge on keystore/truststore concepts was essential for completing the task. "*I had no idea of the algorithm behind the technology I was using, and hence was unfamiliar with the jargons like "keystore", "truststore", "digital certificates", etc. This made it difficult for me to understand the online tutorials*". This was evident while observing screen recordings also. Participants spent a significant time searching for details about keystores and truststores in Google. They search for stuff such as "*keystore*", "*java set ssl truststore*", "*keystore vs truststore*", "*java SSL socket keystore*", etc. On the other hand, P7 who was confident with keystore/truststore concepts was able to complete the task successfully and easily. He mentioned in his think aloud results that "*Having knowledge about keystores helped to do it quickly*".

Not having a previous knowledge on network security also made it difficult to use JSSE API. P2, P4 and P10 mentioned in their questionnaire responses that not having network security knowledge such as knowledge about digital certificates made it hard to use the API. We could identify this in P4's screen recording also.

5.4. *Working framework of JSSE API*

Working framework describes "The size of the conceptual chunk (developer working set) needed to work effectively". When programmers have to work with too many classes/objects/entities simultaneously, they need to keep track of the state of each of them which requires more cognitive work load (Clarke, 2004; Wijayarathna et al., 2017a).

P1, P4, P6, P9 and P11 mentioned in their questionnaire responses that the number of classes they had to work with simultaneously is too complicated. In the responses to the questionnaire, P11 mentioned that "*The API deals with a lot of different security entities, and each of them requires its own bunch of classes (e.g. trust and keystores require many classes just to be loaded and used by the engine)*". However, other participants mentioned that number of classes they had to work with is simpler or just as they expected. P3 elaborated on this in their questionnaire response saying "*Only classes and Objects were there which are just enough to complete the task*"

5.5. *Work-step unit of JSSE API*

Work step unit describes "How much of a programming task must/can be completed in a single step". In the programming context, 'a single step' can be viewed as writing one line of code. This dimension discusses the amount of work that a programmer needs to do (i.e. amount of code that a programmer needs to write) in order to complete a single task (Clarke, 2004; Wijayarathna et al., 2017a).

P2, P4 and P11 mentioned in their questionnaire responses that they had to write more code than they expected to achieve the task. P11 suggested that the API being low level can be a reason for this. He mentioned that "*The used API being low-level, the amount of code to write is not negligible. Common tasks performed during an SSL session must all be implemented, which results in a lot of code, compared to what the server achieves.*". However, 7 other participants mentioned in their questionnaire responses that the amount of code they had to write was less or just as they expected. P3 elaborated on his experience in their questionnaire responses saying "*API provided needed functionality. Only thing had to be done was using those functions*".

5.6. *Progressive evaluation of JSSE API*

Progressive evaluation describes "To what extent a partially completed code can be executed to obtain feedback on code behavior"(Clarke, 2004; Wijayarathna et al., 2017a).

Results from P1, P4 and P6 indicated that it was difficult to evaluate their code because a working client was required to test server functionalities and a working server was required to test client functionalities. Therefore, it was difficult to test the code they developed and evaluate its progress. P1 in their questionnaire response commented on this and mentioned "*It was somewhat difficult [to evaluate the progress]. In order to test the server and client we should have their counterpart implemented*". We observed this in the screen recordings and think aloud results of P4 and P6. Once the client is implemented, P6 wanted to test the code they developed, but they could not do so. It was evident in his think aloud results where they mentioned "*We now have client socket, of course it does not run, because we need server side*".

Other participants mentioned in their questionnaire responses that stopping in the middle of the task and evaluating the progress was easy. From their responses, we identified that the reason for this mismatch of opinion of participants was their level of expectation. Participants who did not expect to evaluate the progress after completing only 1 component (i.e. server or client) were happy with the opportunities they had to evaluate the progress. P10 in their questionnaire response

Table 5 – Issues related to penetrability. Q - Observed in the questionnaire response. V - Observed in the screen recording.

Issue	Participants who experienced the issueLetter/s within bracket shows the method/s that indicated the issue)
Details about the API are not well structured.	P1(Q), P4(Q,V), P6(Q), P8(V), P9(V), P11(Q)
The information about the API is all over the place and the API does not 'guide' programmer into doing the right thing.	P3(V), P4(V), P5(V), P6(Q), P9(V), P11(Q)
Difficult to find end to end examples on how to use the API.	P1(Q,V), P2(Q,V), P4(V), P10(V)
Lack of examples.	P2(Q), P4(Q,V), P11(Q)
Example codes available have assumed that keystore parameters will be set outside the code, but do not convey the idea to the reader properly.	P1(V), P4(V), P10(V)
Difficult to find details about keystores required to use the API.	P3(Q), P4(V), P9(Q), P10(Q,V)
There was not enough information to read about keystores, truststores and SSL protocol.	P3(V), P4(Q,V), P7(Q), P8(Q,V), P11(Q,V)
Error messages are not informative.	P3(V), P4(V), P5(Q), P8(Q,V), P9(Q)

commented on their experience of evaluating the progress in the middle of the task saying "*[Stopping in the middle and evaluating progress was] Quite easy. I checked the compilation / execution after: 1. defining the new SSLSockets 2. adding System properties (keystores) 3. tampering with the keystore.jks passwords*". We identified when observing the recording that P10 did not get the client and the server to work properly at every time they executed the client and the server. However, they had some expected outcomes of each execution and checked whether or not it gives the expected outcome.

5.7. *Premature commitment of JSSE API*

Premature commitment describes "The amount of decisions that developers have to make when writing code for a given scenario and the consequences of those decisions."(Clarke, 2004; Wijayarathna et al., 2017a). The API is more usable if the programmer needs to take minimum decisions while using the API.

P3 and P4 reported in their questionnaire responses that there were certain subtasks that needed to be completed prior to others and they had to learn that through trial and error. P3 mentioned in their responses to the questionnaire that "*I had to learn through trial and error, which parts need to be implemented first, for example, to write code to add keystore first or do that after creating the connection*". P4 also described a similar experience. They mentioned that "*Some objects require parameters you have to account for beforehand, I had to identify those by learning through trial and error*".

5.8. *Penetrability of JSSE API*

Penetrability describes "the way the API facilitates exploration, analysis and understanding of its components and its security related information, and the way a targeted developer should go about retrieving what is needed."(Clarke, 2004; Wijayarathna et al., 2017a). This discusses how components of API including its documentation facilitate programmers to explore, analyze and understand the API. Interestingly, this was the aspect that participants of our study reported to have the most number of issues.

From the 25 issues we identified in the area of penetrability, 17 were only experienced by 1 participant. Therefore, as we did for other dimensions, we gave more focus to the 8 issues (Table 5) that were experienced by at least 2 participants.

We identified 4 main areas of concern in these 8 issues, which are:

- Issues related to the structure/presentation of details of the API
- Issues related to examples on how to use the API
- Issues related to details available about keystores/truststores and SSL protocol
- Issues related to the amount of details shown in error messages

Participants reported in their responses to the questionnaire that details about the API are not well structured. P1 highlighted this in their response to the questionnaire mentioning "*It was difficult to find details about the API while using it, because documentation is not written in a structured manner*". P4 also suggested in their questionnaire response that "*The JSSE API is all over the place, so it was hard to find details*". P6 also provided useful thoughts on this mentioning that "*Details are described from an implementation point of view, not developer's [i.e. application developer who use the API] point of view*". We observed this issue while observing screen recordings and think aloud results of P2, P4, P8 and P9 as well. Another issue related to the structure of details of the API is that details of the API are all over the place and the API does not provide any guidance for programmers into doing the right thing. P6 mentioned in their response to the questionnaire that there was no guidance provided to find required details of the API. We identified that P3, P4, P5 and P9 also experienced this issue while observing their screen recordings and think-aloud results.

Another aspect of penetrability that we identified was related to examples of using JSSE API. We observed in questionnaire responses of P1, P2 and P11 as well as in video recordings of P1, P2, P4 and P10 that it was difficult to find end to end examples on how to use the API. P4 and P11 said in their questionnaire responses that there is a lack of examples. P1 in his response to the questionnaire mentioned that "*end to end examples were more difficult to find*". We observed this in participant screen recordings and think-aloud results where participants searched for sample code, but failed to find what they were looking for. Some participants found exam-

Table 6 – Issues related to domain correspondence. Q - Observed in the questionnaire response. V - Observed in the screen recording.

Issue	Participants who experienced the issue (Letter/s within bracket shows the method/s that indicated the issue)
Classes use factory pattern.	P4(V), P6(Q,V), P9(Q), P10(Q), P11(V)
Difficult to map from ideas in programmer's head to code.	P4(V) P5(Q), P6(Q,V), P9(Q), P10(V), P11(Q)

ples, but could not get them to work. P4 mentioned in their think-aloud results : "*They need simple examples of how I can use this*".

We also observed that participants experienced difficulties due to the lack of details exposed about the use of keystores and truststores, and due to the difficulty in accessing them. We observed in screen recordings that most of the resources that participants referred (including official JSSE reference guide (2017b)) have been presented with assuming that keystores and truststores for the client/server will be set out-side the code, however, have not properly communicated it. Even though, participants did not report this in their questionnaire responses, we observed it in screen recordings and think aloud results. In think-aloud results, P4 mentioned that "*Details about keystores and truststores were not written anywhere in the doc*". They tried to follow SSLSocketClient.java that is available at JSSE reference guide (2017b), which did not mention anything about keystore or truststore and it made participants think that they do not have to do anything with truststores in the client. This misunderstanding made them to spend more time to figure out that they need to specify the keystore and the truststore that the client and the server have to use. Furthermore, participants mentioned in both questionnaire responses and think-aloud results that it was difficult to find details about keystores that are required to use the API and how to connect to them from Java code. P10 mentioned in his questionnaire response that "*How to use keystores was difficult to find. Most websites demonstrate the use of keytool, which did not help. I had to figure out how to use keystores from external sources and sample codes*".

Moreover, participants experienced issues due to the less informativeness in the error messages returned by the API. P9 described an issue they faced because of this in their response to the questionnaire. They described "*API gave an invalid handshake error when I used API incorrectly. It should be more specific. I like error messages that tell exactly what is wrong and why*". We identified this issue while observing screen recordings and think aloud results as well. Both P3 and P4 received *NoSuchAlgorithmException* when they used a wrong password for the keystore file. They could not identify the root cause for the error they got by just looking at the initial part of the exception, but had to follow complete stack trace to identify the cause. We observed many other exceptions and error messages such as "*SSLHandshakeException : Received fatal_alert: handshake_failure*", "*SSLHandhshakeException: no cipher suites in common*", "*SunCertPathBuilderException*" and "*ValidatorException*" where participants failed to get an idea about what has really gone wrong. Appendix F shows a complete list of exceptions encountered by participants and how they reacted to them.

5.9. *Role expressiveness of JSSE API*

Role expressiveness describes "how apparent the relationship is between each component exposed by an API and the program as a whole."(Clarke, 2004; Wijayarathna et al., 2017a). This is mainly associated with how easy it is to tell what a code that uses the API does, from reading the code. Furthermore, it discusses whether or not class/interface names and exceptions are self explanatory.

Feedback we received on role expressiveness was mostly positive compared to other dimensions. Participants mentioned in their questionnaire responses that classes and methods of the API were appropriately named per the functionality they provided. P1 mentioned that "*classes were appropriately named relating to the functionality*" and also they said that "*names are self explanatory*". However, we observed some issues with naming, especially in exceptions. Exception names such as *NoSuchAlgorithmException* and *SunCertPathBuilderException* did not communicate their meanings properly to the programmer. This was discussed in more detail under penetrability. Furthermore, some participants mentioned in their questionnaire responses and think aloud output that they failed to understand functionalities of some classes such as *KeyManagerFactory* and *X509ExtendedKeyManager*.

Participants also mentioned in their questionnaire responses that it was easy to know what classes and methods of the API to use in their implementation and it was easy to read the code that use the API.

5.10. *Domain correspondence of JSSE API*

Domain correspondence describes "how clearly the API components map to the domain and any special tricks that the developer needs to be aware of to accomplish some functionality."(Clarke, 2004; Wijayarathna et al., 2017a). It describes how closely related are the classes and methods of the API to the conceptual objects that programmers make in their heads while programming. It also discusses how easy is it to map ideas in the programmer's head to code using the API.

Table 6 shows issues related to the domain correspondence we discuss in this section and participants who experienced each issue.

Participants reported in their questionnaire responses and think aloud output that the use of factory pattern was not something that were in their heads initially. P10 described this in their questionnaire response pointing out the code they had to write using factory pattern. P10 said that "*The code I had to write was bit different from the commonly used technique for declaring Socket objects*". P9 also provided some thoughts in their response to the questionnaire mentioning "*They have* *more classes than my abstraction; my abstraction will never include a factory or anything*". We observed this issue in screen recordings also where some participants (P4, P6 and P11) tried to create objects of classes using constructors. This resulted in IDE showing syntax errors for programmers that made them surprised before they realised that they need to use the factory pattern. This issue was also highlighted by Ellis et al. (2007) in an experiment they conducted.

Some participants mentioned in their questionnaire responses that it was difficult to map from ideas in their head to code while using JSSE API. P11 elaborated saying that "*Even though each step of the SSL protocol is mapped to a class or method in the API, the way classes and methods are organized surprised me in several occasions and did not match the representation of the API I had in mind*". One of the concepts that participants faced issues was keystores and truststores, which did not map to conceptual objects they had in their heads. P9 mentioned in their questionnaire response that "*It was hard to understand what those keystore files you provided were*". This was visible in screen recordings as well where participants searched for keystore related things in Google. The usage of those were also differed from what participants had conceptualized before. While searching for example code that use keystores, P4 was surprised as the usage was different from their expectation and mentioned "*This is not what I expected*". P10 also mentioned this in their think aloud results saying "*The problem I am receiving is jargons like keystores*".

However, most participants mentioned in questionnaire responses that some parts of the API mapped well to the concepts they had in their head. P3 mentioned that "*Classes and methods exposed by the API closely match to the conceptual objects. API provides expected operation and properties*".

5.11. *'End user protection' of JSSE API*

End user protection describes whether or not the security of the end user of an application developed using the API depends on the programmer who developed the application (Wijayarathna et al., 2017a). A better API should depend less on programmers for providing security to end users (Gorski and Iacono, 2016; Green and Smith, 2016; Wijayarathna et al., 2017a).

Participants of our study believed that security of end users who will use the application they developed will depend on choices they made while using the API and on how they used the API. Several participants provided interesting thoughts on this aspect. For example, P7 mentioned in their response to the questionnaire that "*Security of the end user depends on the correct usage of the API as well as the security API being correct. Incorrect usage and incorrect implementation could lead to errors and exposure of user data*". While observing participant recordings, we observed that there were certain decisions that participants had to make that had an impact on the security of the code they developed such as selection of protocols and protocol versions. If the API can be improved to depend less on programmers who use it to provide security to end-users, its usability will also improve.

5.12. *Testability of JSSE API*

Testability describes whether or not the API provides any help to test the security of applications that are developed using the API (Wijayarathna et al., 2017a).

Only 5 participants who developed a functional code reached a state where they could test the security of the programme they developed. Some of them did not try to test the security of the code they developed. As per our observation, only 1 participant did proper security testing of the developed code. P7 and P10 mentioned in their questionnaire responses that API did not provide sufficient help to test the security of the code they developed. P7 mentioned in his questionnaire response that "*API did not provide any guidance to test the security of the code. I hastily assumed it worked, considering that I had SSL-related errors along the way.*". We observed this in participant screen recordings also. In P4's think aloud results, they mentioned that "*My concern is its going to be hard to test if this actually has SSL. I am not entirely sure how I am going to test it or if I will test it, I just hope it will work*". This suggests that JSSE API needs to provide more guidance to programmers on how to test the security of applications that are developed using it.

Issues identified in API elaboration, API viscosity, Consistency and hard to misuse were experienced by only 1 participant, therefore, we have not discussed them here.

6. Discussion

In this section, we discuss how identified usability issues could lead developers to develop vulnerable applications. Even though, all usability issues that we identified might not result in security vulnerabilities, there were some issues that we like to analyze more closely.

In the results, we identified that some components of the API have a lower abstraction level that leaves programmers to do a lot of work on their own while using the API. It leaves programmers to do security critical tasks such as host name verification and protocol selection, which makes the security of the end users of applications depends a lot on programmers who develop them. Because of leaving such critical stuff to be done by the programmer, Georgiev et al. suggested that the code implementing SSL/TLS for non browser software as the most dangerous code in the world (Georgiev et al., 2012). Fahl et al. also stressed the importance of providing SSL/TLS APIs with higher abstraction level that depends less on programmers to provide protection for end users (Fahl et al., 2013). Unfortunately, these issues have not been addressed yet and still exist.

Our results also identified that there are too much information to read and learn about SSL/TLS and the API in order to successfully use the API. As we observed with our participants, programmers have to spend a significant time to learn these stuff. Furthermore, there are security related concepts such as keystore and trustsore concepts that programmers need to know to use the API. In real world scenarios, programmers work with tight deadlines and it is more than probable that they would go on to use the API without having a proper understanding about the API. Especially, programmers who are not security experts can miss important details such as details related to host name verification which would make them develop vulnerable applications.

Participants also blamed the API for the bad structure and presentation of it and its documentation. We observed that this led programmers to follow insecure behaviors while developing applications that introduced security vulnerabilities to their applications. Even though some information that programmers required while using the API were available in the official JSSE reference guide, programmers preferred to search for those information in Google and use online resources such as Stack Overflow to find those details. For example, when they came up with exceptions such as "*RuntimeExceptions : No cipher suites in common*", participants referred unreliable sources and trusted solutions given with those sources. Previous research has investigated and criticized this behavior as it leads to the inclusion of untested code in to security critical software that would make them vulnerable (Fischer et al., 2017; Green and Smith, 2016). Even though causes and solutions for above exception are clearly mentioned in the official documentation, participants did not notice them due to the bad structure of the documentation.

Some issues we discovered in this study do not have a direct effect to the security of applications that are developed using the API. For example, even though issues related to working framework and premature commitment reduce programmers' efficiency and performance, we did not observe programmers developing less secure code due to those. However, when standard APIs such as JSSE, which are considered more secure compared to other APIs are less usable, programmers tend to use alternative APIs that are less secure, which will eventually result in less secure applications (Wurster and van Oorschot, 2009). For example, programmers can use an API like Apache HttpClient, which provides more high level functionalities to embed SSL/TLS in to their applications. Even though it provides easy to use functionalities with less complexity and higher abstraction level to programmers, there are some known security vulnerabilities in some versions of Apache HttpClient that would affect the security of the resulting applications (Georgiev et al., 2012). Therefore, addressing issues identified in all 15 cognitive dimensions will have a positive effect on computer security in general.

7. Implications of the study

We believe results revealed in this study would provide a good platform to improve the usability of JSSE API as well as of other SSL/TLS related APIs. From the results, it is visible that further consideration needs to be given to the aspects like abstraction level, learning style and penetrability of the API. Following are some of the improvements that can be introduced to JSSE API to improve its usability.

- Provide more examples about how to use the API.
- Improve official JSSE reference guide (2017b) and improve its structure.
- Provide more information on using certificates, keystores and truststores with the API.
- Provide more guidance with testing the code that use the API.
- Improve API documentation of high level classes such as SSLSocket and SSLEngine providing more details about how they work inside.
- Include details about common security critical mistakes that programmers do while using the API into documentation.

This also agrees with Myers and Stylos (2016) where they suggest that providing good and easy to understand documentation would help programmers to develop programmes with better functionality and security.

Participants of our study referred the Java 8 version of the documentation (JSSE reference guide, 2017b) and our results and recommendations are mainly based on it. We observed that some steps have been taken to fix some of the issues related to documentation that we have reported here in the Java 9 version of the documentation (JSSE reference guide, 2017a), which is a good improvement However, still most of these issues exist to some extent.

Issues related to some dimensions would be harder to attend, because addressing issues related to abstraction level, working framework, work step unit and error messages would require to change available classes, methods and relationships between them, which will be hard to do with preserving the backward compatibility. It is suggested that such aspects should be evaluated in the early stage of the API development, and otherwise they would be hard to fix. However, results we obtained in those dimensions would be useful when developing new APIs that provide TLS functionalities. Following are some suggestions that we can derive from our results.

- Provide more high level classes to set keystores, truststores and certificates.
- Minimize the use of factories and class casting.
- Improve exceptions with more meaningful and easy to understand messages.
- Use more user friendly class and method names while avoiding names such as X509ExtendedKeyManager.

7.1. *Utility of the cognitive dimensions questionnaire as a tool to evaluate the usability of security APIs*

In this work we used two techniques to identify usability issues exist in JSSE API. Our results suggest that the cognitive dimensions questionnaire based methodology is a better technique to identify usability issues of a security API compared to user observation with think-aloud approach that we used. Cognitive dimensions based methodology identified 51 usability issues of the API while user observation identified 24 usability issues. Out of the 24 issues identified by user observation, 16 (67%) issues were identified in the cognitive dimension questionnaire method. We observed that some of the issues that participants reported in questionnaire answers are difficult to identify by observing participant behavior and their think aloud output. Participants tend to report issues in their questionnaire responses even when their behavior and immediate thoughts were not affected by those in an observable manner, but they later (by the time of completing the questionnaire) felt that there was an issue. For example, after completing the task, some participants reported that there are too many classes in the API, which made it difficult for them to figure out what classes to use. However, we could not identify this by observing their behavior and thoughts in the screen recordings.

Furthermore, evaluating questionnaire responses are more efficient and less time consuming compared to observing users and analyzing their video recordings. This was stated in previous research as well (Naiakshina et al., 2017). Therefore, cognitive dimensions questionnaire based methodology seems to be a better option compared to user observation in its thoroughness and operational cost.

However, we observed that user observation with think aloud gives more details about **some** issues than the cognitive dimensions questionnaire. For example, questionnaire method revealed that classes in the API have a lower abstraction level than what programmers expect. However, we could not identify what are the exact classes that participants referred by just looking at the questionnaire results. In contrast to this, user observation revealed that SSLEngine, KeyManagerFactory and TrustManagerFactory classes have a lower abstraction level than what programmers expected. Similar observations were made in the results obtained related to the amount of details revealed in error messages under penetrability dimension. We believe these observations are noteworthy when doing future improvements to the cognitive dimensions questionnaire method.

7.2. *Effectiveness of conducting a remote behavioral study*

As we mentioned previously, there are many obstacles in conducting a study by bringing programmers into a lab and observing them. Some of the main obstacles for this are,

- High cost
- Low response rate
- Lack of access to professional software developers that are capable of coming to the laboratory

Therefore, we conducted our study as a remote behavioral study. Even though there have been several API usability studies that were conducted remotely (Acar et al., 2017a; 2017b), they have not studied the behavior of participants in depth as we did in our study. Lack of think aloud output was the main issue that we thought would occur as we might not able to nudge participants if they remain silent. However, this was not the case as we did not observe any lack of quality in the results we collected due to conducting the study remotely. In screen recordings, we observed that participants provide explanations for majority of the decisions they took in their think aloud output and they kept talking throughout the whole period of the experiment.

There are some advantages of conducting a remote behavioral study compared to conducting a lab study as well. Since lab environment is different from participants' original workplace, their behavior will be different from their usual behavior. However, when conducting a study remotely, participants will complete the study at their home or at their work place using their own equipment. Therefore, their behavior would not change much from their original behavior of software development as it would happen in a lab study. Furthermore, remote study would provide a geographically distributed sample which is very difficult to achieve in a lab study (Acar et al., 2017b).

Stransky et al. (2017) mentioned that participants were willing to spend as much time as lab participants, Myers and Stylos (2016) suggest that in a remote study, participants may not provide full effort. However, this was not an issue in our case since we analyzed the participant behavior closely through screen recordings. If someone does not put enough effort for completing the task, we could easily identify that when observing the screen recordings. We came up with 1 such participant where we did not consider that participant's results.

8. Limitations

As we mentioned in the section IV, there is still no proper way to design programming tasks to use in this kind of experiments. Therefore, we designed a programming task to closely match with the real world use cases of JSSE API. Having a proper approach to design programming tasks for similar experiments would help to increase the quality of data that are being collected. At the end of our study, we identified that the programming task should require programmers to take more security critical decisions, so it would enable experimenters to identify more details on what are the security critical mistakes that programmers would do while using the API.

As we mentioned in the methodology section, the cognitive dimensions questionnaire we used in this experiment is not validated even though it has been proposed as a tool to use in security API usability evaluations. In our work, we utilized this questionnaire as it was the only methodology that was proposed to evaluate the usability and identify usability issues of security APIs at the time we conducted this experiment. Therefore, we used user observation and think aloud method to validate issues that we identified through the cognitive dimensions questionnaire. Our results showed useful evidence on the effectiveness of the cognitive dimensions questionnaire.

Since the main objective of our study was to identify usability issues of JSSE API, we used a pool of 11 participants. By using 11 participants we could reach data saturation where no new issues could be identified by recruiting more participants. It has been acknowledged that a user study would identify about 80% of usability issues of an user interface by employing 4–5 subjects (i.e. participant users) (Virzi, 1992). Virzi also identified that additional subjects are less likely to reveal new information and the most severe usability problems are likely to be detected in first few subjects (Virzi, 1992). Furthermore, more recently, Hwang and Salvendy (2010) introduced "the 10 ± 2 rule" where they argued that 10 ± 2 users will be sufficient to conduct a usability evaluation. They also stated that small participant pools are adequate when using think aloud approach for usability evaluations. Even though these results are based on user studies conducted for user interfaces, our results suggest that this holds for API usability studies as well. Our results exhibit that participant pool we used was effective in exploring the research questions we wanted to answer in this study. However, because of the small sample pool we used, we could not infer any statistically significant results such as correlation between demographic variables and outcomes. We are planning to extend this study and explore these aspects in a future study.

9. Conclusion

In this study, we conducted a remote behavioral usability study with 11 software developers to identify usability issues that exist in JSSE API. Participants were asked to complete a simple programming task which requires to develop a secure socket connection using JSSE API. They had to think aloud and record their screens while completing the task and once they finished the task, they had to answer the cognitive dimensions questionnaire (Wijayarathna et al., 2017a). Through the data we collected, we identified usability issues that exist in JSSE API.

We identified that issues related to abstraction level, learning style and penetrability contributed more towards making the life of programmers hard while using JSSE API. We observed that JSSE API's role expressiveness and domain correspondence dimensions are good which helped participants to have a better experience while using it. We analyzed and discussed how usability issues we identified in this study affected the experience of programmers who use JSSE API and the security of applications they develop.

Finally, we discussed how the issues we identified should be addressed to improve the usability of JSSE API as well as of other APIs that provide SSL/TLS related functionalities. We expect our results would help to improve usability of similar security APIs that will be developed to provide security functionalities.

Appendix A. Filtering questionnaire

- What is your name?
- What is your email address?
- What is your level of education?
  - Less than high school diploma
  - High school diploma
  - Undergraduate degree
  - Postgraduate degree
- How many years you have worked in software engineering field (including contributing to open source products)?
  - I have no experience
  - Less than 1 year
  - More than 1 up to 3 years
  - More than 3 up to 5 years
  - More than 5 up to 10 years
  - More than 10 years
- How many years you have been working on programming using Java?
  - I have no experience in Java
  - Less than 1 year
  - 1–2 years
  - 2–3 years
  - 3–5 years
  - More than 5 years
- How many hours per week you spend coding using Java in present?
  - Currently I don't use Java for programming
  - 1–10 hours per week
  - 11–20 hours per week
  - 21–30 hours per week
  - 31–40 hours per week
  - More than 40 hours per week
- Have you used Java Secure Socket Extension (JSSE) API for developing applications at least once?
  - yes
  - no

Appendix B. Task description

What is in this directory Client-src / * : This folder contains the source code of a simple java socket client. You will have to improve this source code to complete the task, which you are expected to do in this experiment.

Server-src/ * : This folder contains the source code of simple java socket server. You will have to improve this source code to complete the task, which you are expected to do in this experiment. resources / * : This folder contains a keystore file and a truststore file which you can make use in completing the task which you are expected to do in this experiment. Password for keystore : ***** , Password for client- truststore : *****

Software prerequisites

- You need to have installed a screen recording software. If you are a Windows or a Mac user, we recommend you to use "activepresenter". If you are a Linux user, you can use "Kazam" or "Record My Desktop" or a similar software you are familiar with.

Goal We have provided source codes for simple socket server and a socket client. When the socket server is started, it will start listening to port 3000 in the computer which the programme is running. A client can send messages to the server by connecting to the port 3000 of that computer.

When the provided sample client is run, it will send a message to the socket server via a socket connection. When the server receive this message, it will print it and will send a reply to the client through the same connection.

To try this out, first run the sample socket server we have provided (you can do this by opening the Server.java class in an IDE or by running it using command line) and then run the Client.java class.

In this scenario, data between server and client is transferred as plain text and in an insecure manner. If anyone could listen to this communication by tapping the communication channel from the middle, they will also be able to understand this communication and retrieve data communicated. Also client has no way to identify whether he is sending data to correct server or not. So this communication is not secure to send any sensible data.

This channel can be made secure by integrating SSL to protect the communication. SSL keep sensitive information sent across the Internet encrypted, so that only the intended recipient can understand it. Also, it provide authentication, trust, privacy, critical security and data integrity for the communication.

In this task what you have to do is secure the communication between socket server and client with SSL. You need to use Java Secure Socket Extension (JSSE) API to achieve this.

Instructions

1. Open client source code and server source code in two instances of your preferred IDE or text editor.
2. Run the server programme first and then client programme. If you are using an IDE, you can easily do this by running the project in IDE. Otherwise, you can compile and run 2 programmes in command line interface using javac and java commands. Observe how client and server communicate.
3. While performing the programming task, you have to talk what you are thinking related to the task, so that they will be recorded with screen recording. If you have never performed a think aloud study before, please refer this video (https://www.youtube.com/watch?v=g34tOmyKaMM) to see how thinking aloud protocol work, before you start on the task.
4. Now you can start working on integrating SSL to the communication between client and server. Before starting, make sure you start screen recording and choose it to capture complete screen. Select audio source as microphone, so it will also record what you say.
5. Once you completed the task, stop screen recording and save it.
6. After completing the task, please go to questionnaire and complete it with your experience of completing the task. This is to help us understand any issues you came up with while completing the task.
7. After completing everything please create a folder, copy Server.java and Client.java files, any other additional classes/files you created and screen recording to this folder. Compress the entire folder and email it to the us. Thank you very much for your voluntary participation.

Appendix C. Server code

```
import java.net.*;
import java.io.*;

public class Server extends Thread {

 private ServerSocket serverSocket;

 public Server(int port) throws IOException {
  serverSocket = new ServerSocket(port);
  serverSocket.setSoTimeout(100000);
 }

 public void run() {
  while(true) {
   try {
    System.out.println("Waiting for client on port " +
         serverSocket.getLocalPort() + "...");
    Socket server = serverSocket.accept();
    System.out.println("Just connected to "
                  +
                       server.getRemoteSocketAddress());
    DataInputStream in =new
         DataInputStream(server.getInputStream());
    System.out.println(in.readUTF());
    DataOutputStream out =new
         DataOutputStream(server.getOutputStream());
    out.writeUTF("Thank you for connecting to "+
         server.getLocalSocketAddress() + "\nGoodbye!");
    server.close();
   }catch(SocketTimeoutException s) {
    System.out.println("Socket timed out!");
    break;
   }catch(IOException e) {
    e.printStackTrace();
    break;
   }
  }
 }

 public static void main(String [] args) {
  try {
   Thread t = new Server(3000);
   t.start();
  }catch(IOException e) {
   e.printStackTrace();
  }
 }
}
```

Appendix D. Client code

```
import java.net.*;
import java.io.*;

public class Client {
 public static void main(String [] args){
  String serverName = "localhost";
  int port = 3000;
  try {
   System.out.println("Connecting to " + serverName
                + " on port " + port);
   Socket client = new Socket(serverName, port);
   System.out.println("Just connected to "
                + client.getRemoteSocketAddress());
   OutputStream outToServer = client.getOutputStream();
   DataOutputStream out =new
        DataOutputStream(outToServer);
   out.writeUTF("Test Message");
   InputStream inFromServer = client.getInputStream();
   DataInputStream in =new
        DataInputStream(inFromServer);
   System.out.println("Server says " + in.readUTF());
   client.close();
  }catch(IOException e) {
   e.printStackTrace();
  }
 }
}
```

Appendix E. Cognitive dimensions questionnaire

Abstraction level

- Were you able to implement the core functionality of the task you completed using only one class from the API or more than one class?
  - Only one class was needed
  - More than one class were needed
- Did you expect to use only one class or more than one class to implement the core functionality of the task you completed?
  - I expected to use only one class
  - I expected to use more than one class
- How would you describe your experiences with respect to the types of classes that you worked with while completing the task?
  - They were just as I expected
  - They were too low level
  - They were too high level
- Explain your selection for question 3.

Learning style

- How did you go about learning how to use the API?
  - Wrote a couple of lines of code to try to get something working and then build up an understanding from that
  - Copy sample code provided with the API
  - Read a high-level overview of the API first and only start writing code once you had an idea about the architecture of the API and how each class in the API relates to other classes in the API
  - Other
- Do you think you were successful in using this approach (you mentioned in question 1) to learn how to use the API?
  - Yes
  - No
- How would you describe your experiences related to learning how to use the API successfully? Was it just right, was there too much to read or learn, or was there not enough to read?
  - It was just right
  - There was too much to read and learn
  - There was not enough information to read
- Why (referring to question 3)?
- Which of your previous computer security knowledge areas helped you in using the API for completing the task?
- Do you think it is impossible to complete the task without above mentioned previous knowledge (referring to question 5)?
  - Yes
  - No
  - May be
- Do you think, if you had previous knowledge of any specific area related to computer security, it would have been easier to use the API?
  - Yes
  - No
- What security specific knowledge you think that would have been useful?

Working framework

- How would you describe your experiences with respect to the number of classes and objects that you had to work with simultaneously to accomplish the task you completed?
  - Too simple
  - Too complicated
  - It was just as I expected
- Why (referring to question 1)?

Work-step unit

- How would you describe the amount of code that you had to write for the task? Did you have to write more code than you expected or did you have to write less code? Please explain.

Progressive evaluation

- How easy was it to stop in the middle of the task you were working on, and check your work so far?
- Were you able to do this (check the work you have completed so far) whenever you wanted to or needed to?
  - Yes
  - No
- If not, why not?
- Could you find out how much progress you have made, or check what stage in your work you are?
  - Yes
  - No

- If not, why not?

Premature commitment

- When you are working with the API, could you go about the job in any order you like, or did the system force you to think ahead and make certain decisions first?
  - I could complete different sub tasks in any order I wanted
  - Certain sub tasks had to be completed before other sub tasks
- If you had to think ahead and make certain decisions, what were the decisions you had to make in advance?
- Was it obvious that you needed make those decisions or did you learn about this through trial and error?
  - It was obvious
  - I had to learn through trial and error

Penetrability

- What were the details of the API you had to understand in order to be able to use the API successfully in completing the task?
- How much of those details (you mentioned in question 1) were exposed in the API(including its documentation)?
- How easy was it to see or find the details of the API while you were using it? Why?
- What are the details about the API that were difficult to see or find?
- Did the API (including its documentation) provide enough information about the security relevant specifics related to the task you completed? What security specific information was missing or you had to find by referring to external sources?

API elaboration

- Were you able to use the API exactly 'as-is' or did you have to create new objects by deriving from classes in the API?
  - I could use the API exactly as it is
  - I had to create new objects by deriving from classes in the API
- To what extent you could extend the API to meet your needs?
- How would you describe your experiences regarding extending the API? Was it just right, was there not enough opportunity to extend the API or were you forced to extend it against your will?
  - There was not enough opportunity to extend the API
  - I was forced to extend it against my will
  - It was just as I expected it to be

API viscosity

- When you need to make changes to code that you have written using the API, how easy was it to make the change? Why?
- Are there particular changes that were more difficult or especially difficult to make in your code? Which ones?

Consistency

- Were there different parts of the API that do similar things? If there were different parts of the API which do similar things, were they clear from the way they appear? Please give examples.
- Were there places where some things ought to be similar, but the API makes them different? What were they?

Role expressiveness

- When reading code that uses the API, was it easy to tell what each section of the code does? Why? Which were the parts that were more difficult to interpret?
- When using the API, was it easy to know what classes and methods of the API to use for writing code?
  - It was easy
  - It was difficult
- Explain your choice for question 2.

Domain correspondence

- How closely related were the classes and methods exposed by the API to the conceptual objects that you thought about manipulating when using the API? Why? (For example, if an API for file I/O exposes a File class, does that map well to your understanding of a file?)
- When using the API, was it easy to map from ideas in your head to API code? Why?

Hard to misuse

- Did you find yourself using this API in ways that are unusual, or ways that the implementers might not have intended?
  - Yes
  - No
  - May be
- Did the API give any help to identify that you used the API incorrectly? If there were any similar incidents, please explain.
- Did the API itself give proper error messages in case of exceptions and errors, or did you have to handle them at your programme level? If you had to handle them at your level, please mention the scenario/s.

End user protection

- Do you think the security of the end user of the application you developed, depends on how you completed the task? Or does it depend only on the security API you used?
  - The security of the programme solely depends on the way I implemented it
  - The security of the programme depends on the way I implemented it as well as on the security API
  - The security of the programme solely depends on the API used
- If you think security of the end user depends on how you completed the task, in which ways does it depend?

Testability

- Did you test the security of your application after completing the task?
  - Yes
  - No
- If you answered no for question 1, can you explain why you didn't test the security of the application you developed?
- Did the API provide any guidance on how to test the security of the application you developed?
  - Yes
  - No
  - Not sure

Appendix F. Exceptions encountered by participants and their reactions

Table 7 – Exceptions encountered by participants and their reactions.

Context	Exception message	Reaction of the participant
P1 tried to run server while server instance is running.	java.net.BindException: Address already in use: JVM_BIND	Stopped both server instances and moved ahead.
P1 copied a client example that use "*hostname*" as the server address and executed it without changing	java.net.UnknownHostException: hostname	Go through the code and changed the client to comment to "*localhost*"
P1 ran server and client without specifying keystore and truststore to use	Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure, Client -: javax.net.ssl.SSLHandshakeException: no cipher suites in common	Go to JSSE reference guide and read section "sample code illustrating a secure socket connection between a client and a server". Search Google "javax.net.ssl.SSLHandshakeException: no cipher suites in common" and visit first result. Add client.startHandshake() method invocation to the client.
P1 - above solution did not work	Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure, Client -: javax.net.ssl.SSLHandshakeException: no cipher suites in common	Go to JSSE reference guide and read section "Running SSLSocketClientWithClientAuth". Check sample codes to see what is the error they have made
P1 - set system SSL system properties, but used truststore.jks as the keystore in Server.	Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure, Client -: javax.net.ssl.SSLHandshakeException: no cipher suites in common	Add client.startHandshake() method invocation to the client.
P1 - above solution did not work	Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure, Client -: javax.net.ssl.SSLHandshakeException: no cipher suites in common	Revisit StackOverflow post where the participant identified details about keystores and truststores.
P2 - tried to run a code copied from documentatio without specifying correct keystore password	java.io.IOException: keystore is tempered with, or password is incorrect.	Opened Keystore.java file where the exception was thrown and looked at the line where the exception was thrown. Then search for the correct keystore password in task guidelines file.
P2 - used same password for both keystore and truststore.	java.io.IOException: keystore is tempered with, or password is incorrect.	Commented the code that load truststore.jks and executed the code. From this P2 identified that truststore password they used were incorrect.
P3 - used wrong truststore path when running Client.java.	Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure, Client -: javax.net.ssl.SSLHandshakeException: no cipher suites in common	Added resource files to a resources folder inside the project. Searched the received exception in Google.
P3 - set ssl keystore property instead of truststore property in Client.java.	Client -: java.net.SocketException: java.Security.NoSuchAlgorithmException: Error Constructing Implementation.	Participant read stacktrace of the exception and then search Google about the exception.
P3 - used empty password as truststore password.	Client -: javax.net.ssl.SSLHandshakeException: sun.Security.validator.ValidatorException: PKIX path building failed, Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: certificate unknown	Participant tried using the absolute path of the truststore. Searched the client side exception in Google and identified the cause for the error.
P4 - ran server with wrong keystore password.	Server -: java.net.SocketException: java.Security.NoSuchAlgorithmException: Error Constructing Implementation	"*What is going on?*", Participant read stacktrace, "*So I am probably loading the keystore wrong*", Search in Google "Keystore tempered with or password is incorrect".

(*continued on next page*)

Table 7 (*continued*)

Context	Exception message	Reaction of the participant
P4 - ran server without setting keystore password system property.	Server -: java.net.SocketException: java.Security.NoSuchAlgorithmException: Error Constructing Implementation	Commented line that set keystore system property and executed code, which removed the error.Then tried to set keystore using KeyManagerFactory class referring to a sample code identified from StackOverflow.
P4 - ran server with a wrong keystore password (password used in KyeManagerFactory code copied from StackOverflow).	Server -: java.io.IOException: keystore is tempered with or password was incorrect.	"*Not sure what the key store password is*". Search google for "Keystore".
P6 - ran server and client without setting keystore and truststore.	Client -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake failure	"*I was basically expecting that*". Read JSSE reference guide to see how to set keystore and truststore.
P6 - ran server, but SSLContext is not initialized with keystore.	Server -: javax.net.ssl.SSLHandshakeException: no cipher suites in common.	Searched the exception in Google. Visited a StackOverflow post and identified the mistake.
P7 - set ssl.keystore system properties instead of ssl.truststore system properties in Client.java.	Client -: javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX Path Building Failed	Searched in Google "unable to find valid certification path to requested target". Referred a StackOverflow post and identified the mistake.
P8 - ran server, but SSLContext is not initialized with keystore.	Client -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake failure	Went through sample codes in JSSE reference guide to see what is wrong. Then searched the exception in Google.
P9 - executed client without correctly setting SSLContext.	Client -: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake failure	Searched the exception in Google and visited a StackOverflow post shown.
P10 - ran server and client without setting keystore and truststore.	Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: certificate unknown	"*I suppose this is because I didn't used client-truststore certificate, let me see how to use the client-truststore certificate*".
P10 - tested completed solution using an incorrect truststore path.	Server -: javax.net.ssl.SSLHandshakeException: Received fatal alert: internal error	"*What I think is this password[i.e. password of the truststore]is no use*". Removed system property that defined truststore password.
P10 - tested completed solution without specifying password for the keystore.	Server -: java.net.SocketException: java.Security.NoSuchAlgorithmException: Error Constructing Implementation	"*For server, I think I need to have the correct password*".
P11 - ran code after only changing the server. "*If I run the client, I should get an error*"	Server -: java.net.SocketException: javax.net.ssl.SSLException: Unrecognized SSL Message, plain text connection?	"*Exactly it is the problem*".

REFERENCES

Acar Y, Backes M, Fahl S, Garfinkel S, Kim D, Mazurek ML, Stransky C. Comparing the usability of cryptographic apis. In: Proceedings of the IEEE symposium on security and privacy (S&P). IEEE; 2017a. p. 154–71.

Acar Y, Fahl S, Mazurek ML. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In: Proceedings of the cybersecurity development (SecDev), IEEE. IEEE; 2016. p. 3–8.

Acar Y, Stransky C, Wermke D, Mazurek M, Fahl S. Security developer studies with github users: Exploring a convenience sample. Proceedings of the symposium on usable privacy and security (SOUPS), 2017b.

Barnes R, Thomson M, Pironti A, Langley A. Deprecating secure sockets layer version 3.0. IETF RFC 7568, 2015.

Basit T. Manual or electronic? the role of coding in qualitative data analysis. Educ Res 2003;45(2):143–54.

Blackwell AF, Green TR. A cognitive dimensions questionnaire optimised for users. In: Proceedings of the 12th annual meeting of the psychology of programming interest group; 2000. p. 137–52.

Brooke J, et al. Sus-a quick and dirty usability scale. Usability Eval Ind 1996;189(194):4–7.

Clarke S, Measuring API Usability, Dr. Dobb's J. Special Windows/.NET Supplement, May 2004.

Cohen J. A coefficient of agreement for nominal scales. Educ Psychol Measur 1960;20(1):37–46.

Coleridge R. The cryptography API, or how to keep a secret. https://msdn.microsoft.com/en-us/library/ms867086.aspx. Accessed: 2017-11-11.; 1996.

Ellis B, Stylos J, Myers B. The factory pattern in API design: a usability evaluation. In: Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society; 2007. p. 302–12.

Fahl S, Harbach M, Muders T, Baumgärtner L, Freisleben B, Smith M. 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. ACM; 2012. p. 50–61.

Fahl S, Harbach M, Perl H, Koetter M, Smith M. Rethinking ssl development in an appified world. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM; 2013. p. 49–60.

Fischer F, Böttinger K, Xiao H, Stransky C, Acar Y, Backes M, Fahl S. Stack overflow considered harmful? the impact of copy&paste on android application security. In: Proceedings of

the IEEE symposium on security and privacy (S&P). IEEE; 2017. p. 121–36.
Freier A, Karlton P, Kocher P. The secure sockets layer (SSL) protocol version 3.0, IETF RFC 6101, 2011.
Georgiev M, Iyengar S, Jana S, Anubhai R, Boneh D, Shmatikov V. 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. ACM; 2012. p. 38–49.
Golafshani N. Understanding reliability and validity in qualitative research. Qual Rep 2003;8(4):597–606.
Gorski PL, Iacono LL. Towards the usability evaluation of security apis.. In: Proceedings of the HAISA; 2016. p. 252–65.
Green M, Smith M. Developers are not the enemy!: The need for usable security apis. IEEE Secur Priv 2016;14(5):40–6.
Hwang W, Salvendy G. Number of people required for usability evaluation: the 10 ± 2 rule. Commun ACM 2010;53(5):130–3.
JSSE reference guide. https://docs.oracle.com/javase/9/security/java-secure-socket-extension-jsse-reference-guide.htm#JSSEC-GUID-93DEEE16-0B70-40E5-BBE7-55C3FD432345. Accessed: 2018-08-03; 2017a.
JSSE reference guide. https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html; 2017b. Accessed: 2018-08-03.
Lombard M, Snyder-Duch J, Bracken CC. Content analysis in mass communication: assessment and reporting of intercoder reliability. Hum Commun Res 2002;28(4):587–604.
McLellan SG, Roesler AW, Tempest JT, Spinuzzi CI. Building more usable apis. IEEE Softw 1998;15(3):78–86.
Meyer C, Somorovsky J, Weiss E, Schwenk J, Schinzel S, Tews E. Revisiting ssl/tls implementations: new bleichenbacher side channels and attacks.. In: Proceedings of the USENIX security symposium; 2014. p. 733–48.
Mindermann K. Are easily usable security libraries possible and how should experts work together to create them?. In: Proceedings of the 9th international workshop on cooperative and human aspects of software engineering. ACM; 2016. p. 62–3.
Myers BA, Stylos J. Improving API usability. Commun ACM 2016;59(6):62–9.
Naiakshina A, Danilova A, Tiefenau C, Herzog M, Dechand S, Smith M. Why do developers get password storage wrong?: A qualitative usability study. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. ACM; 2017. p. 311–28.
Network Security Services. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS. Accessed: 2017-11-11.; 2017.
OpenSSL. https://www.openssl.org/. Accessed: 2017-11-11.; 2016.
Piccioni M, Furia CA, Meyer B. An empirical study of api usability. In: Proceedings of the ACM/IEEE international symposium on empirical software engineering and measurement. IEEE; 2013. p. 5–14.
Stransky C, Acar Y, Nguyen DC, Wermke D, Redmiles EM, Kim D, Backes M, Garfinkel S, Mazurek ML, Fahl S. Lessons learned from using an online platform to conduct large-scale, online controlled security experiments with software developers. Proceedings of the 10th USENIX workshop on cyber security experimentation and test (CSET'17). USENIX association, 2017.
The GnuTLS transport layer security library. http://www.gnutls.org/ Accessed: 2017-11-11; 2017.
The legion of the Bouncycastle. https://www.bouncycastle.org/index.html. Accessed: 2017-11-11; 2017.
Ukrop M., Matyas V. Why Johnny the developer can't work with public key certificates: an experimental study of OpenSSL usability; Springer International Publishing.
van Someren MW, Barnard YF, Sandberg JAC. The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes. San Diego, CA: Academic Press; 1994.
Virzi RA. Refining the test phase of usability evaluation: how many subjects is enough? Hum Fact 1992;34(4):457–68.
Welsh E. Dealing with data: using nvivo in the qualitative data analysis process. Proceedings of the forum qualitative sozialforschung/forum: qualitative social research, 2002.
Wijayarathna C, Arachchilage NAG. Why johnny can't store passwords securely?: A usability evaluation of bouncycastle password hashing. In: Proceedings of the 22nd international conference on evaluation and assessment in software engineering. New York, NY, USA: ACM; 2018. p. 205–10. doi:10.1145/3210459.3210483.
Wijayarathna C, Arachchilage NAG, Slay J. A generic cognitive dimensions questionnaire to evaluate the usability of security APIS. In: Proceedings of the international conference on human aspects of information security, privacy, and trust. Springer; 2017a. p. 160–73.
Wijayarathna C, Arachchilage NAG, Slay J. Using cognitive dimensions questionnaire to evaluate the usability of security APIS. Proceedings of the 28th annual meeting of the psychology of programming interest group, 2017b.
Wurster G, van Oorschot PC. The developer is the enemy. In: Proceedings of the 2008 new security paradigms workshop. ACM; 2009. p. 89–97.
yaSSL embedded ssl library. https://www.wolfssl.com/products/yassl/. Accessed: 2017-11-11; 2017.

Chamila Wijayarathna is a Ph.D. candidate in computer science at School of Engineering and Information Technology, University of New South Wales Canberra, Australia. His Ph.D. thesis is titled as æDeveloping a Systematic Approach to Evaluate the Usability of Security APIsg. His research interests include usability of security APIs, human aspects of software engineering and programmer psychology. Chamila received a B.Sc. (Hons) in Computer Science and Engineering from University of Moratuwa, Sri Lanka.

Nalin Asanka Gamagedara Arachchilage currently works as a Lecturer in Cyber Security at the University of New South Wales (UNSW). He holds a Ph.D. in Usable Security entitled æSecurity Awareness of Computer Users: A Game-Based Learning Approachg from Brunel University London, the UK where he developed a game design framework to protect computer users against æphishing attacksg. Prior to undertaking his current position at UNSW, he worked as Research Fellow in Usable Security and Privacy in the Laboratory of Education and Research in Software Security Engineering (LERSSE) at the University of British Columbia (UBC), Canada. Before moving to Vancouver, he was a Postdoctoral Researcher in Systems Security Engineering in the Cyber Security Centre, Department of Computer Science at Oxford University. Nalin has presented his research at Facebook Headquarters, Menlo Park, California, USA and collaborated with HP in a research capacity at the HP Lab, Bristol, UK. His research has been featured in numerous media outlets including ABC News Radio, SYN Radio 90.7 FM, Sky News Australia, Daily show on Radio 2SER 107.3, Choice - Australia, Guardian labs (sponsored by Intel Corporation, Australia) and UNSW TV. He has been an invited speaker for conferences both nationally and internationally. Nalin is a Sun Certified Java Programmer (SCJP) at Sun Microsystems (now Oracle), USA. He is also a professional member of Association for Computing Machinery (MACM), the Institute of Electrical and Electronics Engineers (MIEEE) and the Australian Computer Society (MACS).