2020-09-15

A Developer Usability Study of TLS Libraries

Jonathan Blake Armknecht
Brigham Young University

A Developer Usability Study of TLS Libraries

Jonathan Blake Armknecht

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

September 2020

ABSTRACT

A Developer Usability Study of TLS Libraries

Jonathan Blake Armknecht
Department of Computer Science, BYU
Master of Science

Transport Layer Security (TLS) is a secure communication protocol between a client and a server over a network. The TLS protocol provides the two endpoints with confidentiality through symmetric encryption, endpoint authentication using public-key cryptography, and data integrity using a MAC. However, studies show that security vulnerabilities within TLS connections are often caused by developers misusing TLS library APIs. We measure the usability of four TLS libraries by performing a developer user study. Participants were given code that connects to google.com through HTTP, and tasked with using a TLS library to change the code so that it connects securely to Google through HTTPS. Our results help show what makes a library usable and what problems arise for developers using these TLS libraries. We found that the main way to ensure a TLS library is usable is to focus on having clear documentation. From our results, we provide suggestions on how to create usable documentation.

Keywords: TLS protocol, API usability, developer study

ACKNOWLEDGMENTS

I would like to acknowledge my parents and beautiful wife, Melissa, for their love and support in following my dreams and obtaining my MS degree. I would like to thank my advisor, Kent Seamons, for his wonderful advice and constructive feedback and help throughout the process of obtaining my degree. I would also like to thank Daniel Zappala, for his valuable insights that helped strengthen my thesis. Finally, I would like to thank Garrett Smith for help with performing the usability studies and thematic analysis.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The Transport Layer Security (TLS) protocol was first developed to provide security for messages sent through the Hypertext Transfer Protocol (HTTP), creating HTTP secure (HTTPS). TLS is now the most common protocol on the Internet for securing a connection between a client and a server. TLS ensures cryptographic security of messages sent between a client and a server through encryption and authentication. Cryptographic security prevents people from reading an intercepted message, as well as provides the message's intended receiver assurance that the message came from who it says it is from.

A common way to incorporate TLS into an application or program is through a developer using a TLS library. There are many TLS libraries available to developers to use in their programs. However, TLS connections in many programs are insecure [8] and can lead to user's sensitive information (e.g. banking account, credit card numbers, medical records) leaking out to malicious third parties. The problem of insecure TLS connections stems from developers utilizing the TLS library incorrectly in their program. TLS is implemented incorrectly by developers because using the API of many TLS libraries may be too difficult to understand [12], or the documentation is arduous to read, leaving developers confused about what needs to be incorporated into their program in order to have a secure connection [15]. From all of this confusion, developers abandon using the library and attempt to write their own TLS client. The TLS handshake is complex and it is easy to have mistakes in the implementation that lead to insecure connections [9]. Overall, insecure TLS connections emanate from the low usability of current TLS libraries.

Conducting developer usability studies is an emerging field of research within human-computer interaction that may provide insights into problematic aspects of an API and what can make an API more usable. However, there have been very few developer usability studies that have focused on TLS libraries [9, 12, 25]; these studies focused on the usability of one TLS library. We compare the usability of four TLS libraries to gain insights into what makes a library usable over another library and what problems arise from the use of each library. Our approach is novel because no prior work compared the usability of multiple TLS libraries.

Chapter 2

Background

This chapter describes the TLS protocol, common TLS insecurities, and related work.

2.1 TLS Protocol

The TLS protocol is used to establish a secure connection between parties, namely a client and a server, over an insecure network such as the Internet. During its embryonic stage, TLS was known as the Secure Socket Layer (SSL) which was created by Netscape in 1994 [20].

When a client connects to a server, TLS performs the handshake that sets up the cryptographic keys between the parties and performs authentication of the server. Optionally, the server authenticates the client. The handshake occurs in nine to ten steps (see Figure 2.1) depending on the method used to exchange the secret key.

In the client hello message, the client sends the server a random set of bytes, a session ID, and the cipher algorithms it can operate with. The random bytes will be used later to generate the master key which is used to create the symmetric encryption keys and MAC keys. The MAC keys gives each side confidence that the other party knows the MAC keys, and had to possess the master secret to derive those keys.

In response, the server sends the server hello message. The server sends the client a random set of bytes (used with the client's random bytes to generate the master key), a session ID, and the chosen cipher algorithm. Next, the server sends its certificate chain so that the client can authenticate the server. If the key exchange method is Diffie-Hellman

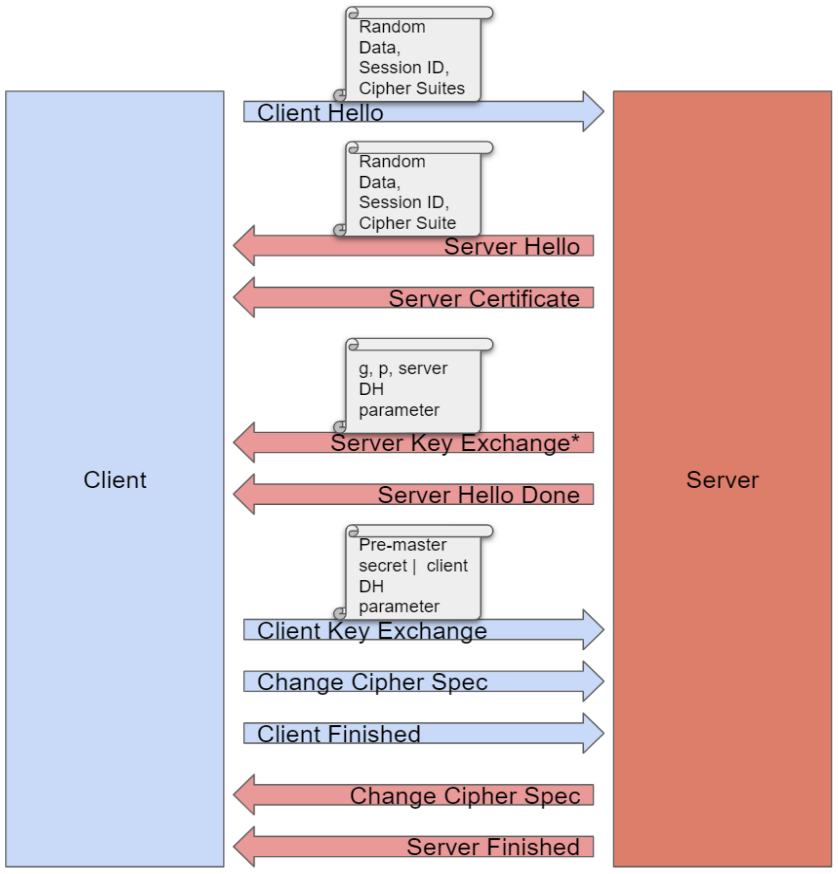


Figure 2.1: The TLS Handshake

(DH), the server sends the client the variables needed for DH in the server key exchange message. The server ends its hello stage with a server hello done message.

Next during the client key exchange method, the message sent to the server either contains the pre-master secret, a large random number encrypted with the server's public key if the exchange method is RSA, or the client's DH chosen random number. In the change cipher spec message the client tells the server that the future messages will be encrypted using the chosen cipher algorithm. The client signals the server it is done in the client finished message.

Finally, the server changes to the specified cipher algorithm and lets the client know it has changed through the change cipher spec message. The server signals the client it is done with the server finished message. Both client and server finished messages are encrypted and include a MAC that was created using the negotiated shared keys. From now on all communications between the client and the server are secure until the connection between the two of them is closed.

2.2 Insecure Connections

TLS connections become insecure when one or more of the handshake steps are performed incorrectly. Common implementation errors are in server certificate authentication and cipher algorithm selection. These areas are implemented incorrectly because the deployment process of an API is too complicated for developers [12] so developers write insecure code [9], or fail to override insecure defaults [15].

A common certificate authentication error for a client is to unintentionally trust all certificates. Trusting all certificates can lead to a man-in-the-middle (MITM) attack. If any certificate is accepted then a third party can insert itself between the client and the server and impersonate the server to the client. As a result, the third party is able to read all messages between the client and server, access any sensitive information, and even modify the content of the messages.

When using a TLS library the developer may choose a list of cipher algorithms, whether it be by default or the developer's programming choices, that are known to be vulnerable. This list is sent to the server in the Client hello phase of TLS through calls in the TLS library. The server chooses one of these vulnerable algorithms in the Server hello message to be used for encryption and authentication of messages. Choosing an insecure cipher algorithm can lead to another party obtaining messages as they are sent and decrypting them, leaking sensitive information to the third party.

Many developers must meet certain deadlines so security can fall by the wayside. This happens because learning to use a vague crypto API hinders the progress of building the code of the application. Security does not even cross some developer's minds when creating an application [16]. In order for their application to have some security, developers end up calling the defaults of the API where some defaults are not secure [15].

Some developers may end up implementing the TLS protocol fully themselves [9]. This occurs when there is a lack of awareness of existing TLS libraries, or when the developer finds the API difficult to use. These homegrown TLS libraries often lead to insecure connections.

2.3 Related Work

2.3.1 Developer Usability Studies

Much research has been done looking at the usability of APIs in general [18, 19, 27]. The more usable an API, the better a developer is at implementing it correctly into their application. Scheller et al. [23] created a framework that would automatically measure an API's usability based on certain criteria. However, because of its limitations, the framework can only fully measure a limited pool of APIs. This pool does not include TLS libraries because of the complexity of these libraries' documentation.

Others have narrowed the focus of usability testing to cryptographic APIs [5, 8, 15, 16, 26]. Acar et al. [5] measured the usability of five python crypto libraries. Their results showed that while a simple API does provide better security implementations, it is not enough. They postulated better documentation and easy-to-use code examples give a crypto API higher usability. Looking at how crypto APIs were used in 11,748 Android applications, Egele et al. [8] found that 10,327 of these applications violated at least one ground rule of cryptography, making the application insecure. In contrast, Xie et al. [26] studied developers instead of code so they could find the reason programmers make insecure applications. Their results showed that it is not the API, but that developers introduce the vulnerabilities. Insecurities are introduced because developers either do not understand security, or they feel that they are not responsible for incorporating security into programs. Oliveira et al. [16] added to what Xie et al. found by conducting a study to see how developers think. They found that security is not a priority to a development environment, it is not on a developer's mind when coding, and that to think securely required cognitive effort. If a developer was primed to protect a program from a security vulnerability, then the correct secure code was added. Naiakshina et al. [15] also found this result; they conducted a usability study to find how a developer goes about storing a password. Their results found that developers concentrate on functionality before security, and that many APIs require the developer to opt-in to use secure options. Prior studies all focus on general cryptographic APIs and not the usability of TLS APIs specifically.

Finally, there have been very few usability studies of TLS libraries. Krombholz et al. [12] conducted a usability study on the deployment of HTTPS. Their results showed that the deployment process is performed incorrectly and that server defaults are not strong enough. Both of these give rise to insecure TLS connections. Their study looks at the setup of configuring a server and not the TLS library calls that need to be made to set up the client's connection. Ukrop et al. [25] measured the usability of OpenSSL. The results of the study showed that OpenSSL's usability is insufficient. However, participants only used the OpenSSL command line portion of the library. Fahl et al. [9] investigated how TLS is used in apps. Their findings show that many developers go to online forums for help when trying to implement TLS. Answers found on the forums show that there is a lack of understanding of how TLS works and how to correctly implement it. Some answers contain incorrect code that gets propagated into other apps because the developer wanted to customize the TLS connection.

Overall, there has been much research into the usability of general APIs. However, there have been very few usability studies of how easy it is for developers to use TLS libraries.

2.3.2 Guidelines for Creating Usable APIs

Some research has focused on creating guidelines for those that create an API so that when followed, the API is much more usable. Myers and Stylos [14] suggested ten guidelines that would make an API usable to a developer. While their guidelines are helpful, they do not take into account security. The ten guidelines set forth by Green and Smith [10] are a set of criteria that makes a crypto API usable and secure. We use them to hypothesize which of the four libraries chosen will be the most usable to developers.

Taking the lessons learned from making end-user programs usable, Acar et al. [6] talk about how to aim these lessons towards developers. They propose a research agenda on how to conduct a developer usability study. The agenda focuses on what one can do to conduct a thorough developer study by keeping in mind how developers think. Acar et al. speak about how it can be hard to get actual professional developers with how busy they can be, because of this, their research suggests using students in place of professional developers. They performed a study that showed that professionals did not outperform students when making a program more secure. The agenda also suggests that when performing a developer usability study researchers should use interviews to obtain qualitative data and surveys to obtain quantitative data. Finally, they proposed that researchers should focus on evaluating APIs that are similar to one another and learn about the benefits and drawbacks of each API. We did this by comparing four TLS libraries that are similar.

2.3.3 Improving Usability of TLS Libraries

While there has been much research in the area of testing an API's usability, few have taken on the task of making an API more usable. Jain et al. [11] redesigned the Android location API and then compared code written by developers using the original API and the redesigned one. The research found that developers were nudged into making secure programming choices when using the redesigned API. The redesigned API protected the privacy of users more by collecting user's location less often. Jain et al.'s redesigned API was more usable to participants than the original API because the documentation was easier to understand, the API was easier to implement, and overall the API was straightforward to use.

Other studies have focused on designing usable TLS libraries. O'Neill et al. [17] designed a new TLS library that requires fewer lines of code to set up a TLS connection. The Secure Socket API (SSA) overloads POSIX function calls to set up a TLS connection. The SSA allows developers to use function calls that are familiar when setting up a TLS connection. In order to create the SSA, OpenSSL was assessed to see what could be implemented better. They found that many functions in OpenSSL had "non-intuitive semantics, confusing names, or little-to-no use in applications." To make the SSA transparent, the researchers worked on reducing the number of TLS functions in current APIs to the POSIX functions network developers are familiar with. Thus developers are able to use POSIX calls to set up a TLS connection by selecting TLS as the protocol to use, instead of currently built-in protocols TCP or UDP. Another strength of the SSA is that by default it always uses the latest secure version of TLS ensuring that connections are as secure as they currently can be. In order to test how usable the SSA was developers were asked to port four applications to use the SSA. The four applications were wget, lighttpd, ws-event, and netcat. The researchers' data showed that porting these applications to use the SSA removed thousands of lines of OpenSSL code and was relatively easy even for developers without knowledge of the code beforehand. The time it took the SSA to download an encrypted 1MB file and decrypt it was compared to OpenSSL's time. They saw that the SSA and OpenSSL took the same amount of time for local connections but was faster than OpenSSL with remote connections. Thus, the SSA improves the overall security of an application without sacrificing time to do so.

Amour et al. [7] created a new TLS library called libtlssep that is a simpler API to improve security when compared to OpenSSL. Libtlssep improves security by making a separate process that is in charge of all TLS communications. By spawning a separate process, it provides isolation of cryptographic secrets, so an application's secrets cannot be leaked even if a developer added code that would normally reveal the secrets through an exploit. They analyzed the overall security, performance, and usability of libtlssep when compared to OpenSSL. Libtlssep requires fewer function calls compared to OpenSSL to securely set up a TLS connection. All libtlssep's functions and return errors are easy to understand without developers having to read about them in the documentation. From all this, they found libtlssep has higher security and usability when compared to OpenSSL. Higher security and usability stems from the developer not having to call a certificate verification function. Instead, libtlssep does this under the hood, and will not return a valid TLS connection if verification fails. To continue testing the usability of libtlssep they ported two programs, wget and lighttpd, to use libtlssep instead of OpenSSL. Developers that were asked to port the two programs to use libtlssep did not have a background in either program. Their research showed that it was easy for the developers to replace OpenSSL code with libtlssep code, and that it required fewer API calls and lines of code to establish a secure connection. To evaluate performance, they compared libtlssep's and OpenSSL's latency and throughput of a client connecting to a local server. Latency was measured to be the same between OpenSSL and libtlssep. The throughput of both libraries was measured to see how long a client took to download 10MB, 100MB, and 1,000MB files. Throughput was measured on average to be the same between OpenSSL and libtlssep. Overall, the researchers showed that libtlssep was more secure and usable compared to OpenSSL and that this increased usability and security did not sacrifice libtlssep performance.

Fahl et al. [9] created an Android framework that helps write secure TLS code and perform certificate validation checks on vulnerable applications. The framework eliminates the responsibility of writing TLS code for developers. The framework performs all of the TLS connections with the developer only needing to turn on TLS and choose from a predetermined set of secure options for how the framework should configure the connection. Another ability of the framework is that it can distinguish between a developer device and an end-user device. This provides the developer with the ability to prototype their application by trying things out, without having to worry if their tinkering puts the end-users at risk of an insecure connection if the developer forgets to take out a piece of code. One such thing developers can do is turn off certificate validation of certain apps only when that app is run on a developer-enabled device, thus keeping the end-user's TLS connections on their devices secure. They ran a usability study on the framework by having developers add TLS to their applications. Fahl et al. then showed how the framework could do all of the TLS code as long as the developer configured it. After interviewing the developers, Fahl et al. found that the developers' reactions to the framework were all positive and that they wanted to use the framework to ensure easily made and secure TLS connections in their applications.

Chapter 3

Pre-Study

This chapter describes how and why we chose the four TLS libraries used in our study. We also present a preliminary analysis of the libraries.

3.1 Chosen TLS Libraries

We identified a number of potential libraries for our study and conducted a preliminary analysis. The main factor for selecting a library was that it was implemented using C, allowing us to recruit participants in our study that only need to know C and avoid confounding factors due to programming language choice. The C programming language was also chosen as it is the language used in classes at BYU that teach students about POSIX and network functions. A quick search on the Internet provided us with a list of currently available libraries written in C. From this list we favored libraries that were well-known and up-to-date. In the end, we selected four TLS libraries to test. These libraries include OpenSSL, GnuTLS, s2n, and the SSA.

OpenSSL is a widely used open source library for creating TLS connections. OpenSSL has many available options present for a developer to set up a secure TLS connection, if the developer uses the library correctly. These options include what version of TLS to use for the connection, what cipher algorithm to use, and if certificates should be verified.

GnuTLS is written for use in programs that run under the GNU general program license (GPL) [4], since these programs cannot use OpenSSL. Like OpenSSL, GnuTLS has many of the same available options present for a developer to set up a secure TLS connection.

Amazon's s2n is designed to be light and fast, having security as a priority [3]. Ultimately, it was created to be more intuitive to developers that understand POSIX calls [3]. s2n has fewer lines of code compared to OpenSSL, 6,000 lines to 500,000 lines [1]. s2n was designed to make encryption more developer-friendly and save the time it takes a developer to create a TLS connection within their application. In fact, Amazon Web Services (AWS) has replaced OpenSSL on all of their server machines with s2n for any TLS traffic to their Amazon Simple Storage Service (Amazon S3) [1].

Finally, the SSA boasts of higher usability for making TLS connections [17], however, there has not been a formal usability study on the SSA. Therefore, we wanted to be sure to include it as one of the studied TLS libraries.

3.2 Preliminary Analysis

Green and Smith [10] presented the following ten principles in creating a more usable crypto API:

1. Integrate crypto functionality into standard APIs so regular developers don't have to interact with crypto APIs in the first place;
2. Design APIs that are sufficiently powerful to satisfy both security and non-security requirements;
3. Make APIs easy to learn, even without cryptographic expertise;
4. Don't break the developer's paradigm (common developer knowledge and practices);
5. Make APIs easy to use, even without documentation;
6. Make APIs hard to misuse, with incorrect use leading to visible errors;
7. Ensure that the defaults are safe and never ambiguous;
8. Include a testing mode;

9. Ensure that APIs are easy to read, and that it's easy to maintain the code that uses them (updatability); and
10. Allow APIs to assist with and/or handle end-user interactions.

Principle	1	2	3	4	5	6	7	8	9	10	avg.
SSA	10	7	9	10	9	8	10	1	10	5	7.9
OpenSSL	5	6	3	10	2	1	6	1	7	1	4.2
GnuTLS	5	6	5	10	3	1	2	1	1	2	3.6
s2n	7	7	7	10	6	1	10	1	7	5	6.1

Table 3.1: Scores of each TLS Library on each principle

As a preliminary analysis we experimented with each of the four TLS libraries by writing a simple TLS client that connected securely to https://google.com. We then rated how well each library supported each principle on a scale of 1-10. A rating of 10 means that the TLS library fully satisfies the principle. Table 3.1 lists the scores of the four libraries for each principle. The SSA scored the highest with an average score of 7.9 over the ten principles. This higher usability comes from setting up TLS connections through a simple extension to the POSIX API. Second was s2n with an average of 6.1, third was OpenSSL with an average of 4.2, and last was GnuTLS with an average of 3.6. OpenSSL and GnuTLS have many options available for setting up a TLS connection. However, developers can get bogged down with not knowing what options to include or may forget to include a necessary function call. The remainder of this section explains the rationale for the ratings in the table.

Principle 1 The SSA scored perfectly because it integrates nicely with POSIX calls and has strong default parameters. s2n also uses POSIX calls but the defaults were not as strong as the SSA. Both OpenSSL and GnuTLS scored the lowest because the developer has to call crypto functions to make a connection, and the default parameters are hard to find.

Principle 2 Both the SSA and s2n scored well because they both satisfy all security needs through the use of POSIX calls. OpenSSL and GnuTLS did adequate on the security side, but both allow for tinkering by a developer that could make the program insecure. None of the four libraries took non-security into mind.

Principle 3 By far, the SSA was the easiest to learn since it uses POSIX calls and does not require extra lines of code that s2n needs. OpenSSL was hard to learn because of all the API calls available to a developer. OpenSSL's documentation did not make it any easier. GnuTLS has the same problem of many API calls. However, its documentation did help with learning what calls need to be made.

Principle 4 All four stuck to common security protocol knowledge, thus keeping the developer's paradigm intact, scoring perfect on principle 4.

Principle 5 The SSA is easy to use since it uses standard POSIX calls and only requires one change when creating the socket to create a TLS connection. s2n is easy to use as well after consulting the documentation for what functions need to be called. Both OpenSSL and GnuTLS have many API calls available but the documentation for both is unclear. A more effective approach to learning how to use the library was to search Stack Overflow for examples to follow.

Principle 6 Only the SSA generates an error when certificate validation is turned off. However, it does not specify that the error is for not performing certificate validation. None of the other three TLS libraries generate an error when certificate validation is turned off. This leaves these three vulnerable to a MITM attack if the developer forgets to turn certificate validation on.

Principle 7 The SSA defaults to use the latest uncompromised TLS version. s2n's default cipher suite is strong and cannot be changed by a developer. OpenSSL's defaults are ambiguous even after looking through its documentation. The GnuTLS documentation says to always start a session using the default settings, but the settings could not be found.

Principle 8 None of the four TLS libraries had a testing mode that easily turned security code off and on. This leaves it up to the developers to write their own testing mode but this mode can be mistakenly left in the final product leaving the program vulnerable.

Principle 9 The SSA keeps protocols up to date and uses the most secure TLS version. s2n and OpenSSL are both easy to update with unofficial online help. GnuTLS scored the lowest because it relies on the gmplib library. The website for the gmplib library [2] said that GMP servers are not up to date with the latest security features and may have vulnerabilities to Meltdown and Spectre. Therefore, any of the resources found on this website, including gmplib, may be compromised.

Principle 10 None of the four libraries assisted with error handling. They all required the developer to write their own error handling code. However, some of the libraries performed better on this principle because all functions returned a known standard error code of 0 for failure and 1 on success. By returning the values 0 or 1, the developer can better use them in a logical expression when checking the return value of a function call and not have to look up the meaning of the value in the documentation. The SSA and s2n handled this principle the best with both of them returning standard error codes. GnuTLS defines errors through macro names, but the names are ambiguous and require the developer to look them up in the documentation. Lastly, OpenSSL does not follow standard error codes. Some of OpenSSL's functions can return 0 on an error while others return -1 on an error. The documentation for OpenSSL does explain what the return value of a function can be, but this places an extra burden on the developer.

Based on these preliminary results using Green and Smith's [10] ten principles, we hypothesize that our participants will score the SSA as the most usable, followed by s2n, then OpenSSL, and finally GnuTLS.

Chapter 4

Methodology

We conducted a 29-person within-subjects laboratory study approved by our institution's ethics review board to assess the usability of the four TLS libraries.

4.1 Participant Recruitment

As previously mentioned in section 2.3.2, Acar et al. [6] found that when professional developers and students were given security-related programming tasks neither group outperformed the other in the area of being secure. Therefore, we recruited 29 BYU students due to the ease of recruiting these participants over professional developers and to reduce costs.

Obtaining 29 participants allowed us to have a sample size of 29 for OpenSSL, and 9-10 for GnuTLS, s2n, and the SSA for usability testing. Having 29 participants for the OpenSSL API allowed us to find on average 99% of the problems associated with using it with a confidence level of 90% [13, 21, 24]. As for the other three APIs, the 9-10 participants per API allowed us to find on average 95% of the problems associated with each API with a confidence level of 90% [13, 21, 24].

Students that participated in the study were required to have a knowledge of network communications, sockets, and HTTP in order to complete assigned research tasks. Before starting the study, study coordinators asked participants if they had taken CS 324 (our department's second systems programming course, where they learn about sockets, networking, multiplexing I/O with threads and polling) or an equivalent class which teaches these concepts. We recruited students using flyers (see appendix 8.6) and word of mouth. The flyers were posted throughout campus, handed out in classrooms, sent through Slack, and emailed to prospective participants.

Initially, participants were compensated 12 USD for an hour of their time. However, we recruited only 9 participants in the first month. This low level of participation is highly unusual for our lab, as previous studies have obtained over 30 participants in less than two weeks. We believed this was due to the demographics of participants our study was looking for. Our prospective participants may not learn about network communications at BYU until they are juniors or seniors in their undergraduate degrees. By this time, many of them may already have jobs off-campus as developers where they are getting paid much more for an hour of their time. Therefore, we increased the pay to 15 USD for an hour, but this did not cause any new participants to sign up. We raised the pay to 20 USD after a comment by a student who said, “I wouldn’t do it for less than 20 dollars”. After raising the pay to 20 USD, we obtained 20 new participants over one month.

4.2 Participant Tasks

Once participants signed consent forms and answered demographic questions (see appendix 8.3) we began the developer study. We conducted a within-subjects study. Participants were assigned two of the four TLS libraries. With each assigned library, the task was to use the library to connect securely to google.com. Overall, each participant completed two tasks, one task for each library assigned. At first, we hoped to perform a normal within-subjects study where all four libraries were given to all participants to perform four separate tasks with each library. However, we found this to be impractical from our pilot study with five participants. Some of the five participants took up to two and a half hours to complete all four tasks. Due to potential user fatigue, we modified our design to only have participants use two of the four libraries.

For our study, participants were provided with code for a simple client program (see appendix 8.1). This client program uses POSIX calls to establish an insecure connection to a server. Participants were told they would be fixing an insecure client program to connect securely to a server and that they would be doing this twice, once with OpenSSL and a second time using either GnuTLS, s2n, or the SSA. They were also told that they were able to use any online resources to help them accomplish both tasks and that the official documentation for each library was bookmarked for them in the browser to use as a reference if they would like to. The links given to participants are presented in section 8.5 of the appendix.

Participants were given 25 minutes for each of the two tasks. If the participant believed they had completed the task, they informed the study coordinator and ended the task early. After the first 25 minutes, participants were given the same insecure client program and again asked to make it secure using the second TLS library assigned to them.

Each participant was assigned OpenSSL as their first library. The second library was randomly chosen between GnuTLS, s2n, or the SSA. We wanted all participants to use OpenSSL as this is the most widely used TLS library. By doing it this way, we are able to compare OpenSSL's usability to the other three libraries. We were able to observe the problems participants had with each library as well as what participants found usable with each library.

4.3 Data Collected

We gathered and analyzed both quantitative and qualitative data. All data we collected was anonymized. The data collected included: 1) program analysis, 2) semi-structured interview responses that measure participants' perceptions of the two assigned libraries, 3) single ease question (SEQ) scores of what the participant thought of the two assigned libraries, and 4) the time it took for participants to perform each of the two tasks.

After each task the participant was asked to fill out an SEQ rating to measure their perceived difficulty or easiness of the task (see appendix 8.4). The SEQ uses a standard question ("Overall, how difficult or easy was the task to complete?") on a 7-point Likert scale. Even though the SEQ is one question it is still a reliable metric in determining the usability of a task [22]. We chose the SEQ so participants would not get survey fatigue since they were also asked interview questions after each task. The interview questions were structured to help us learn about the participant's perception of the usability of the two assigned TLS libraries used. The interview questions are presented in appendix 8.2.

We validated each participants' secure client program by verifying that it compiled and analyzing the creation of a secure TLS connection. Two study coordinators independently analyzed the interview responses to identify categories and themes. The coordinators then met to reach consensus on any differences.

The coordinators studied the steps in the TLS handshake, and identified the functions in each library that are used to perform a secure handshake. The coordinators also identified the correct parameters needed for each function (i.e., TLS version, cipher suites, certificate checks). The coordinators then analyzed participant's code to verify if the proper functions and parameters were present. The coordinators first performed their analysis independently and then met to reach consensus on whether the code was correct or not.

4.4 Demographics

Our participants were all young and highly-educated males, with 28 (96.5%) ages 20-29 and 1 (3.5%) age 30-39. Over half of the participants (17; 58.6%) had some college but no degree, 4 (13.8%) had an associate degree, and 8 (27.6%) had a bachelor degree. All participants considered themselves to have average or above average programming skills. Only one (3.5%) participant considered themselves to be far above average, 15 (51.7%) somewhat above average, and 13 (44.8%) average.

Chapter 5

Results

In this chapter, we analyze and discuss results from quantitative and qualitative data gathered during our developer usability study.

5.1 Timing

The time it took participants to complete each task was obtained from screen recordings. Timing started when the study coordinator informed the participant they may begin to when the participant said they were finished, or ran out of time.

Timing data for participants that received OpenSSL and GnuTLS as their assigned libraries is presented in Figure 5.1. From this group, only P22 did not run out of time. They said they completed the task at 24 minutes and 54 seconds when the task was using OpenSSL. However, all participants ran out of time for GnuTLS and did not complete the task.

The time it took participants that received OpenSSL and s2n as their assigned libraries is presented in Figure 5.2. Only one participant did not run out of time when using OpenSSL. P29 finished the OpenSSL task at 19 minutes. All participants in this group ran out of time for s2n and did not complete the task.

Finally, looking at the group that received OpenSSL and the SSA as their assigned libraries we see a decrease in time taken (see Figure 5.3). When given the SSA, seven out of

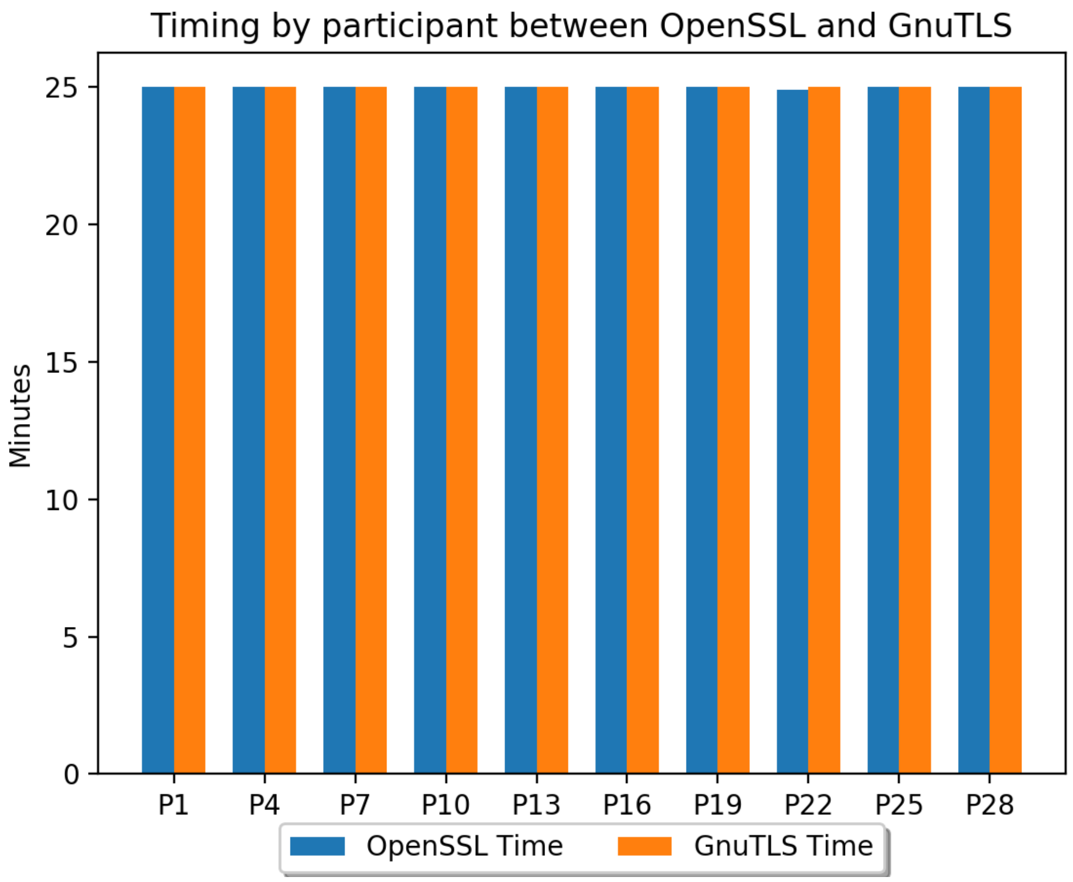


Figure 5.1: OpenSSL vs GnuTLS timing

nine participants (P3, P6, P9, P15, P18, P21, and P27) finished the task in under 25 minutes. P9 and P15 also finished the OpenSSL task in 25 minutes and 21 minutes respectively.

In the end, four participants (P9, P15, P22, and P29) believed they finished the task with OpenSSL before time was up. While seven participants (P3, P6, P9, P15, P18, P21, and P27) believed they finished the task with the SSA before time was up. There were no participants that completed the task with GnuTLS or s2n. In each group, some participants completed the task with OpenSSL but not the second library assigned. We believed this would be the case because the libraries are different from each other. They can be used to perform the same task, but the process is different for each library.

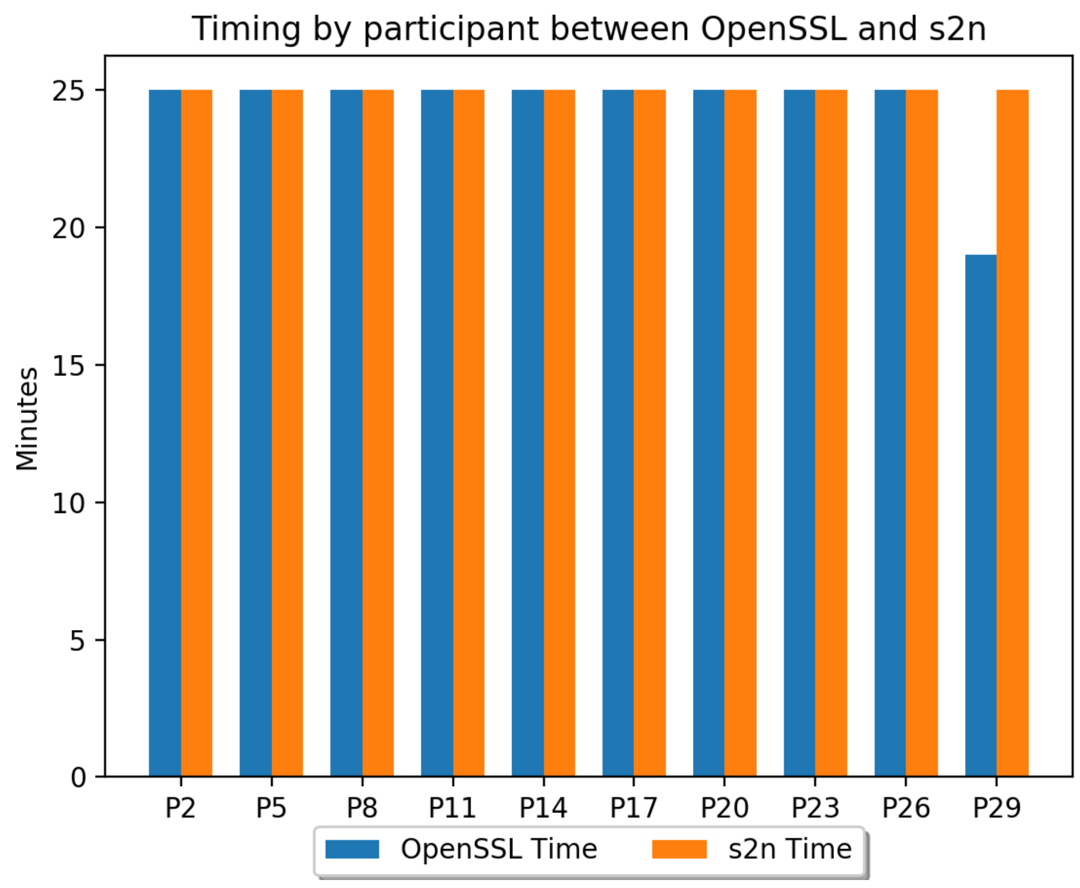


Figure 5.2: OpenSSL vs s2n timing

Many participants who ran out of time were close to establishing a connection. Only one or two minor components were incorrect within their code when they ran out of time that prevented them from compiling the program. We discuss this in section 5.3.1.

5.2 Usability

We measured the usability of the four TLS libraries using the SEQ survey. Higher scores represent greater usability.

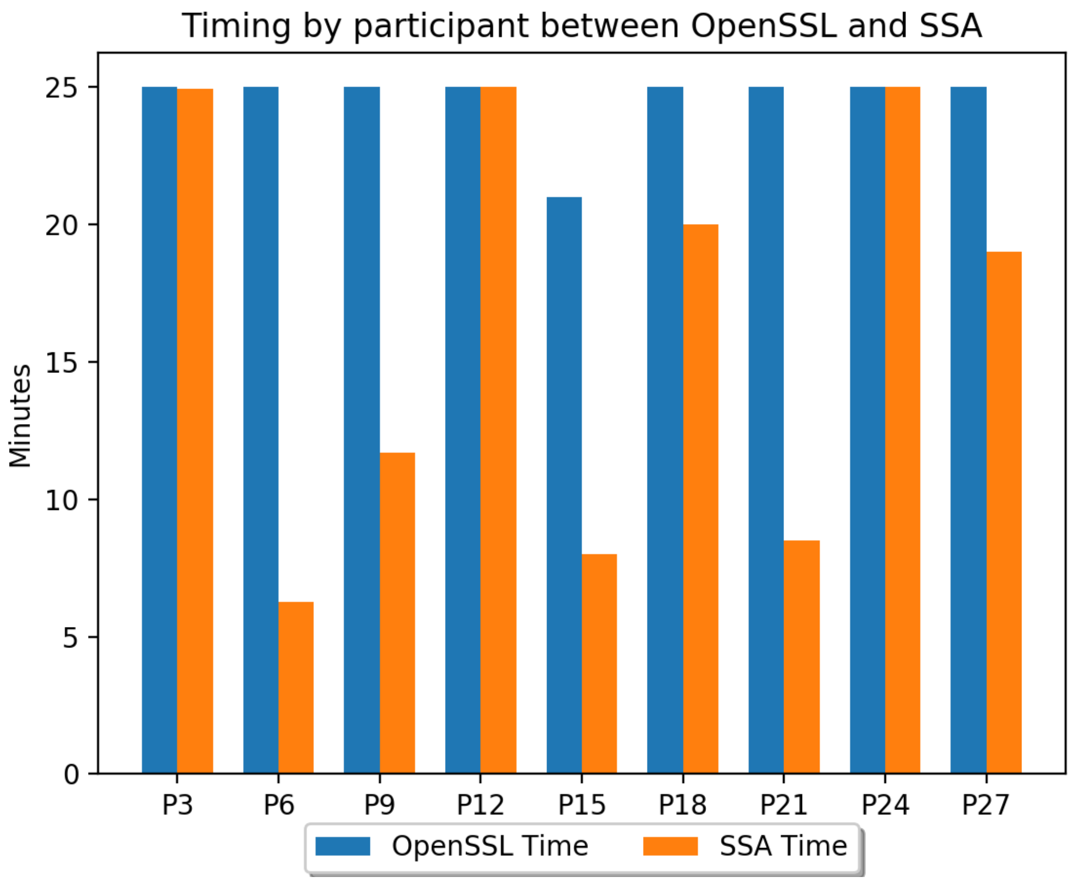


Figure 5.3: OpenSSL vs SSA timing

5.2.1 SEQ Scores

Participants answered the SEQ after completing (or being unable to finish) each of their two assigned tasks. Mean SEQ scores are shown in Table 5.1. The higher the score the more usable the library was to our participants. The distribution of participants' perceptions about the ease of completing the task with their assigned TLS library can be seen in Figure 5.4.

OpenSSL	GnuTLS	s2n	SSA
2.8	3.7	3	5.9

Table 5.1: Mean SEQ Scores

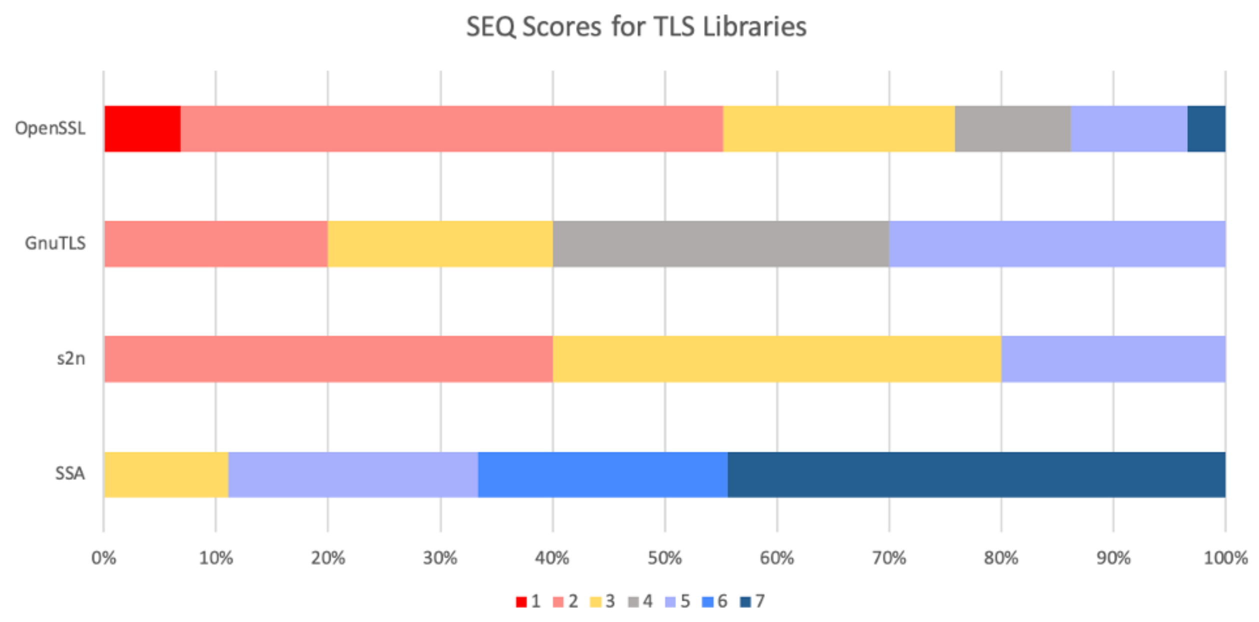


Figure 5.4: SEQ: Ease of the task

Participants rated the SSA most usable followed by GnuTLS, then s2n, and finally OpenSSL. This rating is in part because of the structure of the documentation rather than whether the participant believed they used the library correctly. This can be seen with the SEQ scores participants P9, P15, P22, and P29 gave, which were 2, 4, 3, and 5 respectively. Even though these four participants finished the task using OpenSSL, they still felt it was harder than it should be, rating it low in usability. Our claim is also supported by the fact that no one finished the task using GnuTLS or s2n but rated both of these higher than OpenSSL in usability. Participants commented on how helpful the official documentation was for GnuTLS, s2n, and the SSA. As for OpenSSL many participants did not find the documentation helpful, oftentimes using outside sources for help. See section 5.3.3 for more analysis on participants' thoughts of each library.

Preferences about which libraries they'd use in the future were reflective of how highly the users had rated the usability of the respective systems. The majority of participants preferred to use GnuTLS, s2n, or the SSA over OpenSSL. See 5.4.1 for an in-depth analysis of library preference.

Lastly, participants' ratings were different than our hypothesis from our pre-study. GnuTLS had the lowest rating using Green and Smith's [10] ten principles during our pre-study. This was mainly because it scored poorly on principle 9. Principle 9 deals with keeping protocols up-to-date and one of the libraries GnuTLS relies on is not up-to-date with the latest security features. However, since the participants did not have to install the libraries themselves they would not have come across this problem. This may explain, or this may be part of the reason why GnuTLS was rated higher by our participants during the study.

5.3 Qualitative

After each task, participants were interviewed regarding what they liked, disliked, and what they would change with the libraries they were assigned. Participants' programs were collected to analyze later. In this section, we discuss common themes and ideas observed in participants' responses, as well as common pitfalls discovered during the program analysis. Analysis of interviews and programs were performed by two study coordinators until both agreed on common themes and pitfalls seen.

5.3.1 Program Analysis

There were many common pitfalls participants had that prevented them from compiling their program and setting up a secure connection. A breakdown of every problem along with insights into the use of each library can be seen in Table 5.2. In this section, we discuss the most common problems and possible vulnerabilities a problem may introduce.

The most common pitfall participants had was not changing the port from the port for HTTP (80) to the port for HTTPS (443). This occurred 32 times across the libraries used. The four TLS libraries hard fail in making a secure connection to a server if the port number is not changed. Out of the 18 times the port was not changed with OpenSSL,

Problems	OpenSSL	GnuTLS	s2n	SSA
Never Changed Port	62%(18/29)	70%(7/10)	60%(6/10)	11%(1/9)
Deprecated Method	34%(10/29)	-	-	-
No certificate checking	48%(14/29)	10%(1/10)	-	11%(1/9)
Incorrect Compiling	24%(7/29)	40%(4/10)	80%(8/10)	-
Lost where to start	21%(6/29)	-	-	-
Used server methods	21%(6/29)	-	30%(3/10)	-
Program in endless loop	3%(1/29)	-	-	-
Used command line tools	7%(2/29)	-	-	-
HTTPS used as request	3%(1/29)	-	-	-
POSIX to BIO functions	3%(1/29)	-	-	-
Socket used before its creation	-	10%(1/10)	-	-

Insights	OpenSSL	GnuTLS	s2n	SSA
Copy Pasted Code	62%(18/29)	70%(7/10)	70%(7/10)	78%(7/9)
Used Official Documentation	97%(28/29)	100%(10/10)	100%(10/10)	100%(9/9)
Used Outside sources	86%(25/29)	70%(7/10)	40%(4/10)	11%(1/9)

Table 5.2: Program Analysis: Problems and insights participants had with each library

four participants (P7, P8, P14, and P26) needed only to change the port to establish a connection. Four participants (P1, P7, P13, and P22) out of the seven for GnuTLS would have established a connection if the port was changed. However, P13 also was not linking the library correctly when compiling their program. With s2n port misconfiguration was just one of several issues that prevented participants from connecting. Finally, P24 did not change the port when working with the SSA, thus preventing P24 from making a connection. Only the official documentation for the SSA explained to participants as to why the port number needs to be changed. This helped participants know that the port needed to be changed and is seen from four participants who did not change the port with OpenSSL but did with the SSA. We believe that if the other libraries patterned their documentation after the SSA by providing an upfront description as to why the port needs to be 443 more participants would have changed the port number.

A problem seen the most with OpenSSL, but not the other libraries, was the use of deprecated methods. Many participants found unofficial examples when looking at outside sources such as Stack Overflow, blogs, Github, etc. Even an official example on a Wiki owned by OpenSSL uses a deprecated method. Methods become deprecated because the ciphers within them are not as strong as more recent ones. However, due to some servers and clients only being able to use the old cipher suites, they remain in the library. One participant (P9) that believed they finished the task with OpenSSL acutally did not. P9 used a deprecated method which could give rise to an insecure connection between the client and the server. Out of the four participants that did not change the port when using OpenSSL, two (P7 and P14) used deprecated methods which would have given rise to an insecure connection if they had changed the port. This insight shows the need for correct and up-to-date official examples. We believe that if these examples were available, our participants would not have used the deprecated methods they found in outside sources or in OpenSSL's official Wiki.

Another common error was the lack of participants validating certificates. If a certificate is not validated the connection is encrypted between the client and the server but the client cannot be sure it is speaking to the correct server. This is a serious vulnerability that can lead to a MITM attack. Three participants (P9, P15, and P29) who believed they had performed the task correctly with OpenSSL did not perform this verification. Therefore, the connections these three set up were insecure. There were five (P6, P7, P8 P18, P26) participants who were close to compiling their program that did not do certificate verification with OpenSSL. For the SSA, one (P12) participant did not perform certificate validation and this was the reason they did not complete the task. Unlike OpenSSL, GnuTLS, and s2n, the SSA hard fails when certificate validation is not performed. Thereby a developer using the SSA is saved from accidentally connecting insecurely. OpenSSL, GnuTLS, and s2n fail silently and continue to create the connection even though it is insecure. We believe that if the other libraries followed the SSA's hard fail policy and included an explanation as to why certificate checking should be done, none of the participants would have left it out. The explanation needs to be easily found in the documentation. We suggest the explanation be within example code.

The next common problem participants had was during compilation time. This problem encompassed not linking the library, missing header files and environment flags, and misuse of the GCC compiler. The problem of not linking the library occurred with five participants for OpenSSL, three for GnuTLS, and seven for s2n. As for the problem of missing header files and environment flags this occurred once with OpenSSL, GnuTLS, and s2n with a different participant for each library. Finally, only one participant misused the GCC compiler when using OpenSSL saying that they have not compiled a program with GCC in a long time. With four participants these problems kept them from having their program connect securely. Two participants (P10 and P25) would have finished the task with GnuTLS, and the other two participants (P23 and P26) would have finished with s2n. As can be seen in the Table 5.2 not one participant had trouble with compiling the client with the SSA library. This is because the SSA's documentation provides easy to locate instructions needed to compile the program correctly for the developer. As for the documentation for OpenSSL, GnuTLS, and s2n, this information was not there or scattered throughout the documentation making it hard for the developer to find it. We believe that if the information needed to compile a program using OpenSSL, GnuTLS, or s2n was as easily available as it was for the SSA developers would not have made any of these mistakes.

Finally, three participants experienced unique problems that stopped them from securely completing the task. Two participants' (P6 and P18) problems happened when using OpenSSL. P6's client was stuck in an endless loop while receiving data, while P18's client performed the GET request as an HTTPS request and not an HTTP request. The secure part of an HTTP request comes from using one of these TLS libraries correctly. Lastly, there was one participant (P28) that set the socket to the TLS session before creating the socket when using GnuTLS. If P28 had created the socket first and then assigned it to the TLS session they would have securely connected to Google.

5.3.2 Library Insights

Most of the participants (28/29; 97%) started with the official documentation of OpenSSL but ended up shortly leaving to find example code from another source (25/29; 86%). From these outside sources many participants (18/29; 62%) ended up copying and pasting insecure code into their clients. This code was found on Stack Overflow, blogs, Github, etc. This shows the need for OpenSSL to create usable documentation so users are not quickly leaving the official site and left vunerable to finding incorrect example code. The documentation should also be up-to-date and easy to find. OpenSSL does have a deprecated example, but it is hard to find.

As for GnuTLS, all participants that used it stuck with the official documentation in the end. Participants Googled for an example using GnuTLS only to find themselves back at the official documentation. The only time outside sources were used was to find out how to link the library and what header files to include. More than half of the participants (7/10; 70%) ended up copying and pasting parts of the code from the example found in the official documentation of GnuTLS.

All participants used the official documentation for s2n. Almost half (4/10; 40%) tried to look for examples online but found nothing that would work. This may be because s2n is less widely known. More than half of the s2n participants (7/10; 70%) copied and pasted the official example code with most copying the whole thing. However, all participants were unable to compile their programs because of the poor documentation of how to do so on the s2n Github repository.

Lastly, with the SSA all participants used the official documentation with only P24 using an outside source to Google the record error code for the failing connection. The error was because the port was not changed, but they never caught this even though it points out in the SSA example code to use 443 for HTTPS connections. Most of the SSA participants (7/9; 78%) copy and paste some or all of the code with the remaining participants not copying the code example but following along with it.

During the use of the four libraries, code was copied and pasted 39 times. This suggests that developers in the wild are more likely to copy and paste code from online. Therefore, we suggest that since establishing a secure connection is a common task with a TLS library, all TLS APIs should have an example program for developers to copy and paste into their code. Using this sample code, the developer would need to only pass in the data that needs to be encrypted for their application.

Library	Correct Implementation	Completed Programs
OpenSSL	1	4
GnuTLS	0	0
s2n	0	0
SSA	7	7

Table 5.3: Number of completed programs that were correct

Through our program analysis, see Table 5.3, we saw that only one (P22) out of 29 succeeded in using OpenSSL to connect securely to Google even though three others believed they had done so. As for the SSA, seven (P3, P6, P9, P15, P18, P21, and P27) out of nine participants connected securely to Google. This is the same amount of participants that believed they had completed the task correctly with the SSA. Therefore, other libraries need to follow the organization of the SSA documentation, specifically by guiding developers around these common pitfalls in example code, like the SSA does. We believe that if the official documentation for OpenSSL, GnuTLS, and s2n had this type of documentation during our study, many more participants would have securely completed the task of setting up a secure connection to Google with these three libraries.

5.3.3 Interview Themes

The two study coordinators transcribed the interviews and conducted a thematic analysis. The coordinators compared the results and discussed any differences until they reached agreement. The themes are shown in Table 5.4. In this section we present common themes from participants' answers.

OpenSSL Themes	
Too complicated	8
Not intuitive	16
Needs example code	18
Not prepared to use OpenSSL	7
Documentation not helpful	19
Too much setup	10
Easy to get wrong	2
Needs Tutorials/Getting Started	12

GnuTLS Themes	
Too complicated	1
Needs example code	3
Too much setup	6
Documentation not helpful	2
Documentation well organized	8
Needs Tutorials/Getting Started	4

s2n Themes	
Needs example code	2
Too much setup	3
Liked transparency of documents and example code	8
Table of Contents needed	5
Needs Tutorials/Getting Started	6

SSA Themes	
Liked similarity to TCP calls	6
Documentation was clear/helpful	9

Table 5.4: Common interview themes based on each library

A few participants (8/29; 28%) found the task to be too complicated using OpenSSL.

P7: *"There's so much initialization going on that it was pretty complicated."*

P1: *"You could probably spend a lot of time in OpenSSL and not get it to work."*

Half of the participants (16/29; 55%) found that using OpenSSL was not intuitive. Many felt they would have been lost if they had not found an outside source from which to copy and paste.

> **P7:** *"None of the calls are very intuitive. I don't think I would have been able to figure out how to do this without copy and pasting. And I don't even know if what I was copying and pasting was right."*

> **P15:** *"It wasn't super difficult but if I hadn't have found that example code and basically copied it and modified it, it probably would have taken me forever to figure out how to use OpenSSL. So it definitely wasn't intuitive."*

A little over half of the particpants (19/29; 66%) felt that the documentation for OpenSSL was not helpful.

> **P21:** *"I really did not like their documentation. They don't have an introduction. They just throw you into their world. They don't introduce you to what OpenSSL is and the different data types. How the library is structured. They don't give you an example. They just throw you into the deep end. I'm not going to use it."*

> **P24:** *"The documentation stunk. There was a reason why I took two looks at it and said nope."*

Some of participants felt that OpenSSL (10/29; 34%), GnuTLS (6/10; 60%), and s2n (3/10; 30%) required too much setup to use the library.

> **P8:** *"It would be nice to have an option to further automate the process because setting up sockets is often similar between programs. But it would be nice even here to have that set up and reduce [using OpenSSL] to a couple command calls. This also reduces the chance for developer errors."*

P16: *"It is annoying you have to set up all of the stuff. I feel like there should be a function that does that every time you use [GnuTLS]. One function that will do it all."*

P14: *"The amount of configuration you have to call before using [s2n]. It just seems like you have to do a lot of extra setup involved in the connection."*

Most of those that were assigned GnuTLS (8/10; 80%), as well as those assigned s2n (8/10; 80%), found the official documentation helpful. All the participants assigned the SSA found its official documentation helpful.

P16: *"In the docs there was organization. Which was nice letting me find what I needed. [GnuTLS] has this introduction thing that gave me an understanding of what was going on. Function calls made sense, I knew what I needed to call next."*

P2: *"They had the example code which made it way easier, because I can see how things are supposed to be going. They had usage of their functions with their parameters and things. It just made [s2n] much easier."*

P9: *"I thought that [the SSA] had good documentation, even with the error I had with the include not in the right place they had a part for that in the documentation. I thought that was really helpful to recognize it. It showed they had really gone through the steps of walking through that example, at least that simple example, of what could go wrong, etc. I really liked that."*

Just over half of the participants (6/9; 67%) that used the SSA liked the similarity to TCP calls when setting up a secure connection.

P27: *"SSA uses a lot of the same functions you would use to set up a normal connection so that was kinda nice. It is already somewhat familiar."*

P3: *"I liked that it wasn't too different from the TCP UDP client usage. It was only different in a few places."*

When asked how each library could improve its documentation many participants answered that example code, a getting started page with tutorials, or a table of contents was needed for OpenSSL, GnuTLS, and s2n.

P9: *"Its documentation is hard to use and really hard without some official examples, maybe I just didn't know where to look."*

P22: *"I think if they had simple tutorials of how to get the basics done and where to go from there it would help."*

P11: *"The examples are really cool. Kinda weird I had to scroll to the bottom for the examples. There needs to be an overview of what it is and how to get started."*

When asked the same question about improving the documentation for the SSA, none of the participants had major suggestions for improvement.

P21: *"Honestly its really good. I really like this README it's very straight forward. It tells you this might be somewhere else for you."*

5.4 Discussion

In this section, we discuss additional findings from our study and limitations.

5.4.1 Library Preference

The last interview question participants were asked dealt with their preference of the library they would use outside of the study. Participants' preferences are seen in Table 5.5.

OpenSSL	GnuTLS	s2n	SSA	No preference
21%(6/29)	80%(8/10)	50%(5/10)	89%(8/9)	7%(2/29)

Table 5.5: Library participants would prefer to use outside of the study

A few participants (6/29; 21%) answered they would rather use OpenSSL. The reason for these participants choosing OpenSSL over the second library was because OpenSSL is a well-known library. By being well-known, participants were able to find more outside sources for OpenSSL when compared to the number of outside sources for the other three libraries.

> **P5:** *"I would use s2n over OpenSSL, oh another disadvantage is that there is not very much on s2n at least through Google so finding help there didn't really work well. If there were more stuff out there for s2n I would probably use s2n. But since there isn't much external help, it seems it would be easier to find, even though OpenSSL documentation kind of sucks, it would be easier to find external sources. I mean I would look for an alternative to both entirely but I would probably go with OpenSSL even though I wouldn't like it. "*

> **P26:** *"I would probably use OpenSSL over s2n just because even though it is convoluted in its documentation more people have used it so there are better sources to learn how to use it and also I feel like it has more versatility than [s2n]."*

The participants that choose OpenSSL also felt that it was audited and patched more than the other three libraries.

> **P24:** *"I would probably use SSA although I'm not sure because I would want to check the community size cause it might be a bit small. OpenSSL is used by everyone so it gets looked at from a lot of bigger companies. So if I saw an audit of the code and it said it was fine I would use the SSA, but without more research I would use OpenSSL and figure it out."*

P29: *"I would say definitely OpenSSL. Partly it's not like I don't trust AWS, I just don't want to have to depend on them because they are like a for-profit company. OpenSSL looks like it is more widely used and trusted. If there is a vulnerability I trust OpenSSL will get fixed faster than s2n."*

Most of the participants assigned to GnuTLS (8/10; 80%) preferred it over OpenSSL because of its documentation.

P22: *"I would prefer GnuTLS. It was a lot simpler and you didn't have to worry as much about setting all the secure settings for creating your connection. There is just that one function you had to call and it would do everything for you. It seemed cleaner than OpenSSL and had better documentation."*

P28: *"I'd prefer GnuTLS because of the documentation and I feel like I made a lot more progress on the task with this library."*

Half of the participants (5/10; 50%) assigned s2n said they would prefer to use it over OpenSSL. The most common reason for choosing s2n was its ease of use.

P11: *"It just seemed easier to get into than OpenSSL. It was more intuitive."*

P17: *"I would definitely use s2n over OpenSSL because I would be able to make it work in a reasonable amount of time."*

Lastly, most of the participants (8/9; 89%) assigned the SSA preferred it over OpenSSL. Participants mentioned how easy the SSA was to learn and use.

P3: *"I would prefer SSA because there is less of a learning curve as OpenSSL used different kinds of objects and structs. Another large consideration would be how much I trust these libraries but right now I would prefer SSA."*

P6: *"I would definitely choose to use SSA, in 10 minutes I was able to get a working prototype, whereas with OpenSSL it seemed a bit more clunky, a bit more antiquated and was more difficult to set up."*

Participants also commented on how straightforward the SSA documentation is.

P12: *"I prefer to use the SSA because I was able to make more progress on that and it was way more informative and straight forward."*

P21: *"Oh this one [the SSA]. Because its got documentation. It shows me how to create a client and a server."*

5.4.2 Limitations

Our participants were all highly-educated and young male university students. A heterogeneous sample may provide more insights into the usability of these libraries than what we found.

We required all students to have taken the class at BYU that equips them with a thorough understanding of POSIX and network functions. We did not ask them the grade they obtained in the class, only that they took the class. Therefore, some participants may not have had as thorough of a background as we assumed.

All of our participants were attending BYU at the time of our study. Some of the participants that were assigned the SSA may have known it was created at BYU. This may have created a bias towards the SSA for these participants.

We limited the study to an hour due to cost and user fatigue. During this hour, there was not enough time for participants to use all four libraries, so each was assigned two libraries for the study. In order for the two libraries to be given equal time and provide time for interview questions, participants were given 25 minutes to complete each task. However, the short time limit may have flustered some participants preventing them from finishing the task. This small allotted time frame may have also influenced the participants' answers to interview questions and the SEQ. Though this time limit may have effected participants, we believe it was the best way to avoid a lengthy lab visit as well as simulate project deadlines in the workplace.

Another way we ensured participants were not stuck for an hour during their single lab visit was by giving them two of the four libraries to use. From this limitation, our within-subjects study allows us to only compare statistically OpenSSL's usability to either GnuTLS's, s2n's, or the SSA's usability. We are unable to compare statistically GnuTLS, s2n, and the SSA amongst each other. However, we still learned many themes that make a library usable and common errors developers make when using a TLS library.

Lastly, assigning all participants OpenSSL first may have influenced how participants felt about the second library assigned. This ordering may have also influenced participants feelings towards OpenSSL as well.

Chapter 6

Future Work

Our study shows many problems developers have while setting up a secure connection. It also shows what can make an API more usable for a developer. In this section, we discuss ideas for future research that could further explore TLS library usability.

6.1 Documentation Improvements

Through interview questions participants provided many ideas on how to improve the documentation for the studied TLS libraries. These span adding a tutorial section, example code, a getting started section, and a table of contents section. This feedback can be used to improve the documentation of the TLS libraries. Once the documentation is updated with all of our participants' suggestions, a new study should be performed to determine how the documentation improvements impact TLS library usability. We believe that after these changes to the OpenSSL, GnuTLS, and s2n documentation are made, the new study would show higher SEQ scores for these libraries as well as more participants using TLS correctly. Studies also can be performed in the future that individually test each idea for documentation improvement. These studies can help point out which of the ideas are more effective in helping participants use a TLS library correctly.

6.2 Studying a More Heterogeneous Population

Future work could include a usability study with more diverse demographics, where the population includes a larger age range of participants, different genders, and different levels of education. It would be interesting to see if a more diverse population has much of the same problems and insights as our study participants found.

6.3 Between-subjects Study

In our study, many participants ran out of time on each of the two tasks assigned to them. Therefore, future work should look at increasing task time to one hour. With this increase of task time, the study design should be a between-subjects study using the same measurement tools our study used but only assigning one library to each participant. Increasing the time but only assigning one library will prevent participants from becoming fatigued.

6.4 Professional Developers Study

Finally using the same methodology performed in this study, professional developers should be recruited for a new usability study. This new study will provide insights into the problems professional developers have when using the same four TLS libraries. If a professional developer struggles with the same things as our participants did, it would show that students and professional developers perform the same on secure programming tasks. This finding would show the replicability of the study performed by Acar et al. [6].

Chapter 7

Conclusion

Transport Layer Security (TLS) is a protocol used to provide confidentiality and authenticity for communications between a client and a server on the web. However, there can be security vulnerabilities within TLS connections leading to a loss of confidentiality and authenticity. These vulnerabilities come from TLS library API misuse by developers.

Our work provides several insights into what can make a TLS library more usable. These include:

1. **Containing a minimal set of function calls required to set up a secure connection.**
2. **Including example code within the documentation.**
3. **Supplying a getting started tutorial as well as tutorials on all API functions.**
4. **Possessing well-organized documentation with a table of contents.**
5. **Incorporating transparency in the API documentation, meaning information is easy to find and clearly presented.**

From these insights, we provide suggestions for these TLS libraries. These suggestions will not only increase the usability of TLS libraries but all APIs in general.

7.1 Getting Started Tutorial

All APIs should be designed to make it easy for any developer to perform all the tasks the API was created to do. Usability of an API can be increased by ensuring its documentation is easy to find and follow. Many of our participants said that code examples were needed for usable documentation and some kind of tutorial was needed in the tested libraries. Therefore, code examples need to be found therein that when followed provide the developer with an understanding of how the API is used. These code examples should be found in a “Getting Started” tutorial that covers what the API does and what is needed for a certain task.

Tutorials will naturally draw developers to the correct place in the documentation they should start reading first. Developers do not want to be bombarded with everything the API can do. Participants found a library was too complicated or not intuitive when it provided too much information. A tutorial breaks the documentation into categories providing the developer with the knowledge needed to accomplish their task.

7.2 Easy to Follow Code Examples and Official Forum

We observed that many participants started with the official documentation for a library, but their first attempt to utilize the API resulted in a program that generated compiler errors. This occurred in each of the four libraries but happened most often with OpenSSL, where most participants left the official documentation early in the task to look for outside sources. Developers should not have to go to a third party when figuring out how to use an API.

Many of our participants learned incorrect information from outside sources such as using deprecated calls or no certificate checking within the client program leading to an insecure TLS connection. This can be solved by having well-documented examples allowing a developer to follow along while understanding what each line of code does. Another way to ensure developers do not read outside information is to have an API have its own Q&A forum where those that have created the API answer questions. The Q&A forum provides developers the correct answers to finish the task they set out to do without having to worry if what they found will work. The forum needs to be kept up-to-date. If the forum becomes stagnate it will become a hindrance for developers and lead to incorrect information being propagated into programs.

7.3 Briefly Define Functions in Example Code

Finally, the example code of an API should explain why a function needs to be called to provide security. Explaining will call attention to the needed function, informing the developer to include the function in their program. For instance, if the example code that some participants found warned participants about the implications of not changing the port number or checking a certificate they would be more likely to do it.

References

[1] AES security blog. https://aws.amazon.com/blogs/security/s2n-is-now-handling-100-percent-of-of-ssl-traffic-for-amazon-s3/.

[2] GMP, arithmetic without limitations. https://gmplib.org/.

[3] S2N. https://github.com/awslabs/s2n.

[4] The OpenSSL license and the GPL. https://people.gnome.org/~markmc/openssl-and-the-gpl.html.

[5] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. Comparing the usability of cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE, 2017.

[6] Yasemin Acar, Sascha Fahl, and Michelle L Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 3–8. IEEE, 2016.

[7] Leo St Amour and W Michael Petullo. Improving application security through TLS-library redesign. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 75–94. Springer, 2015.

[8] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 73–84. ACM, 2013.

[9] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 49–60. ACM, 2013.

[10] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security APIs. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[11] Shubham Jain and Janne Lindqvist. Should I protect you? Understanding developers' behavior to privacy-preserving APIs. In *Workshop on Usable Security (USEC'14)*. Citeseer, 2014.

[12] Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar Weippl. "I have no idea what I'm doing"- On the usability of deploying HTTPS. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1339–1356, 2017.

[13] Ritch Macefield. How to specify the participant group size for usability studies: A practitioner's guide. `http://uxpajournal.org/how-to-specify-the-participant-group-size-for-usability-studies-a-practitioners-guide/`.

[14] Brad A Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59(6):62–69, 2016.

[15] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong?: A qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 311–328. ACM, 2017.

[16] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 296–305. ACM, 2014.

[17] Mark O'Neill, Scott Heidbrink, Jordan Whitehead, Tanner Perdue, Luke Dickinson, Torstein Collett, Nick Bonner, Kent Seamons, and Daniel Zappala. The secure socket API: TLS as an operating system service. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 799–816, 2018.

[18] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. An empirical study of API usability. In *2013 IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 5–14. IEEE, 2013.

[19] Girish Maskeri Rama and Avinash Kak. Some structural measures of API usability. *Software: Practice and Experience*, 45(1):75–110, 2015.

[20] Ivan Ristic. *Bulletproof SSL and TLS: Understanding and deploying SSL/TLS and PKI to secure servers and web applications.* Feisty Duck, 2013.

[21] Jeff Sauro. How to find the sample size for 8 common research designs. `https://measuringu.com/sample-size-designs/`.

[22] Jeff Sauro and Joseph S Dumas. Comparison of three one-question, post-task usability questionnaires. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1599–1608. ACM, 2009.

[23] Thomas Scheller and Eva Kühn. Automated measurement of API usability: The API concepts framework. *Information and Software Technology*, 61:145–162, 2015.

[24] Janet M. Six and Ritch Macefield. How to determine the right number of participants for usability studies. `https://www.uxmatters.com/mt/archives/2016/01/how-to-determine-the-right-number-of-participants-for-usability-studies.php`.

[25] Martin Ukrop and Vashek Matyas. Why Johnny the developer can't work with public key certificates. In *Cryptographers' track at the RSA Conference*, pages 45–64. Springer, 2018.

[26] Jing Xie, Heather Richter Lipford, and Bill Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164. IEEE, 2011.

[27] Minhaz F Zibran, Farjana Z Eishita, and Chanchal K Roy. Useful, but usable? Factors affecting the usability of APIs. In *2011 18th Working Conference on Reverse Engineering*, pages 151–155. IEEE, 2011.

Chapter 8

Appendix

8.1 HTTP Client

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#define MAXBUF  1024
#define MSG "GET /index.html HTTP/1.0\r\n\r\n"

int main(int argc, char *argv[]) {
    int sockfd, bytes_read;
    struct addrinfo hints;
    struct addrinfo *ai_list = NULL;
    struct addrinfo *ai = NULL;
    int ret = 0;
    char buffer[MAXBUF + 1];
    const char *host = "www.google.com";
    const char *port = "80";

    memset(&hints, 0, sizeof(hints));

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;

    ret = getaddrinfo(host, port, &hints, &ai_list);
    if (ret != 0) {
        printf("getaddrinfo: error\n");
        exit(-1);
    }

    for (ai = ai_list; ai != NULL; ai = ai->ai_next) {
        sockfd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (sockfd == -1) {
            perror("socket");
            continue;
        }
        if (connect(sockfd, ai->ai_addr, ai->ai_addrlen) == -1) {
            perror("connect");
            close(sockfd);
            continue;
        }
        break;
    }

    if (ai == NULL) {
        fprintf(stderr, "Failed to find a suitable address for connection\n");
        exit(-1);
    }

    sprintf(buffer, MSG);
    ret = send(sockfd, buffer, strlen(buffer), 0);
    printf("Bytes sent %d\n", ret);
    do {
        memset(buffer, 0, sizeof(buffer));
        bytes_read = recv(sockfd, buffer, sizeof(buffer), 0);
        if ( bytes_read > 0 )
            printf("%s", buffer);
    }
    while ( bytes_read > 0 );

    close(sockfd);
    return 0;
}
```

Listing 8.1: Client participants were given to alter to connect securely.

8.2 Interview Questions

1. What did you like about (insert TLS library) and why?
2. What did you dislike about (insert TLS library) and why?
3. Why did you give (insert TLS library) that SEQ score of (insert rating)?
4. What do you think could be done to improve the documentation for (insert TLS library)?
5. (Asked at the end of all tasks) Is there one of these libraries you would prefer to use, and why?

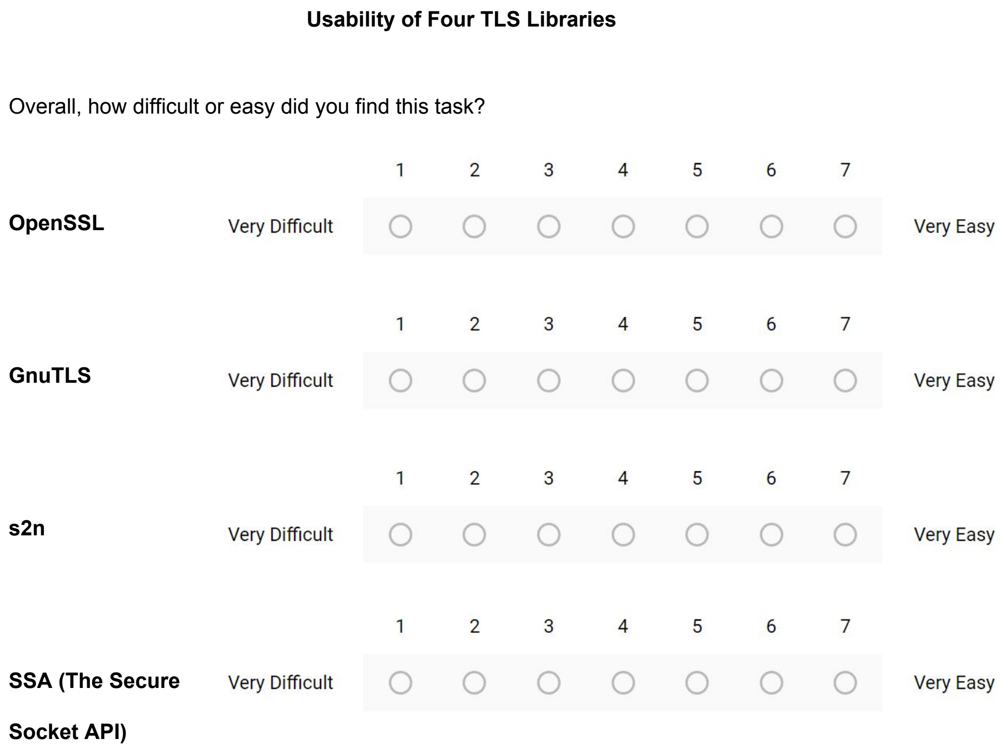
Table 8.1: Questions asked to participant after each task.

8.3 Demographic Survey

1. Please select your gender.
    - Male
    - Female
    - Other
2. Please select the range that includes your age.
    - 18-19
    - 20-29
    - 30-39
    - 40-49
    - 50-59
    - 60+
3. Please select your highest level of education.
    - Some High School
    - High School Diploma or Equivalent
    - Some College, No Degree
    - Associate Degree
    - Bachelor Degree
    - Masters Degree
    - Professional Degree
    - Doctorate Degree
4. How would you rate your programming skills?
    - Far above average
    - Somewhat above average
    - Average
    - Somewhat below average
    - Far below average

Table 8.2: Demographic survey completed before study began.

8.4 SEQ Questionnaire



Usability of Four TLS Libraries

Overall, how difficult or easy did you find this task?

		1	2	3	4	5	6	7	
OpenSSL	Very Difficult	○	○	○	○	○	○	○	Very Easy
GnuTLS	Very Difficult	○	○	○	○	○	○	○	Very Easy
s2n	Very Difficult	○	○	○	○	○	○	○	Very Easy
SSA (The Secure Socket API)	Very Difficult	○	○	○	○	○	○	○	Very Easy

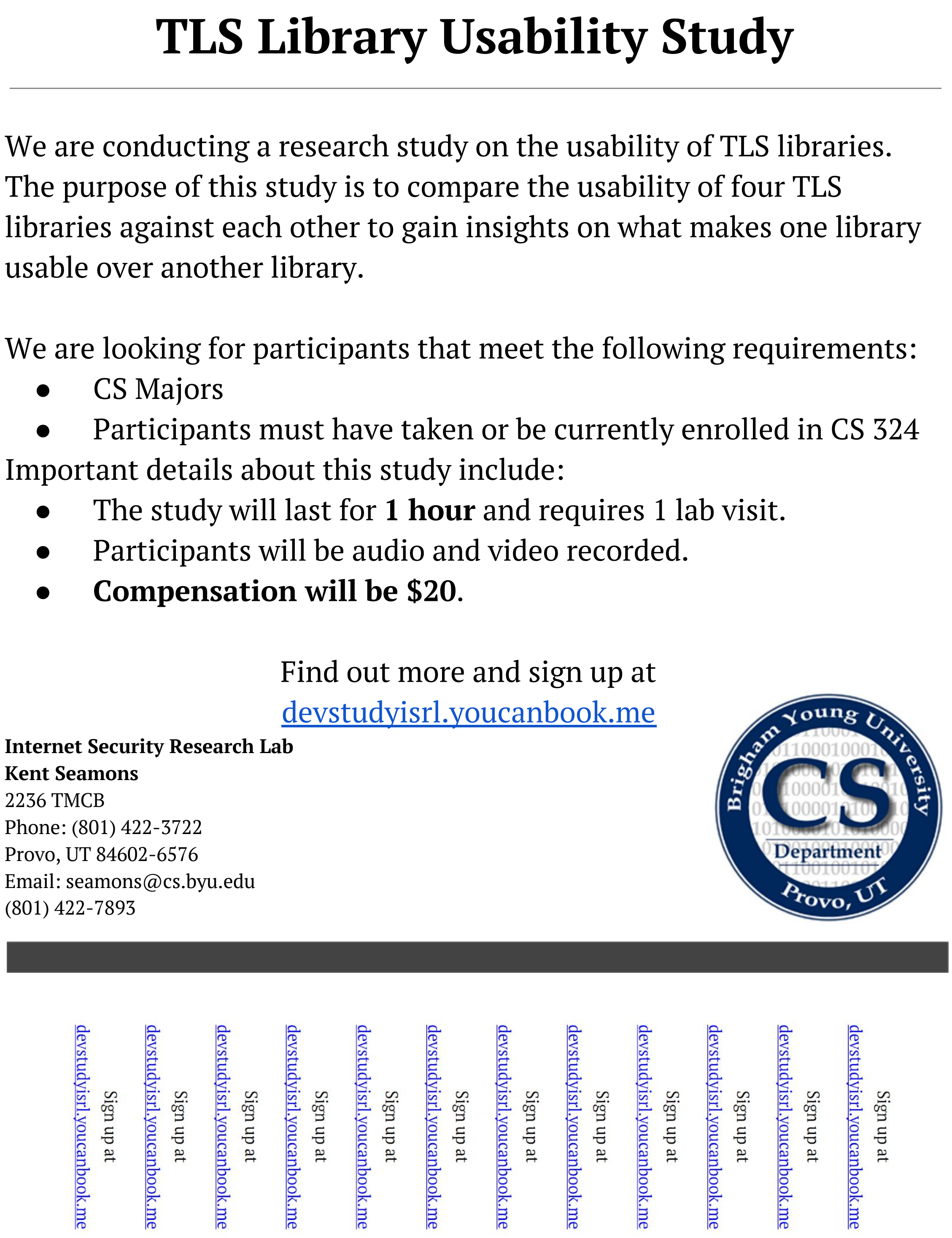
Figure 8.1: SEQ survey given after each task.

8.5 Official Documentation URLs

1. OpenSSL
   - `https://www.openssl.org/docs/manmaster/man3/`
2. GnuTLS
   - `https://gnutls.org/manual/gnutls.html`
3. s2n
   - `https://github.com/awslabs/s2n/blob/master/docs/USAGE-GUIDE.md`
4. SSA
   - `https://github.com/markoneill/ssa-daemon/blob/master/docs/user-documentation.md`

Table 8.3: Bookmarked URLs to a libraries' official documentation.

8.6 Recruitment Flyer



TLS Library Usability Study

We are conducting a research study on the usability of TLS libraries. The purpose of this study is to compare the usability of four TLS libraries against each other to gain insights on what makes one library usable over another library.

We are looking for participants that meet the following requirements:

- CS Majors
- Participants must have taken or be currently enrolled in CS 324

Important details about this study include:

- The study will last for **1 hour** and requires 1 lab visit.
- Participants will be audio and video recorded.
- **Compensation will be $20**.

Find out more and sign up at
devstudyisrl.youcanbook.me

Internet Security Research Lab
Kent Seamons
2236 TMCB
Phone: (801) 422-3722
Provo, UT 84602-6576
Email: seamons@cs.byu.edu
(801) 422-7893

Brigham Young University
CS
Department
Provo, UT

Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me
Sign up at devstudyisrl.youcanbook.me

Figure 8.2: Flyer used to recruit participants for the study.