Detecting Proper SSL/TLS Implementation with Usage Patterns

Justice Adeenze-Kangah and Yuting Chen*
School of Electronics, Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China

*Corresponding author e-mail: jkangah@sjtu.edu.cn

Abstract. The importance of secure communication over the Internet cannot be overstated because of the implications it has for ensuring privacy and safety for users. Much research has been done in this field of study, leading to the creation of the Secure Socket Layer (SSL) protocol and its successor—the Transport Layer Security (TLS) protocol. These protocols serve as a guide for implementing secure connections across the web and as such many libraries have been written to provide Application Programming Interface (API) for their implementation. However, many security risks have arisen due to improper usage of these libraries mainly due to the wrong sequence of API calls. It is also worth noting that the sequence of API calls might differ in certain situations, therefore adding to the level of complexity and increasing the chances of wrongful usage. In this paper, we present a method to detect proper API usage by defining them as usage patterns and testing them against proper implementation models.

1. Introduction

The SSL/TLS protocol is one of the most widely used security protocols on the web for ensuring secured network communication, for example, it is used when browsing websites in the form of Hypertext Transport Protocol Secure (HTTPS) or in the case of sending secured email using Simple Mail Transfer Protocol (SMTP). TLS is the successor of SSL and was developed to solve many of the issues that were discovered in the earlier versions of SSL. The protocol as of the time of writing this paper is at version 1.2 defined in the Request for Comments 5246[1].

Due to its popularity, TLS has been the subject of a lot of research and has been analysed using various different formal methods [7]. However, this analysis focuses primarily on the specification itself. Notwithstanding the potential security flaws found within the specification, more often than not errors arise from the improper implementation of the guidelines of the protocol. Many of the applications that use TLS to secure communication use libraries such as OpenSSL [9] and MbedTLS (formally polar SSL) [8]. In this paper we ask the following research questions:

1. Definition of proper usage: Is it possible to discover and model proper API usage of OpenSSL/MbedTLS automatically?
2. Pattern definition: How can we derive usage patterns that will form the basis of our analysis?
3. Analysis Techniques: How do we design a technique to detect proper usage within a given program?
We address these questions in this paper proposing an approach that uses a well-known graph traversal technique called Depth First Search on the target application to be analysed. The path extracted during the search is stored and then compared to the usage patterns that we obtained from modelling proper API usage from the OpenSSL and MbedTLS library. The main idea behind our technique is leveraging the fact that when a program runs it usually follows an ordered function call. In doing this we are able to convert the problem of API usage into a sequential function call. We are able to then model the problem as a graph, allowing us to traverse and match between the usage pattern and the candidate program. We limit our research to certificate chain validation, but this does not limit the scope of our approach. Our main contributions are summarized in the following:

1. OpenSSL library pattern: We model the correct API usage of a given OpenSSL and MbedTLS library in certificate validation based on call graphs.
2. Evaluation results: We observed notable patterns of proper OpenSSL and MbedTLS implementation within a host of candidate programs.

2. Background

In order to ensure secure communications, an X.509[2] certificate is required by the server and sometimes Communication starts with the client initiating a handshake with the server it wishes to communicate with by first sending a hello message. The client upon initial communication also sends a list of supported cipher suites and in some cases other optional extensions. Upon receiving the initial message, the server also replies with a hello message along with its selected cipher suite and its certificate. Figure 1 shows a summary of the communication model between client and server. The X.509 certificate provided by the server is used by the client to authenticate the client. An X.509 certificate is a digital certificate that follows the widely accepted International X.509 Public Key Infrastructure (PKI) standard for verifying a public key belongs to a specified user, computer or service contained within the certificate itself. The process of generating the certificate is as follows:

1. A key pair first needs to be generated consisting of a public and private key. In some situations, this key pair will be assigned a trusted authority. It is essential that the private key is kept secret and not shared.
2. The second step involves entering information pertaining to the requester (client or server) such as subject name, email address etc.
3. This information is then sent along with a copy of the public key in a certificate signing request to a Certificate Authority(CA). The CA applies whatever policy rules it requires to determine whether the subject name should be bonded to the public key. The signed document returned by the CA is the certificate.
4. With the certificate, the requester is now ready to handle secure connections.

Figure 1. Communication Model.
For the scope of our research, we look at the whole communication process but focus particularly on the certificate validation process. This is because this is the area of communication in which most improper usage occurs. Implementation of this protocol comes by leveraging the API's that have been written in libraries such as OpenSSL. Most of the libraries that implement the TLS protocol have guidelines on the proper usage of the corresponding API. Improper usage could be disastrous because of the potential security risk it poses. In the case of certificate validation, this may lead to situations in which the client is unable to properly determine whether or not it is communicating currently with the intended server. TLS in itself is highly secure but without proper implementation, the client might be communicating with an attacker that is masquerading as another server.

On its own certificate validation can be very difficult to understand and implement, however from a high-level standpoint it can be broken down into 3 distinct steps:

1. **Certificate Chain Validation**: The chain of trust is a succession of certificates starting from the server's certificate leading up to the root Certificate Authority (CA) certificates. In the process of validating the chain, each individual certificate is checked for validity and is signed by the next certificate in the chain. The last but one certificate must be signed by the root CA for the chain to be valid.

2. **Host-name validation**: Every issued certificate must be assigned to a specific hostname or group of hostnames. This means that the certificate provided by a server might be valid, but it might not be the intended server the client wants to communicate with. Therefore, the client does this by ensuring that the name can be matched to either CommonName or SubjectAltName for which the certificate was assigned to.

3. **Miscellaneous Checks**: Other checks may be done in the process of validation such as certificate revocation, which is carried out by checking a certificate revocation list. A large number of applications rely on OpenSSL as the go-to library for handling TLS communications. Developers are prone to making mistakes in the use of the API, but they are not the only ones to blame for this issue as in most cases, the documentation is either poorly written or vague.

**3. Approach**

In this section, we will discuss in more details about the approach and methodology used in evaluating proper TLS implementation in a program. Our matcher works by traversing the callgraph of the candidate software and comparing the path generated after following each node. As mentioned earlier, our approach aims to find vulnerabilities regarding a client's incorrect use of the TLS API in certificate validation.

The candidate software is used as input to profiling software, which generates the call tree. We then match the extracted graph with our proper usage pattern and check for differences between the two of them. The figure below shows a high-level approach used in developing the patterns and analysing the candidate software. We will further discuss how we generate the candidate software model and compare that with the library implementation in section 4.

**3.1. Profiler**

Extracting the callgraph of an input program correctly is essential to our method of analysis, as such we need to discuss code profiling. Code profiling in essence is a form of code analysis that concentrates on measuring instruction execution time, function call frequency and the relationship between callers and callees (a caller calls a function, and a callee is a function called by another). Callgraph profiler is specifically designed to extract information about functions and the relationship between them. The two types of Callgraph profilers are Run Time profilers and Static profilers.

A static profiler extracts the call graph of a program without the execution of the code. On the other hand, run time profiler works by executing the source code. Among the most widely used run time profilers are Prof(Gprof), OpenPat, Valgrind. For the purpose of this paper we shall focus on Valgrind, which has a suite of tools for running dynamic analysis, among its suite of tools is callgrind which records a program's call history and stores this as a callgraph. The decision to use Valgrind comes from the fact that it has a rich set of control options to limit the data returned and speed up profiling
time. In addition, it can be used in conjunction with KCachegrind which is a GUI (Graphical User Interface) that is able to read data collected from callgrind and show it in a graphical interface which can be interacted with.

3.2. Pattern Detection Algorithm

In order to detect proper pattern usage, we leverage the Depth First Search Algorithm in graph theory. By extracting the function call as a graph we are able to traverse all the nodes easily and compare them with the model we have. Algorithm 1. summarizes the implementation of the algorithm:

```
model ::= function list
path := []
proc model.matching(path, model)
for i := 1 to 10 step 1 do
    u := 0
    complete_chain := false
    for q ∈ path
        do if (model[u] = q) do
            do if (model[u] = model.length())
                complete_chain := true
                return complete_chain,
        end
    end
proc DFS(G, v)
    discovered := true
    for u := G.adjacentEdges(v) to G.lastEdge()
        do if undiscovered = true
            path := path + u
        do if u := model.lastFunction()
            model.matching(path, model)
        end
    end
```

Algorithm 1. Matching Algorithm.

The matcher works by traversing the call graph and searching for the first function in the defined model. After finding the first one, it continues down that path to find the subsequent functions that should be called and continues until it reaches the last function.

3.3. Certificates

In cryptology or cryptography, X.509 is the standard defined by the International Telecommunications Union (ITU-T), it is the current standard that defines the format of public key certificates used in TLS/SSL communication. The certificates mainly consist of two parts which are a public key and the identity of the individual or organisation that holds the certificate, which can either be self-signed or signed by a Certificate Authority (CA).

It must be noted that in order for a certificate to be trusted it must have been signed by an authority that is mutually trusted by both communicating parties. This aspect is essential for the certificate path algorithm to work, by allowing for certificates to be signed by multiple intermediary Certificate Authorities, which are in turn signed by other certificate authorities all leading up to a final trusted anchor. If a valid chain of trust starting from the original certificate, intermediary authority’s certificates and ending at the final CA cannot be established then the validity of the certificate cannot be guaranteed.

3.4. Frankencerts

The use of frankencerts [9] has been very effective in uncovering over 200 implementation issues in many libraries such as OpenSSL, NSS, GnuTLS, PolarSSL etc. Therefore, we utilize frankencerts to ensure that we have a large enough sample size of certificates to run our tests on. To test the validity of a TLS implementation, we need to test many different scenarios of both valid and invalid certificates. However, the process of generating multiple certificates is tedious and time-consuming. We, therefore, utilize frankencerts, which involves generating synthetic certificates through a process of sequential mutation.
3.5. Certificate Structure and Chain

In this section we will look at sample structures of real world generated certificates. We will be looking at a production certificate used by www.bing.com and another certificate generated using frankencerts which can be seen in Figure 2 with Bing on the right side. As mentioned above the certificate chain is a succession of certificates starting from the identity certificate (starting certificate) leading up to a root certificate. There may be intermediary certificates in between leading up to the root CA, these intermediary certificates do not matter as long as they are valid and signed by an authority trusted by the client. Using the OpenSSL s_client function we obtain the certificate details as follows.

![Sample Certificate](image)

Figure 2. Sample Certificate.

3.6. There are a few areas to take note of:

1. **Certificate chain**: this part shows all the certificates found in the chain, the entity with the highest number is the root CA at the end of the chain (in this case number 1) and the node with the lowest index 0 is the current certificate we are looking at.

2. **Server Certificate**: this part shows the complete certificate in a non-human readable format, we don't need to pay much attention to this part.

3. **Certificate identity**: this part contains information about the current certificate such as the subject name and issuer of the certificate.

4. **Cypher**: this part of the certificate shows the cyphers used in decrypting the certificate used by the client. Clearly it is easy to see that in a self-signed certificate the chain of trust has the anchor being the same as the identity of the certificate.

4. Implementation and Illustrative Example

The matcher works by traversing the callgraph of the candidate software and comparing the path generated after following each node. As mentioned earlier, our approach aims to find vulnerabilities regarding a client's incorrect use of the TLS libraries API in certificate validation. The candidate software is used as input to profiling software, which generates the call tree. We then match the extracted graph with our proper usage pattern and determine the differences between the two. To better understand our evaluation process, we will now look at an illustrative example in which we first pick a version of OpenSSL that corresponds to an input program. We then confirm our hypothesis of implementation with the library and then use the model to determine whether the API has been correctly deployed in the candidate application.

Different TLS implementations have their own ways of naming functions, the function can be mapped across different libraries because they serve the same purpose. This way we can build a generalized model between different implementations, however this also means that with the evolution of the library with functions being renamed, removed or added the model needs to be updated to prevent giving false results.
4.1. TLS libraries

In this section we will briefly take a look at the TLS libraries that will be used in our analysis. It is important for us to have some background information into the structure of these libraries especially with the naming conventions of the API's. As mentioned earlier we limit the scope of our analysis to two of the most widely used and well-known libraries, namely OpenSSL and MbedTLS.

**OpenSSL.** To better understand our evaluation process, we will now look at an illustrative example in which we first pick a version of OpenSSL that corresponds to an input program. We then confirm our hypothesis of implementation with the library and then use the model to determine whether the API has been correctly deployed in the application.

After obtaining the source code we compile the library on Ubuntu 12.04. The library is compiled in debug mode with debugging options enabled. We need to do this in order to produce debugging information in the operating system's native format. There are a few things to note about the API of OpenSSL, the API function SSL get verify result returns a macro value X509_V_OK when validation is successful. According to the documentation one design flaw of this function and often overlooked by most developers is the fact that the function also returns X509_V_OK when there is no peer certificate, this is because since no peer certificate is returned by the server there are no errors in validation. Therefore, SSL_get_verify_result result should only be used in conjunction with SSL_get_peer_certificate to ensure that the peer certificate is actually presented. It should also be noted that the library provides an API function (SSL_CTX_set_verify) to force this certificate validation, by doing this the connection is immediately disconnected if validation fails. We then generate certificates using frankencerts, the reason behind this is that we need a large dataset of certificates to validate in order to generate a generalized call graph. The first pattern for establishing a secure connection is as follows: SSL_CTX_new() -> SSL_new() -> SSL_connect() -> SSL_get_verify_result().

Another way to establish the connection follows the following order: SSL_CTX_new() -> SSL_new() -> SSL_CTX_set_verify() -> SSL_connect().

**Pattern 1:** As an example, we used Httping, which is a tool for checking the time it takes to make request to a URL, and time it takes for the request to be served and a reply is received. Normally Httping works with non-SSL URLs, but it has a mode in which it can also connect to URLs using SSL/TLS encryption. After running the application and obtaining the call graph, we supplied our tool with the Hypothesis to test the implementation of SSL within this application. Because OpenSSL can be implemented in many different ways, we test it to see if anyone of the proper implementations can be found.

**Pattern 2:** To illustrate the second pattern we profile webfs [3] which is a simple http server for mostly static content. You can use it to serve the content of an ftp server via http for example. Just like Httping by default webfs works over http but can also be switched to an https mode, in which it connects over a secure network. However, unlike Httping we are able to clearly see that it follows the second pattern presented by OpenSSL in validating a server's certificates.

4.2. Literature Review

**SSLINT:** Boyuan He et al [5] present a static analysis tool called SSLINT that does automate and capable analysis of client-side applications. The methodology of their approach is carried out by modelling the problem into one of a program dependency graph (PDG). However, unlike the approach presented in this paper, SSLINT considers the data flows between arguments and returns from the functions. Their approach to analysis has been very successful and in analyzing 381 programs, out of which they were able to uncover 27 programs with previously unknown flaws.

**FLEXTLS:** Is a tool created by Benjamin Barouche et al [3] that works by connecting as either a client program or server. It then sends out programatically generated sequences of TLS messages to properly verify the implementation of the core protocol of a TLS library. It does this by constructing state machines of the implementation and testing out various scenarios to detect abnormalities between the implementation and specification.
4.3. Results

The goal of our approach was to adequately predict if a candidate program met the specifications laid out by the TLS library, and to determine whether or not there are any security flaws. In our analysis we were able to confirm proper implementation in a host of applications and re-affirm flaws detective by less automated methods. Table 1 shows the results of the analysis of Httping, Spamc, Bctoolbox, Umururr and Webfs.

| Program   | TLS Library | Automatic Matching | Pattern Detection |
|-----------|-------------|--------------------|-------------------|
| Httping   | OpenSSL     | Detected           | Confirmed         |
| Spamc     | OpenSSL     | Detected           | Confirmed         |
| Webfs     | OpenSSL     | Detected           | Confirmed         |
| Bctoolbox | MbedTLS     | Detected           | Confirmed         |
| Umururr   | MbedTLS     | Detected           | Confirmed         |

It can be seen that our approach was successful in detecting improper implementations in these programs. However, it should also be noted that after we run our matcher on the programs, we also did manual auditing to ensure that the results were correct. In some instances, such as with Httping, by making changes to the source code, we then rerun the matcher and on the second time, it passed. We, therefore, maintain that our method works well and has the potential to be a handy tool for developers in ensuring proper implementation.

The scope of our approach is not just limited to TLS implementations, in fact it can serve as an implementation detection tool to ensure that developers are following the guidelines set out by the developers or libraries.

5. Conclusion

To correctly detect whether or not an API has been implemented correctly can be a very challenging manual process. The importance of detecting improper usage of libraries that implement SSL/TLS protocols cannot be understated because of its security vulnerabilities. We, therefore, propose our method of analysis in this paper which relies on modeling the correct implementation and using this model to match it against the function call graph. In doing this we convert the problem of analyzing a large program into that of a graph traversal problem. We have shown that by using the library itself we can properly extract usage patterns that can be used to detect whether or not the library has been properly implemented in.

Acknowledgement

This work is sponsored by the 973 Program in China (No. 2015CB352203), the National Nature Science Foundation of China (No. 61572312). Yuting Chen was also partially supported by Shanghai Municipal Commission of Economy and Informatization (No. 201701052).

References

[1] T. Dierks, 2008. The Transport Layer Security (TLS) Protocol Version 1.2.
[2] Gerd Knorr. Webfs, “http://linux.bytesex.org/misc/webfs.html”
[3] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. 2014. FLEXTLS A Tool for Testing TLS Implementations. In 9th USENIX Workshop on Offensive Technologies, WOOT’15.
[4] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL usage in applications with SSLint. In 2015 IEEE Symposium on Security and Privacy (SP). IEEE, 519–534.
[5] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. 2013. On the security of the TLS protocol: A systematic analysis. In Advances in Cryptology–CRYPTO 2013. Springer, 429–448.
[6] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov,” Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations.” in Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 201