Article

A Security Analysis Method for Security Protocol Implementations Based on Message Construction

Jintian Lu ^{1,2}, Lili Yao ¹, Xudong He ¹, Chintser Huang ³, Dejun Wang ¹ and Bo Meng ^{1,*}

¹ School of Computer Science, South-Central University for Nationalities, Wuhan 430074, China; lj45@hotmail.com (J.L.); yaolili93@163.com (L.Y.); hexudong27027@163.com (X.H.); wangdj@whu.edu.cn (D.W.)

² School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510006, China

³ Department of Computer Science and Engineering, University of South Carolina, Columbia, CA 29208, USA; huangct@cse.sc.edu

* Correspondence: mengscuec@gmail.com

Received: 7 November 2018; Accepted: 4 December 2018; Published: 8 December 2018

Abstract: Security protocols are integral to the protection of cyberspace against malicious attacks. Therefore, it is important to be confident in the security of a security protocol. In previous years, people have worked on security of security protocol abstract specification. However, in recent years, people have found that this is not enough and have begun focusing on security protocol implementation. In order to evaluate the security of security protocol implementations, in this paper, firstly, we proposed the Message Construction to Security Protocol Implementation (MCSPi), a message construction method based on application programming interface (API) traces, which automatically generates the constructed client valid request messages. Then, we presented the Security Analysis Scheme (SAS), a security analysis scheme that generates an abstract model of a security protocol server. Next, we proposed a security analysis method to evaluate the security of security protocol implementations on the basis of constructed client request messages generated with MCSPi, corresponding to the server-side response message and server-side abstract model produced by SAS. Finally, we implemented the Security Protocol Implementation Analysis (SPIA) tool to generate client valid request messages and a server-side abstract model to assist in evaluating security protocol implementations. In our experiments, we tested Tencent QQ mail system version 2017 and RSAAuth system and found that RSAAuth is vulnerable and its server has only security checks for user password, while Tencent QQ mail system version 2017 is more secure and has strong security restrictions at server-side besides security checks for user password.

Keywords: protocol security; security protocol implementation; messages construction; API tracing; abstract model

1. Introduction

Security protocols constitute a key part of cyberspace security [1] and are an important method to protect the cyberspace security against malicious attacks. Hence, people are very interested in their security. The classic method for analyzing security protocols is the formal method. However, some researchers have shown that many security protocols proved with formal methods are still not secure in practice owing to incorrect implementations [2]. Apart from that, due to the capability limitation of developers, there may be inconsistencies between security protocol implementations and the abstract specification of security protocols. So, just proof of security properties of a security protocol with formal methods is not enough to provide confidence in its security.

Therefore, verifying security and detecting vulnerabilities of security protocol implementations are significant to cyberspace security [3] and have attracted widespread attention from researchers and developers.

In general, security protocol implementation consists of client-side and server-side implementation. Thus, we can study the security of security protocol implementations based on one of the following four assumptions:

- (1) Do not have client-side implementations and server-side implementations.
- (2) Have both client-side implementations and server-side implementations.
- (3) Have the client-side implementations but not the server-side implementation.
- (4) Have the server-side implementations but not the client-side implementation.

After examining many security applications, especially based on browser/server (B/S) model applications, we found that assumption (3) is suitable for most cases because it is very difficult for researchers to get the server-side implementations in practical applications, but it is relatively easy to get the client-side implementations. Hence, we focus on security analysis with assumption (3). The methods of analyzing the security of a security protocol mainly include program analysis (code analysis), software test analysis, a model extraction method [4], execution trace analysis, and net-trace analysis [5]. Ye et al. [6] analyzed the security of the Facebook login protocol in Android and recovered a specification of the Facebook login protocol based on Client JavaScript implementations, but it is not suitable for PC platforms. Zuo et al. [7] proposed an approach to detect vulnerabilities by modifying the arguments of request messages from client to server. Bai et al. [8] developed AuthScan to recover abstract specifications of security protocols using security protocol client JavaScript implementations; however, AuthScan does not fully dig into the knowledge of cryptographic primitive JavaScript implementations. Kobeissi et al. [9] proposed a model extraction method to transform security protocol ProScript implementations to Applied PI calculus and Blanchet calculus, respectively. Then, ProVerif and CryptoVerif were used to analyze the security of security protocol ProScript implementations, but they also did not pay attention to the internal structures of cryptographic primitive ProScript implementations. Based on the previous typical analysis, we found that the client-side implementations were not fully used to evaluate the security of security protocol implementations and to extract accurately the abstract model from client-side implementations for analyzing security protocol implementations. Therefore, we proposed a security analysis method based on API traces in which we used the knowledge of the internal structures of cryptographic primitive implementations to reconstruct its outputs for evaluating the security protocol implementations. The main contributions are as follows:

- (1) Present the state-of-the-art security analysis method of security protocol implementations and point out that existing researchers have not taken into consideration the internal structures of cryptographic primitive implementations to analyze the security of security protocol implementations.
- (2) Propose the Message Construction to Security Protocol Implementation (MCSPI) method, a message construction based on application programming interface (API) traces. At the beginning, it intercepts and parses messages between client and server. Then, it traces the cryptographic primitive functions in security protocol implementations with API tracing technology. Finally, it reconstructs outputs using out-of-the-box re-execution to replace the located tokens to generate the valid client messages.
- (3) Present the Security Analysis Scheme (SAS), a security analysis scheme that combines MCSPI and the corresponding server-side response message to generate an abstract model of the security protocol server.
- (4) Propose a security analysis method to evaluate the security of security protocol implementations on the basis of constructed client request messages generated by MCSPI, the corresponding server-side response message, and the server-side abstract model produced by SAS.

- (5) Implement the Security Protocol Implementation Analysis (SPIA) to test the RSAAuth system and Tencent QQ mail system version 2017. We found that RSAAuth has just user password verification at server-side, while Tencent QQ mail system version 2017 holds some other restrictions besides user password security verification.

The rest of the paper is organized as follows: the related works are discussed in Section 2, the presented method is introduced in Section 3, the design and implementations of SPIA are demonstrated in Section 4, the evaluation is introduced in Section 5, and the conclusion and future works are reported in Section 6.

2. Related Works

The methods of analyzing the security of security protocols mainly include program analysis (code analysis), software test analysis, model extraction methods, execution trace analysis, and net-trace analysis. At the high level, execution trace analysis and net-trace analysis belong to protocol reverse engineering technology.

About program analysis, Jürjens et al. [10] presented a security verification method for Java implementations, but the correctness of this method has not been proved. Dupressoir et al. [11] presented the analysis methods for authentication and secrecy of C implementations of security protocols based on symbolic model. Bhargavan et al. [12], Backes et al. [13], and Swamy et al. [14–16] analyzed the authentication and secrecy of F implementations based on a type checker of F language.

Execution trace analysis, also called dynamic taint analysis technology, tracks and analyzes the data transformations based on security protocol executions through tracking and labeling transformed data contained in executions and instructions. Then, process operations of execution data are analyzed to infer the specification and state machine of security protocols. This method can be used to analyze noncryptographic and cryptographic protocols. Regarding the inference of message formats of noncryptographic protocols, Lin et al. [17] presented the AutoFormat scheme to extract the message formats of a simple security protocol structure. Cui et al. [18] proposed the Tupni system to infer deterministic finite automaton (DFA) efficiently. On the aspect of inferences about message formats of cryptographic protocol, Caballero et al. [19] developed the Dispatcher, which can locate the corresponding plain text of cipher text. Shi [20] located the corresponding plain texts of cipher texts of security protocols accurately, such as HTTPS, SSH, and TLS protocols. Zhu et al. [21] presented the Ciphertext Filed Identification Approach (CFIA), which can be used to locate the ciphertext boundaries and identify cipher areas, but a limitation is the weakness of the timing sequence protocol.

For net-trace analysis, the core factor is the payloads of security protocols in the process of communications. Firstly, the network payloads of security protocols are intercepted. Next, intercepted payloads are parsed to recover the specification or infer the message formats and key words of security protocols. Regarding the inference about the specification of security protocols, Beddoe et al. [22] introduced a multisequence match algorithm to identify mutable and immutable fields. In order to overcome the weakness in reference [22], Cui et al. [23] developed the Discover system to recover the abstract model of security protocols accurately, but its limitation is low coverage. To address the Discover system's limitation on processing binary protocols, Yipeng developed the Biprominer system [24] and ProDecoder system [25], both of which can recover abstract models of security protocols. The main difference is that cluster methods are used to search key words in ProDecoder system, but the state transformation among key words is used in the Biprominer system. Bermudez et al. [26] presented the FiledHunter system to infer types automatically and to recode the specification of security protocols.

On the aspect of inferring message format and key words of security protocols, Luo et al. [27] proposed the AutoReEngine tool, and its accuracy for inferring message formats is higher than that of Discover, Biprominer, and ProDecoder. Yun et al. [28] developed the Scuritas system to mine semantic information of protocol messages, for which the accuracy was 97.4%. Tao et al. [29] developed the PRE-Bin system to infer binary pieces of security protocols efficiently.

The model extraction method extracts abstract specification models from security protocol implementations. The confidence should be proved with specified tools. Bhargavan et al. [30] proposed an architecture of verifying the security of F# implementations of security protocols, in which an FS2PV tool transforms F# implementations to Applied PI calculus, and ProVerif verifies the security of F# implementations of security protocols. Bhargavan et al. [31] proposed the FS2CV compiler to transform security protocol F# implementations to the CryptoVerif formal model to verify security with the CryptoVerif tool, and the authentication of TLS protocol F# implementations was proved. Aizatulin et al. [32] extracted formal model from security protocol C implementations and authentication, and the extracted formal model was analyzed with CryptoVerif. Zimao et al. [33] proved the authentication of security protocol Java implementations with CryptoVerif.

For software testing, the main goal is to detect the vulnerability of security protocol implementations. Avinash et al. [34] presented a method to detect multiparty web application (MPWA) vulnerabilities. Giancarlo et al. [35] presented the jÄK system to analyze the security of security protocol JavaScript implementations.

3. Solution

In this section, firstly, we propose the MCSPI message construction method on the basis of API trace, in which we use the internal structures of cryptographic primitive implementations to reconstruct its outputs. Then, based on MCSPI, we generate a server-side abstract model of security protocols. The core functions and corresponding methods are shown in Figure 1.

Figure 1. Core functions and corresponding methods.

In Figure 1, we intercept protocol messages using a man-in-the-middle (MITM) proxy; get the internal structure and API traces of cryptographic primitive implementations with API tracing technology; parse the obtained API traces with the algorithm presented in Section 3.1.2; reconstruct outputs to cryptographic primitive implementations by out-of-the-box re-execution, which usually generates the new output of cryptographic functions; and locate message tokens with the sequence match method.

3.1. MCSPI

Using API trace, we propose the MCSPI method. In the beginning, communication messages between client and server are intercepted, which are parsed immediately. Then, the cryptographic primitive implementations in security protocol implementations are traced and the output is reconstructed with API tracing technology. Finally, according to the data of the API trace parse and reconstruction output to cryptographic primitive implementations, legal client request messages of security protocols are constructed. The idea of MCSPI is shown in Figure 2.

Figure 2. The idea of the Message Construction to Security Protocol Implementation (MCSPI).

In Figure 2, *fun* is an implementation function extracted through API tracing technology; *arg* is the original argument of *fun*; *arg_m* is the modified argument for *arg*; *S_g* is the output to the traced function *fun*, which is produced by the re-execution function *fun* with the *arg_m*; *T_n* is the message token of the security protocol; *M* is the original message; and *M_g* is the constructed message. The procedure of MCSPI in Figure 2 is as follows:

- (1) Substitute *arg* with *arg_m*.
- (2) Re-execute *fun* with *arg_m* to generate *S_g*, which is the constructed output to the traced functions. *S_g* is used to substitute *T_n* to generate the constructed messages.

Locate *T_n* (*n* = 1, 2, 3, ...) in parsed message *M* to generate *M_g* by substituting *T_n* with *S_g*.

3.1.1. Intercept and Parse Message

We first use a man-in-the-middle proxy to intercept the security protocol messages. Then, intercepted messages are parsed through identifying message symbols, for example, end symbols, delimiters, and connectors, to locate *T_n* in the original protocol messages.

Definition relation *A*: $T_n \in B_n \in P_n, (T \neq \emptyset, B \neq \emptyset, P \neq \emptyset, n \in N^*)$, where *P* is a set of protocol message pieces, *B* is a set of protocol blocks, and *T* is a set of protocol message tokens. The protocol message consists of the message pieces, which are composed of protocol message blocks. The message block consists of message tokens.

First, security protocol implementations are scanned to extract message symbols, and then, the frequency of the extracted message symbols is computed. After that, these symbols are sorted from high to low frequency to be identified. Finally, message pieces, message blocks, and message tokens are generated.

In general, message symbols mainly include message end symbols, delimiters, and connectors. A message end symbol such as "\r" or "\r\n" indicate the end of protocol messages or message pieces. A delimiter usually delimits different fields and message pieces of protocol messages. The common delimiters mainly include the " ", "\", etc. A connector connects two or more message tokens, for example, "?", "&", and "=", etc.

We take a message of HTTP 1.1 as an example to illustrate the parse of a protocol message in Figure 3.

Here, token parse method by identifying message symbols is shown in (a), and the generated tokens after parsing protocol message in (a) are shown in (b).

The procedure consists of six steps:

- (1) Identify "\r\n" symbols to generate message piece *P*.
- (2) Parse message piece *P* through identifying space symbol " " to produce message token *T₁* and message block *B₁*.
- (3) Parse *B₁* by identifying delimiter symbol "/" to generate message token *T₂* and message block *B₂*.
- (4) Parse message block *B₂* through identifying message connecting symbol "=" to generate message blocks *B₃* and *B₄* and message token *T₇*.
- (5) Parse *B₃* and *B₄* by identifying message connecting symbol "?" and "&" to produce message tokens *T₃*–*T₆*. Note that *T₃* cannot be parsed into "login" and "jsp".

- (6) All of the message tokens T_1 – T_7 in (b) in Figure 3 locate the message token(s) which would be replaced by matching with the original output of traced cryptographic functions.

Figure 3. Net-trace parse.

In order to efficiently parse the intercepted messages, some messages are parsed at the block level and other messages are parsed at the token level. For example, in Figure 3, "127.0.0.1" represents an IP address so it cannot be parsed into message tokens as "127", "0", "0", "1". If it was parsed into tokens as "127", "0", "0", "1", it might cause some bad effects, such as the waste of computing resources and the failure of message token location of parsed protocol messages. For instance, for the message shown in Figure 3, the field of "127" needs to be replaced with the reconstructed output. If the reconstructed output of traced cryptographic primitive implementation is "192", it will be used to replace "127" in the protocol message. After that, the IP address becomes "192.0.0.1". Hence, the service connection between client and server will fail and the message token cannot be successfully located. To avoid it, the special message field or message pieces cannot be parsed at the token level.

3.1.2. Get Internal Structure and Parse API Trace of Cryptographic Primitive Implementation

We propose an API Trace Parse Algorithm (APTA), which is used to parse API traces and build a stack to store the parsed data that is used to reconstruct the output of traced cryptographic primitive implementations. API traces are the input to the ATPA algorithm and are produced by tracing client cryptographic primitive implementations. The ATPA algorithm is shown in Figure 4.

In the beginning, API traces of cryptographic functions are stored in a local log file. Next, the log file (line 1) is traversed to generate API traces API_i , which contains the name and argument of cryptographic primitive implementations (lines 2–3). If API_i is not empty, it is stored to stack (line 6); otherwise, it is removed from API traces. The function name and argument stored in the stack are used to reconstruct outputs of traced cryptographic primitive implementations and to locate message tokens for constructing legal protocol request messages.

```

Algorithm : parsing the API trace
Input : Logs — API trace Logs
output : The stack which store hooked function name and arguments
1 : while !feof < Logs > do
2 :   < f.name, f.input, f.output > ← fread < Logs >
3 :   APIi ← < f.name, f.input, f.output >
4 :   i ← i + 1
5 : while i ≠ 0 do
6 :   if APIi.output ≠ Φ then
7 :     puttostack(APIi.name, APIi.input)
8 :   else
9 :     i ← i + 1
10 : end

```

Figure 4. Application programming interface (API) trace parse algorithm (ATPA).

3.1.3. Reconstruct Outputs to Cryptographic Primitive Implementations

In order to reconstruct outputs of traced cryptographic primitive implementations, we use out-of-the-box re-execution to reconstruct outputs of traced functions, which is usually used to generate the output of cryptographic functions. Firstly, we invoke the function from the stack built by ATPA. Then, we modify original arguments of the invoked function to generate the modified arguments. Finally, we re-execute traced functions with modified arguments through out-of-the-box re-execution to produce reconstructed outputs of traced cryptographic functions.

3.1.4. Location Message Tokens

Message tokens consist of outputs of cryptographic primitive implementations. Therefore, we divide message tokens into two categories. One category is the outputs of cryptographic primitive implementations which do not depend on other cryptographic functions. The other category is the outputs of cryptographic primitive implementations which depend on other cryptographic functions. We mainly consider top-down dependencies of cryptographic primitive implementations, which means that the outputs of previous functions are the arguments of next functions.

The dependencies are shown in Figure 5, in which the arguments of f_n are the output of f_{n-1} .

Figure 5. Dependencies of cryptographic functions f_1 – f_n .

For the first category, in the beginning, we invoke traced cryptographic primitive implementations by name and its arguments from the stack built by ATPA. Next, we re-execute the cryptographic primitive implementations to generate the outputs that are used to match with tokens of parsed message data. Finally, we label the positions of matched tokens that are replaced to generate constructed request messages. The details are shown in Figure 6.

Figure 6. Example of token location in the first category.

In the example shown in Figure 6, f_1 is the traced cryptographic primitive implementation, sg_1 is the output of f_1 with original arguments arg_1 , $Sg_1 \& T_n$ is the Boolean value of message tokens location, and T_n ($n = 1, 2, 3, \dots$) are the message tokens.

In this case, sg_1 is used to match with T_n to locate message tokens and return the Boolean value of $Sg_1 \& T_n$. If message tokens are successfully matched, the Boolean value of $Sg_1 \& T_n$ is true.

For the second category, firstly, we trace cryptographic primitive implementations and build the dependency of these cryptographic primitive implementations by tracing client-side implementations.

Next, we invoke traced functions and re-execute them according to dependency to generate the outputs of traced functions. Finally, we locate the tokens that are to be replaced to generate legal construct messages through matching message tokens with the reconstructed output. The details are shown in Figure 7.

Figure 7. Token location in the second category.

In the example shown in Figure 7, arg_n ($n = 1, 2, 3, \dots$) are the arguments of cryptographic primitive implementations, and sg_n ($n = 1, 2, 3, \dots$) are the outputs of these functions. $sg_1 \& T_n$ is the matched Boolean result between reconstruction output of cryptographic functions and message tokens. In this case, the output arg_n of function f_n is sg_{n-1} , which is the output of function f_{n-1} , and sg_n is used to match with T_n to locate message tokens.

3.1.5. Construct Message

According to the two categories in Section 3.1.4, messages are constructed. In the first category, firstly, we invoke cryptographic primitive implementations and their original arguments from the stack built by the API trace parser. Next, we re-execute the invoked implementations with modified arguments to generate the reconstructed outputs. Finally, the reconstructed outputs are used to substitute the located message tokens to generate constructed protocol messages. The details are shown in Figure 8.

Figure 8. Message construction in the first category.

In the example shown in Figure 8, f_1 is the traced function, arg_m is the modified argument of f_1 , arg_1 is the original argument of f_1 , and T_n is the message token in which T_2 is the located message token. The original message is the intercepted message. In the process of message construction, sg_1 is used to replace T_2 to generate constructed protocol messages after T_n is located. Firstly, we invoke traced cryptographic primitive implementation f_1 with arg_1 from the stack built by APTA. Next, we re-execute f_1 with arg_m to generate reconstructed output sg_1 . Finally, we replace T_2 with sg_1 to generate the constructed message.

In the second category, firstly, we invoke traced cryptographic primitive implementations with its original arguments from the stack built by the API trace parser. Next, we re-execute the invoked first function of dependency, which is fed with arg_m to generate the reconstructed output. The new output is the argument of the next implementation function. Finally, the output of the last traced implementation function is generated using the output of the previous implementation function, which is used to substitute the located message token of the parsed protocol message to generate the constructed protocol message. The details are shown in Figure 9.

Here, we assume that T_2 is the located message token. Firstly, we invoke f_1 and its original argument arg_1 from the stack built by APTA. Next, we re-execute f_1 with arg_m to generate the new output sg_1 as the input of the next function to generate the new output of f_2 function sg_2 . We use the same method to generate the output arg_n of the last implementation function, and arg_n is the reconstructed output to replace T_2 to generate the constructed message.

Figure 9. Message construction in the second category.

3.2. SAS

In this section, we present SAS, a security analysis scheme that combines MCSPI and the corresponding server-side response message to generate an abstract model of the security protocol server. The idea is that if encryption exists in the request message, we think the corresponding decryption should exist in the response message. Thus, we generate an abstract model of the security protocol server based on extracted cryptographic functions, which is shown in Figure 10, where GM_i ($GM_i \in GM$) is a constructed message using MCSPI, RM_i ($RM_i \in RM$) is a response message, PR_i ($PR_i \in PR$) is a result of parsing RM , PR is composed of the cryptographic-related handle result (HR) and the noncryptographic-related results.

Figure 10. Abstract model generation of server.

After successfully parsing response messages, we extract the HR from parsing to generate the abstract model of the server of security protocol. The detail procedure is as follows:

- (1) Send constructed request message GM_1 , which includes one cryptographic primitive, to the security protocol server.
- (2) Intercept response message RM_i by MITM proxy and then parse it to produce the result PR_i .
- (3) Extract HR_i from PR_i .
- (4) Repeat (1)–(4) until the end of the security protocol.

Therefore, the abstract model of the security protocol is generated based on the extracted HR_n .

If request messages contain two or more cryptographic primitives, response messages should contain one or two *HR*. The detailed procedure is shown in Figure 11, where *CP* is cryptographic primitives, and *N-CP* means noncryptographic primitives. The meanings of *GM_i*, *RM_i*, *PR_i*, and *HR_i* are the same as shown in Figure 10.

- (1) Send constructed message GM_i that contains cryptographic primitives to the server.
- (2) Intercept RM_i with the MITM proxy and parse RM_i to generate parse action PR_i .
- (3) Extract HR_i from PR_i that contains several HR , which are shown in Figure 11 as $HR_1, HR_2, \dots, HR_n$.

Figure 11. HR extraction.

3.3. Security Analysis Method for Security Protocol Implementations Based on MCSPI and SAS

A security analysis method for security protocol implementations based on MCSPI and SAS with assumption (3) is proposed in Figure 12, which consists of following two phases.

Figure 12. Security analysis method based on MCSPI and the Security Analysis Scheme (SAS) for security protocol implementations.

First phase: constructed messages are generated based on the MCSPI method, which includes intercepting and parsing protocol communication messages, obtaining the internal structure and parsing API traces, reconstructing cryptographic functions, and locating message token functions.

Second phase: *HR* are extracted through parsing response messages to generate the abstract model of the security protocol server. Security protocol messages consist of request message from client to server and response messages from server to client. The response messages are parsed with the same method presented in Section 3.1.

3.4. Discussion

In order to evaluate the security of security protocol implementations, based on assumption (3), we proposed MCSPI scheme to construct client valid request message. In MCSPI, firstly, we intercept net-traces and parse them. Next, we use API tracing technology to trace cryptographic functions to obtain API traces and dependencies of cryptographic functions contained in client-side implementations. Then, we locate message tokens. Apart from that, we reconstruct the output of traced cryptographic functions with out-of-the-box re-execution. Finally, we replace located tokens with reconstructed outputs to generate constructed client request messages.

In summary, MCSPI has the following three characteristics:

- (1) The analysis objects are the message tokens that are the minimum compositions of protocol messages.
- (2) The internal structures (dependencies) of cryptographic primitive implementations in client-side implementations are used to locate and reconstruct the output of cryptographic primitive implementations, which is powerful for constructed valid request messages.
- (3) MCSPI can deal with both cryptographic and noncryptographic protocols.

In some sense, MCSPI is related to protocol reverse engineering, especially to net-trace technology. In general, net-trace technology is used for the purpose of inferring protocol format [22,23]. For example, PIP [22] inferred protocol format based on the Needleman–Wunsch algorithm [36], and Discover [23] used tokenization clustering and merging technology to infer both text and binary protocol formats based on net-traces. However, on the basis of net-trace technology, the objective in MCSPI is not to infer protocol formats but to generate message tokens to construct client request message of security protocol implementations.

In this sense, MCSPI is similar to AUTOFORGE [7]. However, there is a distinct difference in identifying message tokens between AUTOFORGE and MCSPI. AUTOFORGE uses the Needleman–Wunsch algorithm to identify message tokens, but MCSPI identifies message tokens through matching the output of traced cryptographic functions with message tokens generated by parsing net-traces, which is easier to implement.

Currently, MCSPI is not powerful enough in automatically extracting client-side implementations and analyzing encrypted or obfuscated implementations, which will be enhanced in our future work.

4. SPIA

In order to evaluate the security of security protocol implementations using the methods in Section 3, we designed and implemented the SPIA.

4.1. Design

The architecture of SPIA is shown in Figure 13. SPIA mainly contains an API trace parse module, a net-trace parse module, a token location module, a function reconstruction module, and a server model generation module.

Figure 13. Architecture of SPIA.

The API trace parse module is responsible for parsing API traces and building a stack to store the results of the API trace parse. The net-trace parse module is used to parse the intercepted messages

of security protocol implementations to generate constructed request messages and to extract *HRs* in response messages. *HRs* are used to generate an abstract model of the security protocol server. The token location module provides the function of exactly locating message tokens that are replaced. The function reconstruction module generates the reconstructed outputs of traced cryptographic primitive implementations based on the data stored in the stack built by APTA. The reconstructed outputs are used to substitute the located message tokens to generate constructed request messages. The server model generation module takes charge of producing the abstract model of the security protocol server based on *HRs* in response messages parsing.

4.1.1. Net-Trace Parse Module

The net-trace parse module processes intercepted messages (net-traces) and generates message tokens T_n , which are the inputs to the token location and message construction modules. The net-trace mainly consists of the request and response messages. Firstly, it scans client-side implementations to extract message symbols and divides these symbols into message end symbols and other symbols. Next, it parses net-traces through identifying message symbols to generate message pieces, message blocks, and message tokens. The details are shown in Figure 14.

- (1) Scan client-side implementations to extract the message symbols and build the sets of message symbols.
- (2) Build a set of end symbols and a set of non-end symbols using extracted message symbols.
- (3) Sort the non-end symbols in the order of descendent frequency.
- (4) Get net-traces with MITM proxy.
- (5) Parse net-traces through identifying message symbols to generate message pieces, message blocks, and message tokens.
- (6) Generate message tokens T_n .

Figure 14. Net-trace parse module.

4.1.2. API Trace Parse Module

The API trace parse module is responsible for parsing API traces and building a stack to store results of the API trace parse. The outputs to the API trace parse module are the inputs to the token location and function reconstruction modules. The details are shown in Figure 15.

- (1) Scan the client-side implementations of security protocols and extract cryptographic primitive implementations.
- (2) Trace the extracted cryptographic primitive implementations through API tracing technology to generate the API traces stored in a local log file.
- (3) Get the API traces API_i from log file that contains the function name, arguments, and outputs of the traced cryptographic primitive implementations.

Figure 15. API trace parse module.

Parse API_i with ATPA and build a stack to store the data of API trace parses, which stores the name and argument of parsed API_i .

4.1.3. Function Reconstruction Module

The function reconstruction module generates reconstructed outputs $output_m$ of traced cryptographic primitive implementations, which replace located message tokens to produce constructed protocol request messages. The details are shown in Figure 16.

- (1) Modify arguments stored in stack to produce modified argument arg_m .
- (2) Re-execute the traced cryptographic primitive implementation with arg_m to generate reconstructed outputs $output_m$.

Figure 16. Constructed module.

4.1.4. Token Location Module

The token location module locates and labels the tokens T_n that are replaced when request messages are constructed. The details are shown in Figure 17.

- (1) Invoke the traced cryptographic primitive implementations from the data stored in the stack built by APTA, and modify the original arguments to generate the modified arguments arg_m .
- (2) Re-execute traced implementations to generate the reconstructed outputs $output_m$ of cryptographic primitive implementations.
- (3) Locate and label message tokens through matching message tokens T_n with the output. If successfully matched, label the corresponding token.

Figure 17. Token location module.

In this module, located message tokens are replaced by reconstructed output cryptographic primitive implementations to generate constructed protocol client messages.

4.1.5. Server Model Generation Module

The server model generation module generates the server abstract model after the constructed messages are sent to the server and the server generates the response messages parsed by the net-trace parse module to produce the *HRs*. Using *HRs*, the abstract model can be generated. The details are shown in Figure 18.

Figure 18. Generation of the server model.

4.2. Development

Development environments of SPIA are shown in Table 1.

Table 1. Development environments of SPIA.

Environment	Tools	Languages
Windows 10, ×64 bit	Eclipse 3.7, Java JDK1.7, 64 bit	Java, Java Script

We developed SPIA with Java JDK1.7 (64-bit) in the platform Eclipse 3.7 running under Windows 10 (×64).

Classes JFrame, JPanel, JButton, and JTextArea in the Java language were used to develop SPIA, and spia is the main class of the SPIA project. In UML Diagram, the relations of these classes are shown in Figure 19, where class spia implements class ActionListener. Class JFrame depends on class JPanel, and Class JPanel depends on class JTextArea and JButton.

Figure 19. Relations of classes in developing the Security Protocol Implementation Analysis (SPIA).

The Graphical User Interface (GUI) of SPIA is shown in Figure 20.

Figure 20. Graphical User Interface (GUI) of SPIA.

In Figure 20, net-trace parse phase parses net-traces intercepted by Burp suite, API trace parse phase parses API traces and builds a stack to store parsed data, traced function reconstruction reconstructs outputs of traced cryptographic functions, token location phase locates message tokens that need to be replaced in the process of message construction, and server model generation phase generates the abstract model of security protocols.

5. Evaluation

Considering assumption (3), Tencent QQ mail system version 2017 and RSAAAuth system were analyzed and discussed in detail with SPIA and Burp suite [37]. We found that RSAAAuth system has only security verification for user password, while Tencent QQ mail system version 2017 has strong security restriction besides the security checks for user password. In addition, during evaluation, random strings appeared in figures are encoded, encrypted, and hashed sequences.

Burp suite is a GUI-based tool which consists of proxy, intruder, repeater, sequencer, and decoder functions. Proxy and repeater functions were used in our case analysis. Proxy is able to intercept communication messages of security protocols and repeater can resend the constructed messages to the server of the security protocol and analyze the response messages sent by the security protocol server. In order to intercept communication messages between browser and server, Burp suite and the proxy of browser are set with “127.0.0.1:8080”.

The evaluation using SPIA and Burp suite is composed of six steps (Figure 21).

Figure 21. Analysis process with SPIA and Burp suite.

Step (1). Parse API trace with ATPA to build a stack for storing parsed data after getting client implementations of security protocols. The data in the stack are the inputs to the token location and function reconstruction modules of SPIA. Apart from that, we also generate the dependence(s) of cryptographic functions if there exists dependence(s) among cryptographic primitives of client-side implementations.

Step (2). Generate the message tokens, which is the key factor for message construction. We intercept net-traces with Burp suite. Further, we pass net-traces to the net-trace parse module to generate message tokens, which is the input of the token location module of SPIA.

Step (3). Generate constructed request messages with the message tokens T_n produced by the function reconstruction module with the parsed data stored in the stack and message tokens.

Step (4). Generate reconstructed output of traced functions to replace T_n in step (3).

Step (5). Generate constructed messages through replacing T_n with reconstructed output. Then, send constructed messages to the server with the repeater function of Burp suite. The server sends back a response message, which is intercepted by Burp suite.

Step (6). Parse intercepted response message(s) with the net-trace module of SPIA to generate message tokens. Then, generate the abstract model of security protocols with the server model generation module of SPIA.

5.1. Case One: Tencent QQ Mail System Version 2017

Tencent QQ mail system version 2017 [38] was analyzed because of its worldwide use in digital society.

Because the HTTPS protocol was implemented in Tencent QQ mail system version 2017, the communication messages were encrypted, which were difficult to parse just with SPIA. To address this problem, the certificate of Burp suite was imported to the Firefox browser, which was set to trust the certificate. Therefore, the communication messages of HTTPS could be parsed. In Tencent QQ mail system version 2017 system, its client-side implementations mainly consist of four source code files that are `JsonMsg.js`, `TCapIframeApi.js`, and `c_login_2.js`, and `c_login_2.js` was the implementation file in which we were interested.

For Tencent QQ mail system version 2017, firstly, we extracted `c_login_2.js` from client-side implementations. After that, we obtained API traces and parsed them with SPIA. Apart from that, we generated dependencies of traced cryptographic functions based on `c_login_2.js`, which were used to locate and reconstruct outputs of traced cryptographic functions. Next, we intercepted with Burp suite and parsed it to generate message tokens. Then, we located message tokens, reconstructed the outputs of traced functions, and replaced the located message tokens with reconstructed outputs to generate constructed request messages. Finally, we intercepted and parsed response messages to generate the abstract model of the server to evaluate the secrecy of Tencent QQ mail system.

According to the client-side JavaScript implementation "`c_login_2.js`", dependencies of cryptographic functions were generated, as shown in Figure 22, where the return value of function `u` was encrypted by function "`RSA.encrypt`" with RSA public key. The ciphertext was transformed to an ASCII value by the function `btoa`. The output of function `btoa` was the value of `i.p` in Figure 22, which was a part of the request message.

In Figure 22, "`pt.plogin.armSafeEdit.isSafe`" is the security ActiveX. Dependencies among MD5 functions `c`, `u`, and `i` are $u(i(c(t) + b) + vcode.upper)$, where variable "`vcode.upper`" stores the upper case of the value in `verifycode` and the value of `b` is "`0028`". Arguments of function `c` are the salt value and user password. Return value of `c` is the MD5 value of salt and user password. In function `i`, the argument is the return value of function `c`, and the return value is the MD5 sequence of `b` and return value of `c`. In function `u`, the arguments are the `verifycode` and the return value of function `i`, and the output is an MD5 sequence.

In Figure 23, the Get method was used to send the request message from the client to the server in the QQ mail system. Besides that, the key parts of this request message were some information returned by the server, such as user account u , $verifycode!$ ESF , value of p , salt value and json token- $pt_jstoken$, etc. In Figure 24, function $ptuiCB$ did not receive any successful logon information, such as server's URL, user account, and login type, which means that the request message in Figure 23 was incorrect.

In order to generate message tokens and locate tokens, we parsed the intercepted request message in Figure 23 with SPIA. Part of the result is shown in Figure 25.

Figure 25. Parse result of the intercepted request message.

We matched the original output of the traced functions with message tokens, and the matched message token is the located token in Figure 26.

Figure 26. Result of message token location.

The located token was the value of " $i.p$ " according to Figures 23 and 25, which was replaced by the reconstructed output of the traced functions. In order to reconstruct the output of the traced functions, we re-executed the traced functions (Figure 22) with the corresponding modified arguments to generate the reconstructed output (Figure 27).

Figure 27. The reconstructed output of the traced functions.

We replaced the located token (Figure 26) with the reconstructed output (Figure 27) to generate the constructed request message shown in Figure 28, which was sent to the server.

```

GET /login?u=1029851866&verifycode='ESE&pt_vcode_vl=0&pt_verifysession_vl=cb706f529089182d9bbcf6d7f698ef5113b22503fc1a
ccacc2a5b909dcbfd26c8ed8c1251975fafd26a6da4156ae66451a455cbdd16a6p=nxpfGfMEtTo1cZgEFFYHXKCPQc40rCKOlUdKnyQuBhxdPc
oVFFP1PjDwnn1k73KLE-TRagetiYlTPtFgcgQ472Kzrhuhxz-tPIE9E7v46esHXNqkZx8UW-2WcdVamPmTe1YJRtBC0KJYvgvQavTaibkriRoh
2U019oPnp79C513RQ0MRPQKSETP7E1ldv85MphlpsagfcoVNBvVwVB-Pzo2hnpb8N1u-V9dvtPoEGAWfKZ51f6Uo0etgHakyLArSdBdXc
Byd-S-NKv4CdGq73D8G6k8nn8*4e*WsrSRZn7069FFKXGUXQ-MZhGmvM0eYwEQ_4pt_randsalt=2c2pt_jstoken=1279074848dui=https31A2F
I2Fmail.qq.com?2cFegi-bin?2fegin3Fvt3Dpassport426vwt3Dwpt326ft3Dloginpage426target3D426account3D1029851866&ptre
direct=14h=12f=14g=14kfm_uui=14ptlang=2052&action=4-41-15181783657286js_ver=102334js_type=1&login_sig=aSaZEOp'dgdUBK
Lw246GbYleKM12'D6KwM2voPPRw2Oy1A*W0gANmQfu4JnW&pt_uistyle=25&aid=5220057056&aid=4& HTTP/1.1

Host: ssl.ptlogin2.qq.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: /*/*
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Referer:
https://xui.ptlogin2.qq.com/cgi-bin/xlogin?appid=5220057056&aid=4&s_url=https://mail.qq.com/cgi-bin/login?vt=passpor
t426vwt=vpvt326ft=loginpage426target=4style=2561ow_login=1&proxy_url=https://mail.qq.com/proxy.html?need_qr=0&hide_bor
der=1&border_radius=0&self_reguri=https://zc.qq.com/chs/index.html?type=1&app_id=110057?register_feedback_link=https
://support.qq.com/discuss/3501_shtml&css=https://res.mail.qq.com/ch_CN/html/edition/style/ptlogin_input24e6b9.css
Cookie: pt_guid_sig=be1497dac1a6405a8733aac5b7725560dc41dd935d8cfe9cac6b4ee81cf8affc; pgv_pvi=1346127872;
ptui_loginuin=1029851866; pt2gguin=1029851866;
pt_recent_uins=19ee3f589991d9cbe6b57ca89e8dd1c25943c7d8c10a697ee612a289d2bc4471e0d52bf3f2b528db9237dccc62033f5716ec9
0cf577ec04; RK=1d41ahXbQ; ptcz=150001ec2b3bec0c9825d86f014c48f18429dd9e4b48097168f6dc6371e0a6d;
pt_login_sig=aSaZEOp'dgdUBKwLw246GbYleKM12'D6KwM2voPPRw2Oy1A*W0gANmQfu4JnW&pt_clintip=b66d22a97967e10;
pt_serverip=731100abf0e2f20f9; pt_local_token=-1002450120;
uikey=lc07a5ef0cb0994f274bd3023131d25efae757f798cfc8f4679ac96b2ef98594; pgv_si=ce928114689;
qrsig=dOfBS0bV1Y-LD6WNPfY2BgH-6V8312cmldhJvXn-FpwrOHixp7fo12PMXayVtO8ft; confirmuin=0;
ptdrvs=IV5xjd9LBWtPjcFOlYxZiHc-BDYAHsALhANvUuWFO;
ptvfssession=cb706f529089182d9bbcf6d7f698ef5113b22503fc1accacc2a5b909dcbfd26c8ed8c1251975fafd26a6da4156ae66451a455
cbdd16a; ptisp=edw; ETK=WdQSL0F82OLw5tXfj5pHsRCJ21eNkPMLd08YEtixCqNTQuwq1-ZdgW05bPzDqjN5Q2lul?js81Y1E11IUGWFW_
;

```

Figure 28. The constructed request message.

The corresponding response message was sent to the client after the server received and processed the constructed message, which was intercepted by Burp suite (Figure 29).

```

HTTP/1.1 200 OK
Date: Fri, 09 Feb 2018 12:14:04 GMT
Content-Type: application/javascript
Content-Length: 557
Connection: close
Cache-Control: no-cache, no-store, must-revalidate
Expires: -1
P3P: CP=CAO PSA OUR
Pragma: no-cache
Server: Tencent Login Server/2.0.0
Strict-Transport-Security: max-age=31536000
Set-Cookie: pt2gguin=o1029851866;Expires=Tue, 19 Jan 2038 03:14:07 GMT;Path=/;Domain=qq.com;
Set-Cookie: ETK=/;Path=/;Domain=ptlogin2.qq.com;
Set-Cookie: uin=o1029851866;Path=/;Domain=qq.com;
Set-Cookie: skey=@er1xY63wJ;Path=/;Domain=qq.com;
Set-Cookie: superuin=o1029851866;Path=/;Domain=ptlogin2.qq.com;
Set-Cookie: supertoken=1097696026;Path=/;Domain=ptlogin2.qq.com;
Set-Cookie: superkey=1j3CB89WKEEY2uXjdhrEgUesVVIEjLY7F1TCtM-IIE_ ;Path=/;Domain=ptlogin2.qq.com;HttpOnly;
Set-Cookie:
pt_recent_uins=43394e9e1c37e21f98ab7d5efdf1d5f797ab24f2a9b4fb5f5f94a6b12bfe46bdc130a520834bb9d035436a9040ed57c3d5ef9b4a22
a5ac6;Expires=Sun, 11 Mar 2018 12:14:04 GMT;Path=/;Domain=ptlogin2.qq.com;HttpOnly;
Set-Cookie: pt_guid_sig=c92a0532a01e9bb69c46af9dbb7101bd4583a0a6a5c2ab21fc85cec7e76b0e031;Expires=Sun, 11 Mar 2018
12:14:04 GMT;Path=/;Domain=ptlogin2.qq.com;
Set-Cookie: ptisp=edu;Path=/;Domain=qq.com;
Set-Cookie: ptnick_1029851866=6a696e74;Path=/;Domain=ptlogin2.qq.com;
Set-Cookie: ptcc=150001ec28b3ec6c89825d86f014c48f18429dd9efbac8097168fdc6371e0a0f;Expires=Tue, 19 Jan 2038 03:14:07
GMT;Path=/;Domain=qq.com;
Set-Cookie: ptccz=;Expires=Thu, 01 Jan 1970 00:00:00 GMT;Path=/;Domain=ptlogin2.qq.com;

ptuiCB('0','0','https://ssl.ptlogin2.mail.qq.com/check_sig?pttype=1&uin=1029851866&service=login&modirect=0&ptsigx=2de54bb7
f7f7ae1b2d0f3292ff3c0a6e47a0d503ac731eb3bdc3f7641d17ef587be8fb48420771f8c91e1f75cd26af18d6572e723f00597f60dfb04c1d3c46s
url=https3v3A2F2Fmail.qq.com?ptcc=Fgci-binV2Flogin3FVt33passport2e6vm3dWptv2ef7t3dloginpage2t6target3d426account3d10298
51866&f_uid=4ptlang=2052&ptdirect=101&aid=522005705&daid=46j_later=0&low_login_hour=0&regmaster=0&pt_login_type=1&pt_aid
=0&pt_aid=0&pt_light=0&pt_3rd_aid=0','1','00000000','j3int')

```

Figure 29. Intercepted response message.

As shown in Figure 29, function ptuiCB received some data, in which “*jint*” was the nickname of the user “1029851866” was the register account, and “[httpsmail.qq.com](https://mail.qq.com)” was the URL of the server. This means that the constructed request message was correct.

Accordingly, the abstract model was generated based on the parsed response message and client-side implementations, as shown in Figure 30.

As shown in Figure 30, intuitively, firstly, the value of p in the constructed message was decrypted when the server received the constructed messages. Then, the MD5 values in sequences of p were verified. Finally, the server generated response messages and sent those to the client. Therefore, from Figure 30, we are convinced that Tencent QQ mail system version of 2017 will verify the encrypted and hashed sequences of the user password at its server-side.

Figure 30. Generated model of the QQ mail system server-side.

Furthermore, we combined the constructed messages, the corresponding response message sent by the server, and the server-side abstract model and we found that Tencent QQ mail system version 2017 has other strong security restrictions for login attempts at server-side besides security checks for user password, such as check valid session time and the times of login attempts. In our test, the server directly returned some warning messages, such as “login timeout”, “too many error logins, please try later”, and so on. Apart from that, the server updated the verification code when there were many login attempts with constructed request messages in a certain time window.

5.2. Case Two: RSAAuth System

The RSAAuth system, developed by us, provides user authentication from server to client. In the RSAAuth system, the client encrypts the user password with an RSA public key of the server, and then encodes it with Base64 to generate the sequence related to the password sent to the server. The server decodes the received sequence and then decrypts the decoded sequence with its RSA private key of the server. In the RSAAuth system, client-side implementation mainly contains DBAccess.java and UserBean.java file. Server-side implementation mainly contains MsgP.java file.

We extracted the UserBean.java file from client-side implementations. Then, we generated dependencies of cryptographic functions based on extracted implementations, which were used to locate message tokens and reconstruct outputs of traced cryptographic functions. Meanwhile, we intercepted net-traces with Burp suite and parsed them to generate message tokens. Next, we located message tokens and reconstructed outputs using out-of-the-box re-execution. After that, we used the reconstructed outputs to replace located message tokens to generate constructed request messages. Finally, we intercepted and parsed response messages to generate the server model of security protocols to evaluate the security of RSAAuth. We found that RSAAuth is vulnerable to password brute force attack.

The details of the procedure are as follows: the dependencies among cryptographic functions were generated (Figure 31) based on the UserBean.java file extracted from the client-side implementation.

In Figure 31, “ljt0” is the encrypted object. Firstly, “ljt0” was encrypted through *Cipher.init()* and *cipher.doFinal()* to generate the ciphertext “[B@276fa677]”. Then, ciphertext was encoded with Base64 to generate the output “JV1OivWWYW*****”. The dependencies were mainly used to locate message tokens and reconstruct outputs of traced cryptographic functions.

In order to get net-traces, firstly, registered user account information was used to log on the system. Then, communication messages were intercepted by Burp suite. The intercepted request and response messages are shown in Figures 32 and 33, respectively.

```

cipher.init(Cipher.ENCRYPT_MODE,
    Sun RSA public key, 1024 bits
    Modulus: 9174669946*****
    Public exponent: 65537
);

byte[] [B@276fa677] = cipher.doFinal( ljt0 );

String JVIOivWWY***** = new String(Base64.encode([B@276fa677]));
    pwd

```

Figure 31. Dependencies of cryptographic functions in RSAAuth client-side implementations.

```

POST /tomcattest/LoginAction.jsp HTTP/1.1
Host: localhost:8081
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0
.9,*/*;q=0.8
Accept-Language:
zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3
,en;q=0.2
Referer:
http://localhost:8081/tomcattest/Login.jsp
Content-Type: application/x-www-form-urlencoded
Content-Length: 195
Cookie:
JSESSIONID=DCA35B91586E0705E17BEB1B98CF296C
Connection: close
Upgrade-Insecure-Requests: 1

username=ljt1&password=mNPQSULEpu4Qdt1pD6uTJxqZyTok
l2hjh2mFY9Dh1GLw0oHQSNtQMhI243zPBsJm/lpQz9Eifbbcu
iLXw2TRDv8X3drsxMdxHRqL1EReH++4JrUbJbfbkbaq6uWhcvm
IqVPkemOZic7VSoQkeNKkHKAA4ZEjjvkrDm+UfbF4ZO=

```

Figure 32. Request message.

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=utf-8
Date: Wed, 21 Feb 2018 10:34:32 GMT
Connection: close
Content-Length: 58

```

□□□□□□□□□□□□□□□□

Figure 33. Response message.

As shown in Figure 32, the *Post* method was used to send request messages from the client to the server in the RSAAuth system. The key part of the message is boxed in Figure 32. Afterwards, the response message shown in Figure 33 was parsed by SPIA.

To locate message tokens, we parsed intercepted request messages (Figure 32) to generate message tokens. Then, we located message tokens which were to be replaced, as shown in Figure 34.

Figure 34. Result of token location.

The value of the password was the message token that was replaced according to Figure 32. We reconstructed outputs of traced cryptographic functions using SPIA to replace the located message token. An example of the output is shown in Figure 35.

Figure 35. Reconstructed output of traced cryptographic functions.

We replaced the located token with the reconstructed output to generate the constructed request message (Figure 36).

```
POST /tomcattest/LoginAction.jsp HTTP/1.1
Host: localhost:8081
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0
.9,*/*;q=0.8
Accept-Language:
zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3
,en;q=0.2
Referer:
http://localhost:8081/tomcattest/Login.jsp
Content-Type: application/x-www-form-urlencoded
Content-Length: 194
Cookie:
JSESSIONID=DCA35B91586E0705E17BEB1B98CF296C
Connection: close
Upgrade-Insecure-Requests: 1

username=ljtl&password=VY9I43THq2tTfDBkGNn1PKR22Wta
u9wN7Ezb4Zt7c2rrAvr897/HhemlTCvgFN44HnUtjqD+s95z8H
UZXUvYtVztHhuovkECQlyLslrDHZxKjtSkDxdDWFCf4XIUh8
SUjAuWqMOLxBDfy2u/x8CDz9H48GAnKXCTIxSBHmKg=
```

Figure 36. Constructed request message.

The server generated and sent a response message (Figure 37) to the client after the constructed request message was sent to the server.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=utf-8
Date: Wed, 21 Feb 2018 10:36:54 GMT
Connection: close
Content-Length: 25

welcome!
```

Figure 37. Response message for constructed request message.

The response message in Figure 37 was parsed by SPIA. The “welcome” was found, which means that the constructed request message was correct.

Finally, the abstract model of the server of security protocols was generated based on the parsed response message, which is shown in Figure 38. It was consistent with the server-side implementations in Figure 39.

Figure 38. Generated server-side abstract model of the RSAAuth system.

```

Decode(String JVIOivWWY***** )
sun.misc.BASE64Decoder decoder = new sun.misc.BASE64Decoder();
[B@276fa677] = decoder.decodeBuffer( JVIOivWWY***** );
cipher.init(Cipher.DECRYPT_MODE, privateKey );
byte[] str[pwd] = cipher.doFinal( [B@276fa677] );

```

Figure 39. Dependencies of server-side security-related implementations.

As shown in Figures 38 and 39, intuitively, the server decoded the request message, and then decrypted it. Generally, the server checks the decoded and encrypted sequences of the user password at its end.

Furthermore, from our experiment, we found that RSAAuth only held the security check for the user password when login attempts occurred, and this check does not guarantee that the system is secure enough. So, it is vulnerable to many attacks, such as MITM attacks, replay attacks, and, especially, password brute force attacks.

5.3. Discussion

In Sections 5.1 and 5.2, we analyzed the Tencent QQ mail system version 2017 and the RSAAuth system and found that the Tencent QQ mail system version 2017 held some security restrictions in a certain time window besides account password security check, such as check valid session time and the number of login attempts. However, the RSAAuth system was vulnerable and it did not hold some security restrictions besides password security check at its server-side. The results are shown in Table 2.

Intuitively, our work is similar to existing application dialogue replay systems, such as PIP [22] and RolePlayer [39], which replay the server through aligning net-traces of security protocols and identifying and mutating some special field for application dialogue replay. Furthermore, our work is related to protocol reverse engineering, especially the ProDecoder [25] and FiledHunter [26], which focus on searching keywords and recovering message formats of security protocols based on net-traces

without using client-side implementations. It is worth noticing that none of these systems focus on internal cryptographic primitive implementations of security protocols, which is the main concern of our work.

Table 2. Results for security analysis of Tencent QQ mail system version 2017 and RSAAuth.

Systems	Client-Side Implementations	Server Abstract Model	Results
QQ mail	$p \rightarrow \text{salt} \rightarrow 3 \text{ times MD5} \rightarrow \text{rsaencrypt} \rightarrow p'$	$p \rightarrow \text{decrypt} \rightarrow \text{verifyMD5}$	Less vulnerable
RSAAuth	$\text{Pwd} \rightarrow \text{rsaencrypt} \rightarrow \text{base64encode} \rightarrow M$	$M \rightarrow \text{crypt} \rightarrow \text{decode}$	Vulnerable

In particular, SPIA is similar to AUTOFOGRE [7], which discovers vulnerabilities for Android applications, while SPIA addresses the Windows platform to evaluate the security of security protocol implementations. It is worth noticing that both SPIA and AUTOFORGE take the internal structures of cryptographic primitive implementations into account to evaluate the security of security applications. However, there are several distinct differences between SPIA and AUTOFORGE; for example, regarding identified message tokens, AUTOFORGE uses the Needleman–Wunsch algorithm to identify message tokens, while SPIA identifies message tokens through directly matching output sequences of traced cryptographic functions with parsed message tokens, which is easier to implement and takes less computational cost. For the output, AUTOFORGE merely outputs the constructed request messages, while SPIA outputs both constructed valid request messages and the server-side abstract model, which is cogent enough to assist in accurately evaluating the security of implemented security protocols and to check the security restrictions when login attempts occur. As for the computational costs, SPIA and AUTOFORGE are mainly amortized by the token identified algorithm and message construction using out-of-the-box re-execution based on internal cryptographic primitive functions. For both SPIA and AUTOFORGE, it is not easy to accurately compute the cost of message construction because of the different cryptographic primitives. About the computational costs of the token identified algorithm, it is intuitive that the time and space complexity of the Needleman–Wunsch algorithm and our method are $O(n * m)$, in which n and m are the length of two evaluated sequences. In practice, AUTOFORGE compares two protocol messages to identify the message token. However, our method matches the output of the cryptographic primitive with the protocol message to locate message tokens, in which the length of output of the cryptographic primitive is much shorter than a protocol message. Therefore, in this sense, SPIA has less computational cost than AUTOFORGE.

In order to mitigate existing attacks, such as replay attacks and password brute force attacks, some technologies can be used, such as strong secure server-side restrictions, limitations on times of login, timestamps, challenge–response, and CAPTCHA [40]. The number of login attempts can be limited in a certain time window. Based on our test, we conclude that the RSAAuth system has no such technology in its server-side, so constructed request messages can be used to attempt to login to RSAAuth many times without any warning messages sent out. Nevertheless, Tencent QQ mail system version 2017 holds some security restrictions for login attempts, such as limit session time window and the number of login attempts. Meanwhile, unlike CAPTCHA, this technology does not influence the user’s experience. Timestamps and challenge–response are effective ways to prevent replay attacks and online password brute force attacks.

6. Conclusions and Future Work

Secure implementation of security protocols is a hot issue in the security protocol and code security fields. In this paper, firstly, we presented the MCSPI method to construct client valid request messages. Then, we proposed the SAS scheme to generate a server-side abstract model of security protocols. Next, we introduced a security analysis method to evaluate the security of security protocol implementations through combining constructed client request messages generated with MCSPI, corresponding server-side response messages, and server-side an abstract model produced by SAS. Further, we implemented SPIA to evaluate the security of security protocol implementations. Finally,

we tested Tencent QQ mail system version 2017 and RSAAuth and found that Tencent QQ mail system version 2017 is more secure because it holds some security restrictions when login attempts occur, such as user password security check, login time window limitation, and number of login attempts restrictions. However, RSAAuth is vulnerable because it only holds a user password security check, and this check does not make the system secure enough.

Using our current methods, we can evaluate the security of security protocol implementations in security applications. Owing to the time limitation, we only present an analysis by comparing existing related works with our scheme, which are expressed in Section 5.3. However, in the future, we will make a quantitative analysis for computational costs. Still, there are several limitations to our current method. For example, the client-side implementation automatic extraction is not at a high level, and encrypted or obfuscated implementations cannot be done efficiently. Therefore, we will keep our focus on the following two issues in future works:

- (1) Universal methods or tools to extract, at a high automatic level, the security protocol implementations from web applications are necessary. The precondition of security analysis is to obtain security protocol implementations. However, the work is currently mainly performed by hand. Although there are some automatic methods for extracting security protocol implementations, these methods are not suitable for all security systems. Therefore, universal methods or tools that can extract the implementations of security protocols from web applications at a high automatic level are important.
- (2) So far, many client-side implementations have been obfuscated or encrypted to enhance the security of web security applications. Hence, handling encrypted or obfuscated client-side implementations is our core work in the future.

Author Contributions: Methodology, J.L., D.W., and B.M.; Software, L.Y. and X.H.; Supervision, B.M.; Writing—original draft, J.L.; Writing—review & editing, J.L., C.H., and B.M.

Funding: This work was supported in part by the Fundamental Research Funds for the Central Universities, South Central University for Nationalities Nos. CZZ18003 and QSZ17007, and in part by the natural science foundation of Hubei Province under the grants No. 2018ADC150.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhang, H.G.; Han, W.B.; Lai, X.J.; Lin, D.D.; Ma, J.F.; Li, J.H. Survey on cyberspace security. *Sci. Sin. Inf.* **2016**, *46*, 125–164. [\[CrossRef\]](#)
2. Meng, B.; Lu, J.T.; Wang, D.J.; He, X.D. A survey of security analysis of security protocol implementation. *J. Shandong Univ. (Nat. Sci.)* **2018**, *53*, 1–18. [\[CrossRef\]](#)
3. Zhang, H.G.; Wu, F.S.; Wang, H.Z.; Wang, Z.Y. A Survey: Security Verification Analysis of Cryptographic Protocols Implementations on Real Code. *Chin. J. Comput.* **2018**, *41*, 288–308. [\[CrossRef\]](#)
4. Meng, B.; Huang, C.T.; Yang, Y.T. Automatic Generation of Security Protocol Implementations Written in Java from Abstract Specifications Proved in the Computational Model. *Int. J. Netw. Secur.* **2017**, *19*, 138–153. [\[CrossRef\]](#)
5. Narayan, J.; Shukla, S.K.; Clancy, T.C. A survey of automatic protocol reverse engineering tools. *ACM Comput. Surv.* **2016**, *48*, 40. [\[CrossRef\]](#)
6. Ye, Q.Q.; Bai, G.D.; Wang, K.; Dong, S. Formal Analysis of a Single Sign-On Protocol Implementation for Android. In Proceedings of the 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), Washington, DC, USA, 9–12 December 2015; pp. 9–12.
7. Zuo, C.S.; Wang, W.B.; Wang, R.; Lin, Z.Q. Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 21–24 February 2016.
8. Bai, G.D.; Lei, J.; Meng, G.Z.; Venkatraman, S.S.; Saiena, P.; Sun, J.; Li, Y.; Dong, S. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS), San Diego, USA, 24–27 February 2013.

9. Kobeissi, N.; Bhargavan, K.; Blanchet, B. Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach. In Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France, 26–28 April 2017.
10. Jürjens, J. Automated Security Verification for Crypto Protocol Implementations: Verifying the Jessie Project. *Electron. Notes Theor. Comput. Sci.* **2005**, *250*, 123–136. [CrossRef]
11. Dupressoir, F.; Gordon, A.D.; Jürjens, J. Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols. In Proceedings of the 24th IEEE Computer Security Foundations Symposium, Cernay-la-Ville, France, 27–29 June 2011; pp. 3–17.
12. Bhargavan, K.; Fournet, C.; Gordon, A.D. Modular verification of security protocol code by typing. In Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Madrid, Spain, 17–23 January 2010; pp. 445–456.
13. Backes, M.; Maffei, M.; Unruh, D. Computationally sound verification of source code. In Proceedings of the 17th ACM Conference on Computer and communications security (CCS), Chicago, IL, USA, 4–8 October 2010; pp. 387–398.
14. Swamy, A.; Chen, C.; Fournet, C. Secure distributed programming with value-dependent types. *ACM Sigplan Not.* **2011**, *46*, 266–278. [CrossRef]
15. Swamy, N.; Hrițcu, C.; Keller, C.; Strub, P.-Y.; Rastogi, A.; Delignat-Lavaud, A.; Bhargavan, K.; Fournet, C. Semantic Purity and Effects Reunited in F*. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, Vancouver, BC, Canada, 31 August–2 September 2015.
16. Swamy, N.; Kohlweiss, M.; Zinzindohoue, J.K.; Zanella-Béguelin, S. Dependent Types and multi-monadic effects in F*. In Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), St. Petersburg, FL, USA, 20–22 January 2016; pp. 256–270.
17. Lin, Z.Q.; Liang, X.X.; Xu, D.Y.; Zhang, X.Y. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In Proceedings of the 15th Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 24–27 February 2008.
18. Cui, W.D.; Peinado, M.; Chen, K.; Wang, H.; Briz, L.I. Tupni: Automatic reverse engineering of input formats. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, USA, 27–31 October 2008; pp. 391–402.
19. Caballero, J.; Poosankam, P.; Kreibich, C.; Song, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS), Chicago, IL, USA, 9–13 November 2009; pp. 621–634.
20. Shi, X.L.; Zhu, F.Y.; Liu, L.; Lin, W. Method of Encrypted Protocol Reverse Engineering. *Appl. Res. Comput.* **2015**, *32*, 214–217. [CrossRef]
21. Zhu, Y.N.; Han, J.H.; Yuan, L.; Gu, W.; Fan, Y.D. Protocol Ciphertext Field Identification by Entropy Estimating. *J. Electron. Inf. Technol.* **2016**, *38*, 1865–1871. [CrossRef]
22. Protocol Information Project. Available online: <http://www.4tphi.net/~awalters/PI/PI.html>. (accessed on 13 April 2018).
23. Cui, W.D.; Kannan, J.K.; Wang, H. Discover: Automatic protocol reverse engineering from network trace. In Proceedings of the 16th USENIX Security Symposium on USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 6–10 August 2007.
24. Wang, Y.P.; Meng, J. Biprominer: Automatic mining of binary protocol features. In Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Gwangju, Korea, 20–22 October 2011; pp. 179–184.
25. Wang, Y.P.; Yun, X.C.; Shafiq, M.; Wang, L.Y. A semantics aware approach to automated reverse engineering unknown protocols. In Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP), Austin, TX, USA, 30 October–2 November 2012.
26. Bermudeza, I.; Tongaonkar, A.; Iliofotou, M.; Mellia, M.; Munafò, M.M. Towards automatic protocol field inference. *Comput. Commun.* **2016**, *84*, 40–51. [CrossRef]
27. Luo, J.Z.; Yu, S.Z. Position-based automatic reverse engineering of network protocols. *J. Netw. Comput. Appl.* **2013**, *36*, 1070–1077. [CrossRef]
28. Yun, X.C.; Wang, Y.P.; Zhang, Y.Z.; Zhou, Y. A semantics-aware approach to the automated network protocol identification. *IEEE/ACM Trans. Network.* **2016**, *24*, 583–595. [CrossRef]

29. Tao, S.Y.; Yu, H.G.; Li, Q. Bit-oriented format extraction approach for automatic binary protocol reverse engineering. *IET Commun.* **2016**, *10*, 709–716. [[CrossRef](#)]
30. Bhargavan, K.; Fournet, C.; Gordon, A.D.; Tse, S. Verified interoperable implementations of security protocols. *ACM Trans. Program. Lang. Syst.* **2008**, *31*, 5. [[CrossRef](#)]
31. Bhargavan, K.; Fournet, C.; Corin, R.; Zalinescu, E. Cryptographically verified implementations for TLS. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, USA, 27–31 October 2008; pp. 459–468.
32. Aizatulin, M.; Gordon, A.D.; Jürjens, J. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS), Chicago, IL, USA, 17–21 October 2011; pp. 331–340.
33. Li, Z.M.; Meng, B.; Wang, D.J.; Chen, W. Mechanized Verification of Cryptographic Security of Cryptographic Security Protocol Implementation in JAVA through Model Extraction in the Computational Model. *J. Softw. Eng.* **2015**, *9*, 1–32. [[CrossRef](#)]
34. Avinash, S.; Alessandro, A.; Roberto, C.; Compaagna, L. Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 21–24 February 2016.
35. Giancarlo, P.; Constantin, T.; Eric, B.; Rossow, C. jÅk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection Research in Attacks, Instructions and Defense (LNCS), Kyoto, Japan, 2–4 November 2015; pp. 295–316.
36. Deedleman, S.B.; Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* **1970**, *48*, 443–453. [[CrossRef](#)]
37. Burp Suite Scanner. Available online: <https://portswigger.net/burp> (accessed on 15 January 2018).
38. QQ Mail Login. Available online: <https://mail.qq.com/> (accessed on 12 January 2018).
39. Cui, W.; Paxson, V.; Weaver, N.; Katz, R.H. Protocol-independent adaptive replay of application dialog. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 21–24 February 2016.
40. Ahn, L.; Von Blum, M.; Hopper, N.J.; Langford, J. Captcha: Using hard ai problems for security. In Proceedings of the International Conference on the Advances in Cryptology—EUROCRYPT, Warsaw, Poland, 4–8 May 2003; pp. 294–311.

© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).