University of Castilla-La Mancha

A publication of the

Department of Computer Science

Formal Verification of the Secure Sockets Layer Protocol

by

Llanos Tobarra, Diego Cazorla,
J. José Pardo and M. Carmen Ruiz

Technical Report **#DIAB-04-11-01** November 2004

DEPARTAMENTO DE INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete - 02071 - Spain
Phone +34.967.599200, Fax +34.967.599224

Formal Verification of the Secure Sockets Layer Protocol *

Llanos Tobarra, Diego Cazorla[†], J. José Pardo and M. Carmen Ruiz
({mtobarra, dcazorla, jpardo, carmenr}@info-ab.uclm.es)
Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha. 02071 Albacete, Spain.

Abstract. Secure Sockets Layer (SSL) has become of the most popular security protocols in the Internet. In this paper we present a formal verification of this protocol using the Casper/FDR2 toolbox. In the analysis of SSL v3.0 we have used a methodology that considers incremental versions of the protocol. We have started with the most basic protocol, and then we have included other features such as server and client authentication, digital signatures, etc. We have also verified SSL v2.0 because of the so called *version rollback attack*. Each version has been modelled and verified, and the results have been interpreted. Using this methodology it is easy to understand why some messages are needed in order to ensure confidential communication between a client and a server.

Keywords: Security protocols, model checking, Casper/FDR2

1. Introduction

Security has become one of the most important topics in the Internet in recent years. The quick development of e-commerce, that needs the exchange of sensitive information (credit card numbers, account numbers, ...), has given great importance to the study of communication protocols that allow the exchange of secret data between two or more agents in a non trusted environment.

Some specific e-commerce protocols have been designed and developed in recent years such as SET, IOTP, and CyberCash. Besides, we may also find, in literature, general purpose security protocols such as SSL [11, 10], TLS [6], and SASL [18]. Usually, these security protocols work by adding a layer to the protocol stack just over the TCP layer, keeping transparent to the end user.

The complexity of some e-commerce protocols (e.g. SET) that need very complicated infrastructures to work, has made it difficult to implement these protocols in the real world. This has led to a situation

* This work has been partially supported by the MCyT project "Description and Performance of Distributed Systems and Application to Multimedia Systems" (Ref. TIC2003-07848-c02-02) and the JCCM project "Design and Implementation of Efficient Multimedia Systems by using Formal Techniques" (Ref. PAC-03001)

† Corresponding author: Diego Cazorla (dcazorla@info-ab.uclm.es). Escuela Politécnica Superior de Albacete. Universidad de Castilla-La Mancha. 02071 Albacete, Spain.

in which general purpose security protocols have been used in order to give the customers confidence in their interactions with on-line shops, electronic banks, etc. In the last years, the most popular security protocol used in e-commerce has been Secure Sockets Layer (SSL), developed by Netscape Communications.

Two versions of SSL have been used in recent years: SSL v2.0 [11], released in 1995, and SSL v3.0 [10] released in 1996 in order to solve some problems in v2.0. In 1999, the Internet Engineering Task Force (IETF), the "protocol engineering and development arm of the Internet", presented the Transport Layer Security protocol (TLS) [6], an Internet standard based on SSL v3.0 that, in fact, can be called the next version of SSL.

In parallel to the development of communication and security protocols, some analysis techniques have also been developed in order to verify that these protocols work as they are supposed to, and they do not have bugs that allow intruders to hear or alter the information. One of the most promising techniques in this line is *model checking*. Model checking [3] is a formal methods based technique for verifying finite state concurrent systems, such as sequential circuit designs and communication protocols, that has been implemented in several tools. One of the main advantages of this technique is that model checking is automatic and allows us to know whether a system works properly or not. In case the system does not work as expected, the model checking tool gives us a trace that leads to the source of error.

Formal verification has become a key point in the design of concurrent and distributed system. Formal verification allows us to ensure the correctness of a design at the earliest stage possible, and has two main advantages over two classical techniques such as simulation and testing: *i)* we do not need to build a prototype of the system, and *ii)* we are able to verify the system against every single execution trace. The latter is very important because using simulation or testing we can only find errors, but we cannot ensure that the whole system behaves as expected (some errors may remain hidden until the system is in production stage).

Some general purpose model checking tools have been developed by different research groups: Spin, UPPAAL, Murϕ, Isabelle. These tools allow us to verify not only the functional properties of a system (e.g. Spin), but also the performance of a real-time system (e.g. UPPAAL). Although we can use these general purpose tools in order to verify security protocols, we consider that it is preferable (and more intuitive) to use a tool devoted to the verification of security protocols. The advantages of using this kind of tool are:

i) The specifications of the protocol may be translated with minor changes to the syntax the tool accepts. Using a tool that accepts a completely different syntax is error prone. As is said in model checking, a verification of a system is as good as the model is, i.e., if the model of the system has errors, the verification results will not be valid. Consequently, a correct model of the system is more likely to be obtained when both syntax are very closed.

ii) When an error is found in a model, it must be interpreted in terms of the protocol. As in the previous case, it is more intuitive and accurate to obtain the meaning of the error in terms of messages between peers, encryption keys and secret data exchange than in terms of a trace that finishes in some line of code of the model specification. Using general purpose model checking tools may allow us to know that the system has errors but it may be difficult to understand the meaning of the error and how it can be solved.

In literature we have found references to two model checking tools oriented to security protocol verification: Casper [13] and CAPSL [15]. Casper is a compiler that accepts a syntax very similar to the syntax used to specify protocols, and translates a model into CSP [12] code which is verified using the model checker FDR2 [9]. Although Casper uses a general purpose model checker, the translation to CSP code is made automatically and transparently to the user, and the results of the verification are presented in terms of the Casper model.

In the same line, CAPSL is not a model checking tool but a formal language for expressing authentication and key-exchange protocols using a very intuitive syntax. The CAPSL model must be translated (automatically) to CIL (CAPSL Intermediate Language) and then a "connector" is needed to export the CIL file to the model checking tool syntax. Several connectors have been designed in order to link CAPSL with model checking tools such as Maude [4] and Athena [14].

In this paper we present a formal verification of the SSL protocol using the Casper/FDR2 toolbox. We have chosen Casper/FDR2 instead of CAPSL because (as far as we know) a real implementation of a CIL connector does not exist. Furthermore, Casper/FDR2 has been used successfully to verify several security protocols, and we consider it is also appropriate for the verification of SSL.

In the analysis of SSL v3.0, especially in the handshake subprotocol, we have used a methodology that considers incremental versions of the protocol. We have started with the most basic protocol, and then we have included other features such as server authentication, client authentication, use of digital signatures, verification messages, and use of random data in order to distinguish between different sessions. We have also considered compatibility between SSL v3.0 and SSL v2.0 that allows the so called *version rollback attack*, so we have also needed to verify SSL v2.0. Each version has been modelled and verified, and the results have been interpreted. Using this methodology it is easy to understand why some messages are needed in order to ensure confidential communication between a client and a server.

In literature we may find several papers that deal with the analysis of the SSL protocol. The most interesting one is [17] where SSL Handshake is analysed using a general purpose finite-state enumeration tool called Murϕ. The tool Murϕ [8, 7] was designed for hardware verification and related analysis, but has also been used to analyse some security protocols [16]. Although the paper presents very interesting results, the main problem, from our point of view, is that Murϕ needs a long code in order to specify the protocol and the properties we want to analyse. A simple security protocol like the 3 messages version of the Needham-Schroeder protocol may be specified in about 20 lines using Casper; using Murϕ more than 400 are needed [16]. A translation of these figures to the case of the SSL protocol, which is very complex, leads us to a size of model so big that is difficult to ensure that the model is correct. Our approach, using Casper, minimizes this problem.

In [19] an informal analysis of SSL Handshake and SSL Record is presented. This analysis explores several possible attacks on the SSL protocol and also gives several ways in which the robustness of the SSL protocols can be improved. Finally in [5] a formal specification and analysis of TLS using UPPAAL is presented. Nevertheless, only functional properties are considered and no security properties are taken into account.

The paper is organised as follows. In Section 2, a brief introduction to SSL v3.0 is presented. In Section 3 the SSL Handshake v3.0 subprotocol is verified using the same methodology that was used in [17], i.e., we start from a simple version of the subprotocol and then we include in subsequent versions other features such as server and client authentication, digital signatures, verification messages, etc. In this section, the SSL Handshake v2.0 is also verified in order to consider the so called version rollback attack. In Section 4 the other main subprotocol of SSL, SSL Record, is verified. Finally in Section 5 we give our conclusions and some lines of future work.

2. A brief introduction to SSL

SSL protocol is a layer, between TCP/IP and application layers (see Fig. 1), that allows us to encrypt (in a transparent way) the application data exchanged between a client and a server in a network.

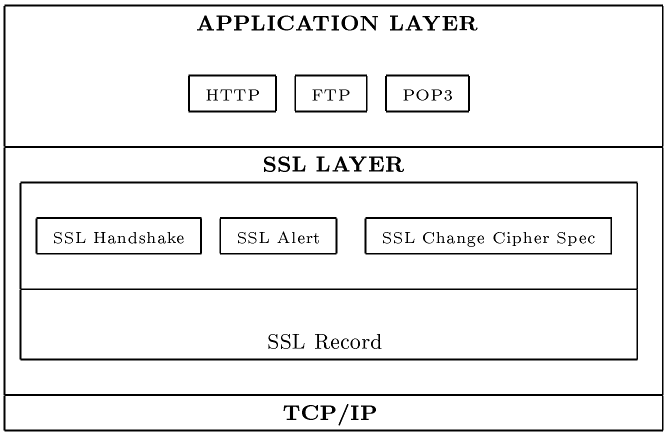

Figure 1. TCP/IP, SSL and Application layers.

SSL is a security protocol based in states with sessions and connections. Each session can be a set of SSL connections and an agent can establish several SSL sessions. Each session and each connection has a set of (different) parameters associated to it.

SSL v3.0 protocol consists of four subprotocols: *SSL Record*, *SSL Handshake*, *SSL Alert* and *SSL ChangeCipherSpec*. Each subprotocol has a clear function: SSL Handshake allows entities to establish sessions and connections, SSL Record exchanges application data between entities, SSL Alert reports on error during SSL execution, and SSL ChangeCipherSpec sets the active configuration, that uses the just negotiated parameters. SSL Alert and SSL ChangeCipherSpec consist of only one message that is usually exchanged during the handshake phase.

When two agents want to establish an SSL session, they first execute the SSL Handshake subprotocol. One agent behaves as a client and the other as a server. A simplified run of the SSL Handshake has the following steps (see Fig. 2):

- The client initiates the SSL Handshake.
- The server sends its certificate in order to be authenticated. The certificate of an agent includes the agent public key and some information about the agent.

– Once the client knows the server public key, it asks the server to prove that it is the owner of the certificate.

– The server encrypts some random data with its private key and sends them to the client.

– After the server authentication, the client generates some secret data and sends them to the server.

At this moment, SSL Handshake has finished and then both agents start SSL Record. Each SSL Record message includes an authentication message code (MAC) which is a hash function computed from application data. MAC is used to check the message integrity. Furthermore, every message is encrypted with a secret session key.

C→S:	Hello	SSL Handshake
S→C:	Hello, this is the Server, Server Certificate	
C→S:	Probe it	
S→C:	Hi, this is the Server, $\{hash[\text{this is the Server}]\}_{ServerPrivateKey}$	
C→S:	OK Server, this is a secret,$\{secret\}_{ServerPublicKey}$	
S→C:	$\{data, MAC[data]\}_{SecretSessionKey}$	SSL Record

Figure 2. Simplified run of SSL between a Client (C) and a Server (S))

More detailed information about SSL v3.0 can be obtained from the Netscape specification [10].

3. Verification of SSL Handshake v3.0

As we have seen before, SSL includes two main subprotocols: *SSL Handshake* and *SSL Record*. SSL Handshake layer allows a client and a server to establish an SSL session and an SSL connection. SSL Record performs a secure exchange application data between peers.

Our analysis centres mainly on SSL Handshake subprotocol. If the peers succeed in establishing a session, the exchange of application data will be secure. The main objectives of SSL Handshake are:

– Synchronise both entities.

– Exchange secret data to generate secure symmetric session keys.

– Negotiate session parameters such as cipher algorithm, compression method, hash functions, size of session keys, session and connection identifiers.

In order to analyse the protocol we have to describe the environment that we consider (see Fig. 3). Each agent can take one of three possible roles: client, server or intruder. There are several honest clients that try to establish a session with a server but they can be disturbed by an intruder.

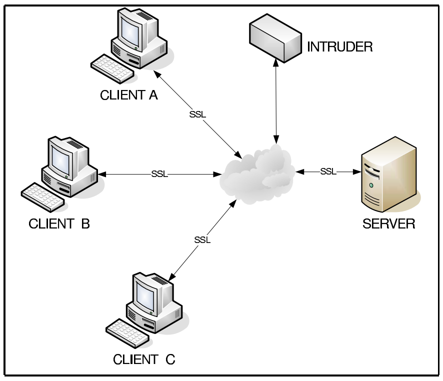

Figure 3. Verification environment

When a client and a server establish a successful SSL connection, SSL guarantees them the following security properties:

A. The connection is private. Each message is encrypted with secure session keys.

B. Each peer is authenticated with both public and private keys.

C. The connection is reliable. Each message includes a message authentication code (MAC) which is computed from application data and some shared secret. This code checks the messages integrity.

Our analysis will find out whether SSL fulfils these security properties or not. In order to check these properties, we need to introduce a new agent in the system model: the intruder. The intruder can perform the following actions:

– Overhear and intercept all the messages over the network.

– Modify the messages, add bytes, delete bytes or change the value of several bytes.

– Generate new messages using its initial knowledge or parts of the overheard messages.

– Send a new or captured message to another entity of the system.

In addition, we assume that:

– The intruder cannot perform any cryptanalysis.

– The data types are atomic, i.e., an intruder cannot guess part of the value of a variable.

– There are trusted Certificate Authorities. They generate certificates that the intruder cannot modify.

– All the entities use secure functions and algorithms.

In order to develop the verification of the SSL Handshake v3.0 subprotocol we will consider an incremental model of the system, starting with a very simple version of the protocol, and introducing new functions into every step. Once we have studied the complete protocol, we will also consider that SSL v3.0 is compatible with SSL v2.0, and this allows a new kind of attack: the version rollback attack. Thus, it is also necessary to verify SSL Handshake v2.0 in order to deal with this attack.

3.1. Basic Protocol

We begin the incremental analysis with a basic protocol (see Fig. 4). Through this initial protocol a client and a server agree on which *CipherSuite* and compression method they use, and some secret data. This basic protocol shows how SSL Handshake works although it does not include any security techniques.

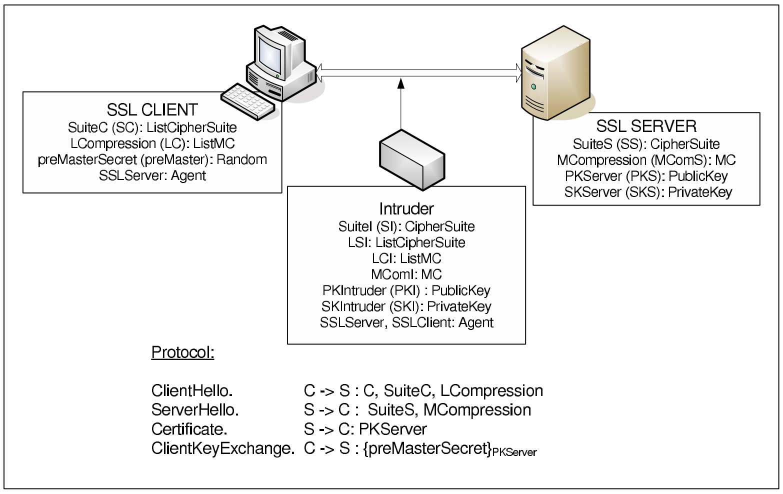

Figure 4. Description of the initial system and the basic protocol

First, a client sends its identifier to the server (C). The client knows the IP address of the server but the server ignores that of the client. Besides, it includes a list of *CipherSuites* (SuiteC) and a list of compression methods (LCompression). A *CipherSuite* is a constant that allows both agents to negotiate a cipher algorithm, a hash function and the size of the pre-master key, among other session parameters.

The server selects a *CipherSuite* (SuiteS) and a compression method (MCompression) from the list and sends them back to the client. It also sends another message to the client that contains the server public key (PKServer). Thus, the client can encrypt some secret data (using the server public key), and send them to the server. Only the server, who knows its own private key, can decrypt them.

We verify that the basic protocol satisfies two security properties:

A. *preMasterSecret* is a secret shared between the client and the server. The intruder cannot find out its value.

B. The client finishes a run of the protocol with the server and both know the same value for several session parameters.

We have found several attacks on the basic protocol, and we can conclude that it does not satisfy any of the two properties. In Fig. 5 we can see how the intruder takes the place of the server and sends its public key to the client. The client generates some secret data, it encrypts them with the intruder public key and sends it back to the intruder. In this case, the intruder knows the value of *MasterSecret* and the client does not run the protocol with the server.

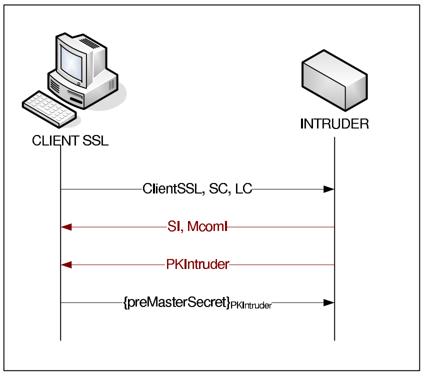

Figure 5. Attack on the basic protocol

The complete Casper specification of the basic protocol and the result of the verification are shown in Appendix A.

3.2. Server Authentication

If we want to prevent the attack on the basic protocol, we must be sure that the client ciphers the secret data with the server public key and the message is sent to the server. In other words, the server has to authenticate itself using a certificate.

We represent certificates as a list with a certificate owner identifier and its public key (see Fig. 6). They are encrypted with a Certificate Authority symmetric key. This key is known by all the entities so we must guarantee that the certificate has not been modified. In addition, we define a verification function called *ValidCertificate* which will allow a client to finish a run (with error) if the certificate has been modified.

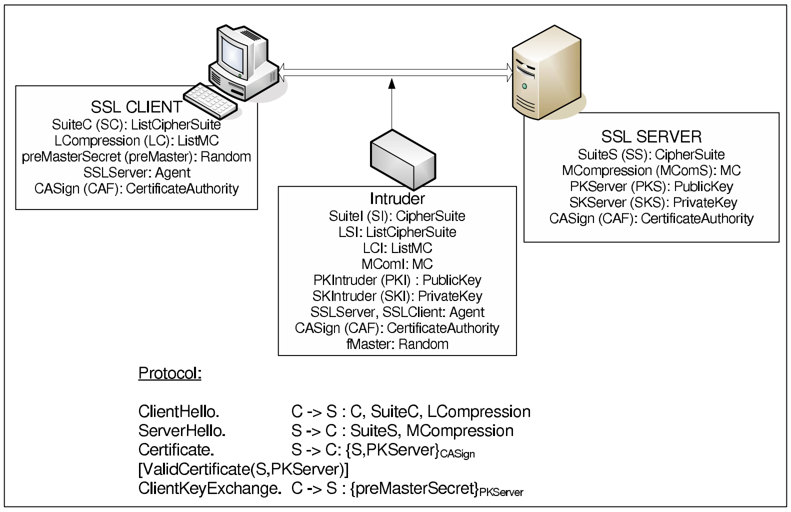

Figure 6. Description of the system and the protocol with server authentication

We verify the same security properties we considered in the basic protocol. This protocol satisfies the first one: client and server shared a secret, *preMasterSecret*, and the intruder ignores the secret values. But, in this case, the intruder can supplant the client and can establish a session as an honest client.

This attack is described in Fig. 7. The intruder deceives the server and sends the client identifier and a list of weak *CipherSuites* to the server. When it knows the server public key, it generates some false secret data and encrypts them with it.

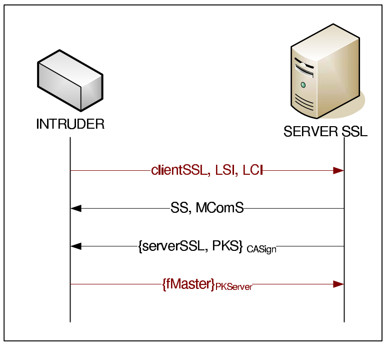

Figure 7. Attack on the protocol with server authentication

3.3. Server and Client Authentication

The attack on the protocol with server authentication may be avoided if the client is also authenticated. In this case, the server will send a request certificate message to the client (see Fig. 8). This message includes a list of trusted CA (Certificate Authority) and a list of acceptable types of certificates. Each entity sends its certificate in a *Certificate* message, and when the other entity receives it, it verifies the certificate integrity with the *ValidCertificate()* function. The selected *CipherSuite* at *ServerHello* message determines whether an agent has to be authenticated or not.

We check the properties A and B in this new protocol version. There is a new attack (see Fig. 9): the intruder cannot supplant an honest agent due to the certificates so it eavesdrops all the messages between the server and the client. This way the intruder knows the server public key and the client certificate and it acts like an intermediate buffer. When the client sends its certificate, the intruder captures all client messages and does not allow the server to receive them. Instead of client messages, the server receives false intruder messages and the intruder takes the place of an honest client.

Besides, the intruder alters *ClientHello* and *ServerHello* messages (see Fig. 9). It changes the list of *CipherSuites* and the list of compression methods. The server has to select among weak configurations, and the intruder alters the selected *CipherSuite* and compression method. This situation may have two main consequences. On one hand, this attack can desynchronise the agents because each one uses a different configuration. On the other hand, the intruder forces them to use a weak cipher mode.

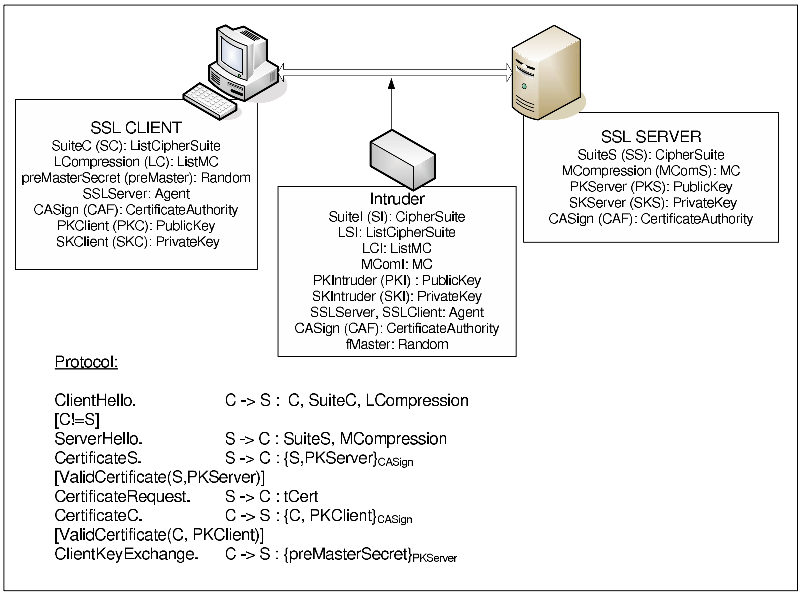

Figure 8. Description of the system and the protocol with server and client authentication

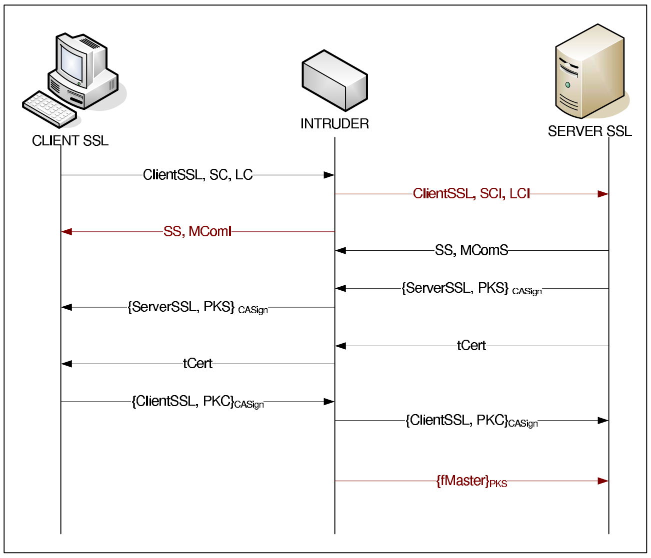

Figure 9. Attack on the protocol with server and client authentication

3.4. Client Digital Signature

We have seen that something more than a certificate is necessary for authenticating the client. A client must be able to prove that it is the certificate owner. Thus, it sends a digital signature in a message called *CertificateVerify* (see Fig. 10). In order to generate its signature, the client applies a hash function to several session parameters and then it encrypts the result with the client private key. Only the real owner of one certificate knows the private key corresponding to the public key and only the owner can cipher some data with it.

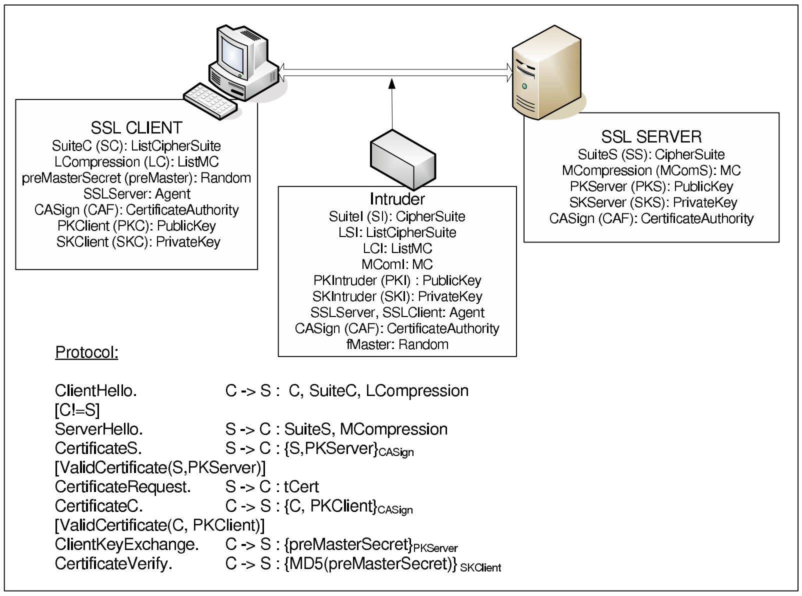

Figure 10. Description of the verification system and the protocol with client complete authentication

At this point, the intruder can supplant neither the client nor the server. But it can perform active attacks on the messages. In particular, it may try to alter non-encrypted parameters in messages, as the list of *CipherSuite* and the list of compression methods (see Fig. 11). These attacks can desynchronise both agents or can reduce the security, as we have seen in previous cases.

3.5. Verification Messages

SSL v3.0 includes messages for avoiding attacks on plain text (text that is not cipher). These verification messages, called *Finished* (see Fig. 12), are computed from all the headings of the previous messages exchanged between the server and the client.

When an entity receives one of these messages, it must check all the included variables, but it does not need to send an acknowledge message. The *Finished* message means that the sender has finished its part of SSL Handshake and it is ready to start SSL Record.

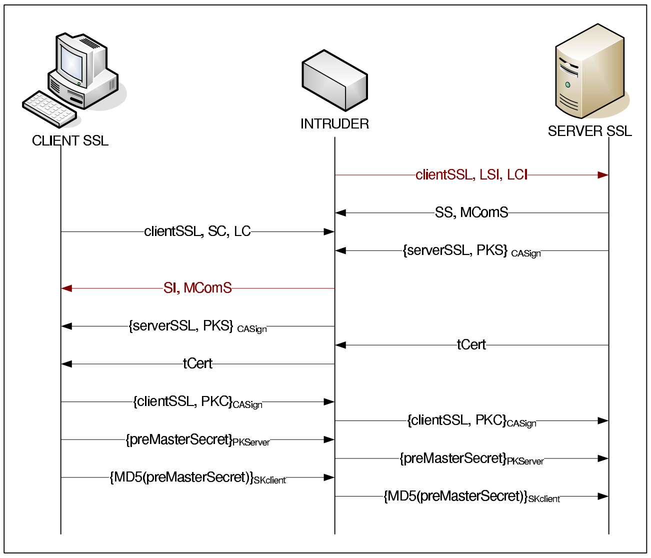

Figure 11. Attack on the protocol with client complete authentication

We have simplified the *Finished* message because the complete message leads to a state explosion in the verification tool. We only include in this message the session variables that have not been checked in other messages.

The verification messages are the first ones to be encrypted with the session key. Each agent has two symmetric session keys: a *write session key* and a *read session key*. In fact, they only generate two keys, but they need to know both keys. One is used to encrypt the sent messages and the other one is used to decrypt the received messages. We represent the process of generating these keys with two symbolic functions. We suppose that the secret data are exchanged successfully, so these session keys are secret and secure.

We add a third property to verify in addition to the two previous ones:

C. The server finishes a run of the protocol with the client and both know the same value for several session parameters.

This new property is analogous to the B property. It checks if the intruder is able to supplant the client and can establish an SSL session as an honest client. We could not add this property before because Casper considered that we did not authenticate the client correctly.

In this protocol version we avoid the attack on plaint text but then we find a "replay" attack. The intruder records all the messages between an honest client and an honest server when they establish a session.

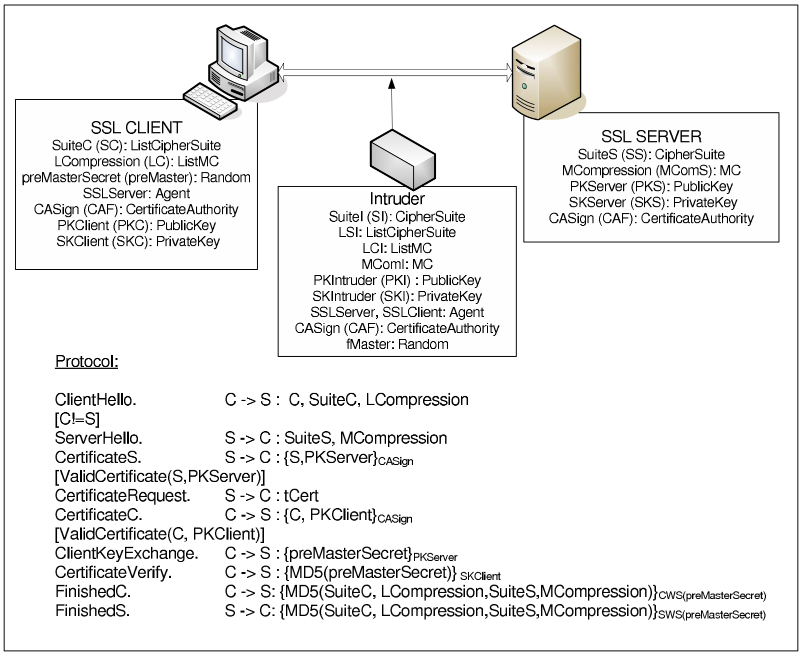

Figure 12. Description of the verification system and the protocol with verification messages

Then the intruder takes the place of the client and behaves like a parrot. It repeats the same sequence of recorded messages to answer the server messages. Maybe the intruder cannot guest the application data but it establishes a new session and the client ignores it.

3.6. Nonces

We must distinguish among runs of the protocol in order to avoid "replay" attacks. So, when an agent starts a new run of the protocol, it generates some random data, called *nonces*, which are sent in its *Hello* message. Each pair of nonces is associated to a connection and it is used to generate the session keys. It gives more security because each pair of session keys is unique so, if an intruder was able to guess a pair, the rest would continue being secure keys. Nonces are verified by *Finished* messages (Fig. 13).

This is a secure version of SSL Handshake which satisfies the three security properties. Nevertheless it is not a complete SSL Handshake specification.

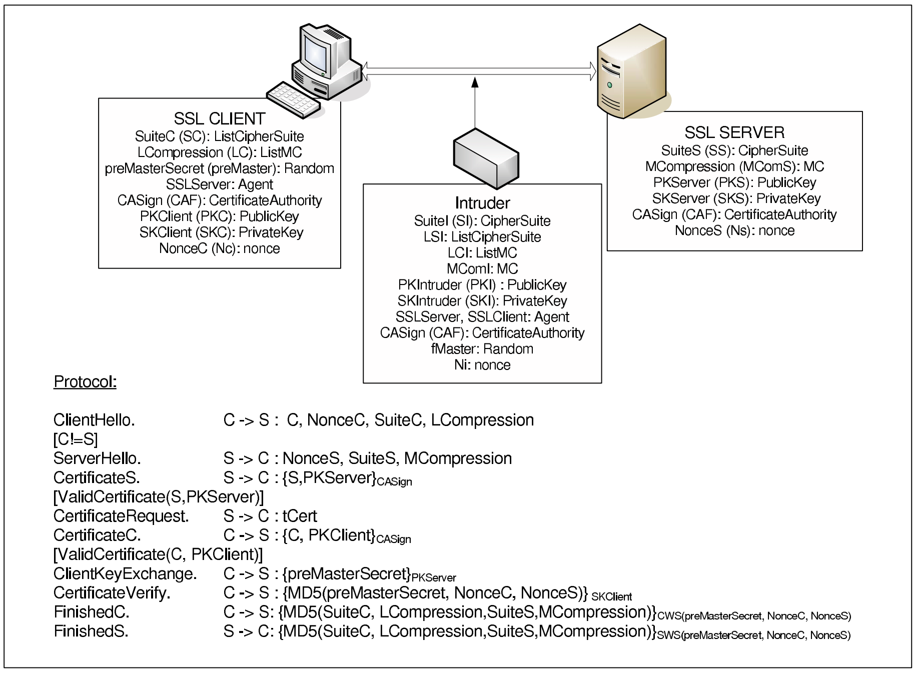

Figure 13. Description of the verification system and the protocol with nonces

3.7. Establishing a session

In the complete SSL specification we need to add two new fields (see Fig. 14):

- SSL agent version. This field indicates to the receiver which SSL version the sender is running. It is included in the *Hello* message.

- Session identifier. This field identifies a set of connections and a session. The server generates this value and it sends it to the client in a *ServerHello* message.

We also add three new messages (see Fig. 14): a *ServerHelloDone* message and two *ChangeCipherSpec* messages. These messages mark the state of the run. They only include a non-encrypted constant. The *ServerHelloDone* is used by the server to indicate that it has finished the *Hello* part of SSL Handshake.

ChangeCipherSpec is, in fact, one of the four subprotocols of SSL v3.0, but may be studied as a message in the Handshake. It consists of a single message. An agent sends this message when it wants to notify the other agent that it will begin to use the just negotiated configuration. The *ChangeCipherSpec* messages are not verified by any message.

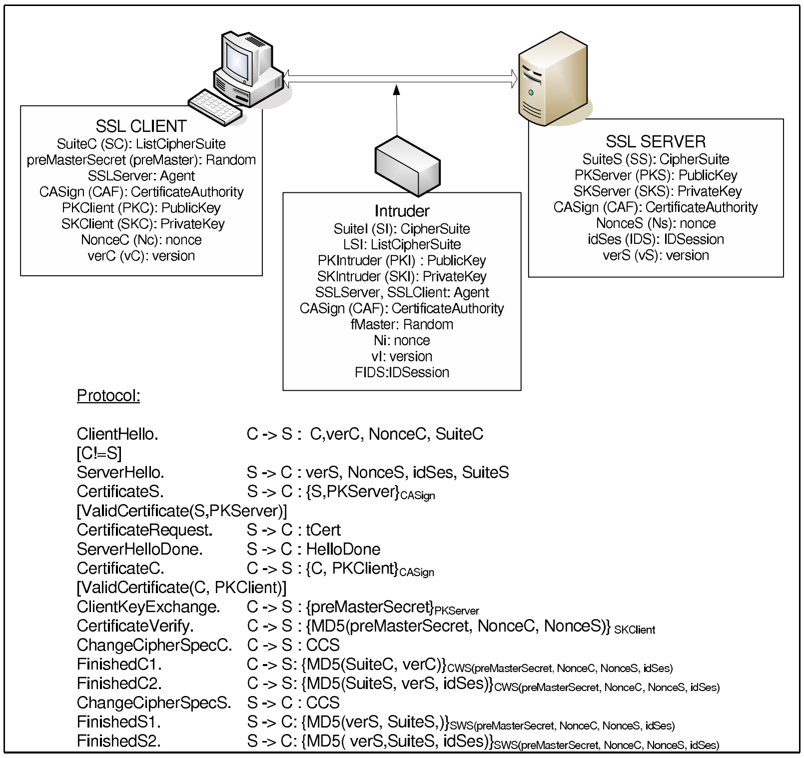

Figure 14. Description of the verification system and the complete SSL Handshake protocol

We check the same three security properties and we verify that this is a secure protocol.

3.8. Restarting a session and establishing a new connection

If a client has executed SSL Handshake and has established a SSL session with a server, when this client wants to resume a SSL connection with the same server, it executes a brief version of SSL Handshake (see Fig. 15). It is not necessary to negotiate a new configuration or exchange secret data in order to generate new session keys. They only exchange *Hello* and *Finished* messages. No agent is authenticated in this short SSL Handshake.

First, the client sends the server the session identifier, a new nonce, the previous list of *CipherSuite* and the list of compression methods. If the server finds the session identifier in its cache, he will send back its *ServerHello* message. Then, both agents send a *ChangeCipherSpec* message and a *Finished* message. This way they restart a session and they establish a new connection.

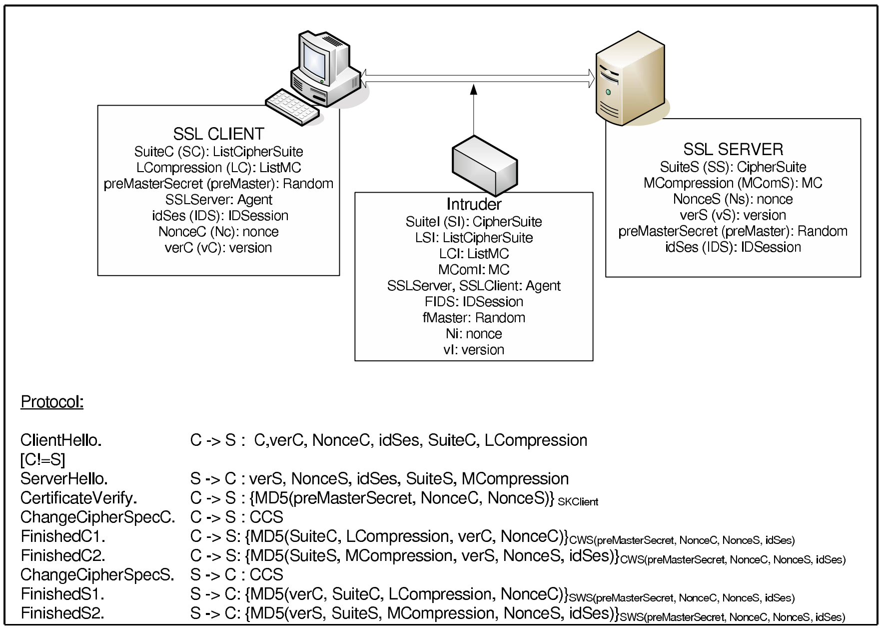

Figure 15. Description of the verification system and the resume protocol

In this case, we can only verify the properties B and C because the server and the client do not exchange secret data. It is a secure protocol because they do not negotiate any session parameter or exchange any secret. They only resume the previous session.

3.9. The version rollback attack: verifying SSL Handshake v2.0

The compatibility between SSL v3.0 and SSL v2.0 allows a client and a server to negotiate which version of the protocol they are going to use. Usually, the client may permit or not using SSL v2.0 (this is a configuration parameter in most internet browsers) but the server must be prepared to accept both kinds of connections. It is very likely that the client (the end user) is not aware of the differences between v2.0 and v3.0, so, usually, both options are marked. At this point, an intruder may force both peers (the client and the server) to use the weakest version of the protocol which, as we will show soon, has several security problems.

Therefore, it is necessary to verify not only SSL Handshake v3.0 but also v2.0. In order to verify the SSL Handshake v2.0 subprotocol [11], we will distinguish four cases to analyse:

1. A client wants to establish a new session with a server and both entities must be authenticated.

2. A client wants to establish a new session with a server, and only the server must be authenticated.

3. A client and a server have established a session before and now the client wants to resume it. Both entities must be authenticated.

4. A client and a server have established a session before and now the client wants to resume it. Only the server must be authenticated.

In cases 1 and 2, a client wants to start an SSL session with a server (see figures 16 and 17). In SSL Handshake v2.0 we have two parts: negotiation parameters phase and authentication phase.

In the negotiation parameters phase, both entities exchange *Hello* messages. The client sends a message called *ClientHello* to the server. In this message, the client includes its identifier because the server ignores it. Besides, the client includes a list of configuration modes and some random data. Each configuration mode is a constant, called *CipherSuite*, which represents a cipher algorithm, a hash function and the size of the secret data. When the server receives this message it selects the most secure *CipherSuite* of the list and generates a new connection identifier in order to identify the connection. Then, it sends a *ServerHello* message which includes the selected *CipherSuite*, the generated identifier connection *idCon* and the server certificate.

We represent certificates as a list of the certificate owner identifier and its public key. A certificate is encrypted with a symmetric key of a trusted Certificate Authority. In order to verify the integrity of the certificate we use a function called *TrustedCertificate* that returns false when a certificate has been modified. When Casper verifies a protocol, if the evaluation of this function is false, it will stop the run and will not take the trace into account.

In SSL v2.0, each entity of a connection uses a pair of symmetric session keys. One of the keys, the *read key*, is used to encrypt all the messages that are sent. The other one, the *write key*, is used to decrypt every message that is received. The client write key is the same as the server read key and the server write key is the same as the client read key.

The client generates some secret data and sends them to the server in the *ClientMasterKey* message. This secret data is used to generate all the session keys. After this message, the server sends a message called *ServerVerify*. The server is authenticated by this message. Only the server knows the corresponding public key so it is the only one that knows the secret data and can generate the correct session key to encrypt the challenge data.

ClientHello. C − > S :
$C, SuiteC, challenge$

ServerHello. S − > C :
$SuiteS, idCon, \{P, PKServer\}_{CASign}$
[TrustedCertificate(S,PKServer)]

ClientMasterKey. C − > S
$\{MasterSecret\}_{PKServer}$

ClientFinished. C − > S :
$\{idCon\%con\}_{WSClient(MasterSecret,challenge,idCon)}$
[idCon == con]

ServerVerify. S − > C :
$\{challenge\%temp\}_{WSServer(MasterSecret,challenge,idCon)}$
[challenge == temp]

RequestCertificate. S − > C :
$\{ChCertificate\}_{WSServer(MasterSecret,challenge,idCon)}$

ClientCertificate. C − > S :
$\{\{C, PKClient\}_{CASign}, \{f(f(MasterSecret, challenge, idCon), ChCertificate,$
$\{S, PKServer\}_{CASign})\}_{SKClient}\}_{WSClient(MasterSecret,challenge,idcon)}$
[TrustedCertificate(C,PKClient)]

ServerFinished. S − > C :
$\{idSes\}_{WSServer(MasterSecret,challenge,idCon)}$

Figure 16. Casper specification of SSL Handshake v2.0. Case 1

If the *CipherSuite* indicates that the client must be authenticated, the server will send a request for client certificate. This message includes a *nonce* (random data) which is used to authenticate the client and a list of valid certificate types. The client will answer with its certificate and a digital signature in a message called *ClientCertificate*.

When both entities have been authenticated, they send their *Finished* messages, which include the session and connection identifiers. At this moment the sender is ready to start SSL Record. Every message in SSL Record is encrypted with the corresponding session key. When an entity receives one of these messages, it has to check the identifier of the message but it does not need to send an acknowledge message.

In case 2 the client does not have to be authenticated and the *RequestCertificate* and *ClientCertificate* are not sent. This case is very common because clients usually connect to a server from different computers without any certificate. For example, clients can be students at the university and the server is an HTTP server with the student's records. Most of the students connect from several computers at different places (at home, at the library,...) and they do not have a certificate in each computer. An intruder, which eavesdrops previous executions, can take the place of an honest client.

ClientHello. C − > S :
$C, SuiteC, challenge$

ServerHello. S − > C :
$SuiteS, idCon, \{P, PKServer\}_{CASign}$
[TrustedCertificate(S,PKServer)]

ClientMasterKey. C − > S
$\{MasterSecret\}_{PKServer}$

ClientFinished. C − > S :
$\{idCon\%con\}_{WSClient(MasterSecret,challenge,idCon)}$
[idCon == con]

ServerVerify. S − > C :
$\{challenge\%temp\}_{WSServer(MasterSecret,challenge,idCon)}$
[challenge == temp]

ServerFinished. S − > C :
$\{idSes\}_{WSServer(MasterSecret,challenge,idCon)}$

Figure 17. Casper specification of SSL Handshake v2.0. Case 2

Cases 1 and 2 may suffer an attack on their plain (non encrypted) text. Each variable which is sent in plain text must be verified by a *Finished* or *ServerVerify* message. But the list of *CipherSuites* and the selected *CipherSuite* are not verified by any message. So, an intruder can intercept the *Hello* message and modify these values. This leads to two main consequences: on one hand, both entities will use a weak configuration mode; on the other hand, the server uses one configuration mode and the client uses a different one. This way, they cannot exchange data.

Cases 3 and 4 consider a different situation. A client has successfully executed SSL Handshake with a server. They have a previous session identifier and the client wants to establish a new connection. They do not need to exchange secret data or negotiate session parameters in either case (see figures 18 and 19). The client sends the server the previous session identifier *idSes*. Then the server looks for it in its session cache. If it finds it, it will send a *ServerHello* with a new connection identifier. At this point the client sends its *Finished* message and the phase of authentication starts.

ClientHello. C − > S :
$C, SuiteC, challenge, idSes\%ses$
[(C!=S)and(idSes == ses)]

ServerHello. S − > C :
$SuiteS, idCon, idEncontrado(C)\%ses$
[idSes == ses]

ClientFinished. C − > S :
$\{idCon\%con\}_{WSClient(MasterSecret,challenge,idCon)}$
[idCon == con]

ServerVerify. S − > C :
$\{challenge\%temp\}_{WSServer(MasterSecret,challenge,idCon)}$
[challenge == temp]

ServerFinished. S − > C :
$\{idSes\}_{WSServer(MasterSecret,challenge,idCon)}$
[idSes == ses]

Figure 18. Casper specification of SSL Handshake v2.0. Case 3

The server sends the *ServerVerify* message in order to be authenticated. If the client must be authenticated, the server will send the *RequestCertificate* message and the client will send a digital signature and its certificate in the same way that we saw in case 2.

There are no attacks on these cases because they do not exchange secret data or negotiate any session parameter. They only confirm the new connection.

As a conclusion, we have seen in the previous cases that there is an important attack on this version of the protocol, and this was the main reason for developing a new version.

ClientHello. C − > S :
$C, SuiteC, challenge, idSes\%ses$
[(C!=S)and(idSes == ses)]

ServerHello. S − > C :
$SuiteS, idCon, idEncontrado(C)\%ses$
[idSes == ses]

ClientFinished. C − > S :
$\{idCon\%con\}_{WSClient(MasterSecret,challenge,idCon)}$
[idCon == con]

ServerVerify. S − > C :
$\{challenge\%temp\}_{WSServer(MasterSecret,challenge,idCon)}$
[challenge == temp]

RequestCertificate. S − > C :
$\{ChCertificate\}_{WSServer(MasterSecret,challenge,idCon)}$

ClientCertificate. C − > S :
$\{\{C, PKClient\}_{CASign}, \{f(f(MasterSecret, challenge, idCon), ChCertificate,$
$\{S, PKServer\}_{CASign})\}_{SKClient}\}_{WSClient(MasterSecret,challenge,idcon)}$
[TrustedCertificate(C,PKClient)]

ServerFinished. S − > C :
$\{idSes\}_{WSServer(MasterSecret,challenge,idCon)}$
[idSes == ses]

Figure 19. Casper specification of SSL Handshake v2.0. Case 4

4. Verification of SSL Record

SSL Record is a subprotocol of SSL that performs the exchange of application data between the server and the client. It guarantees:

– The application data is confidential.

– All sent application data in a message is authenticated.

– Protection against active attacks (as “replay” attacks).

In the SSL Record analysis we suppose that the sever and the client have executed SSL Handshake with success and the session keys have been generated in a secure way.

First, we explain how SSL Record works (see Fig. 20). All the plain data of higher layers is fragmented and it is formatted in structures called record. Then this plain text record is compressed with the selected compression method in SSL Handshake. If *CipherSuite* determines that a block cipher must be used then some random data, called *padding*, will be added at the end of the compressed record. This way, the compressed record size will be multiple of the block size. If a stream cipher must be used, this padding field will be omitted.

After this, the authentication message code (MAC) is computed by applying a hash function to the heading data, the application data, the sender write session key and a sequence number. Finally, the compressed record with its MAC and the padding field, if necessary, is encrypted with the sender write session key. Now the sender can send the message to the other agent.

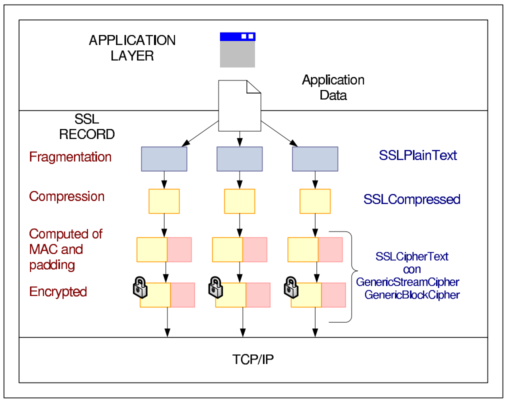

Figure 20. Application data management in SSL Record

When an agent receives a record it computes the MAC from the received data. Then it compares bit to bit the computed MAC and the received MAC. If they agree, the agent will accept the message; otherwise the record will be rejected. This way SSL guarantees the authentication of the message because only an honest agent can generate the write session key and can include it in the MAC. The MAC also allows an agent to verify the integrity of a message.

Each participant has two sequence numbers. One is the number of the message that it has sent since the first SSL Handshake message. The other is the message number that the agent expects to receive. These counters allow the agents to detect if a message is repeated or is lost, and to avoid several active attacks such as “replay”. When a message is lost or repeated, SSL informs the application layer. In addition to the MAC field which guarantees the integrity of the whole message, including the heading, SSL Record is secure against active attacks.

We analyse SSL Record with two simple SSLCipherText records (see Fig. 21). These messages exchange compressed application data, represented with variables *msg1* and *msg2*. Each message has a header which is composed of the agent version, the message type and the message size. In order to avoid state explosion we represent these three fields with *header* type variable. The SSLCipherText bodies are GenericBlockCipher type and they include a field of padding which is represented by the variables *pad1* and *pad2*. Finally, each message includes a message authentication code field which is computed by the hash function *mac*. When an agent receives this field, it must compute this code again and check if both codes have the same value. If they are different the message must be discarded.

After verification, we can conclude that the agents exchange the messages and the intruder cannot guess the value of these messages. The complete Casper specification of the SSL Record protocol and the result of the verification are shown in Appendix B.

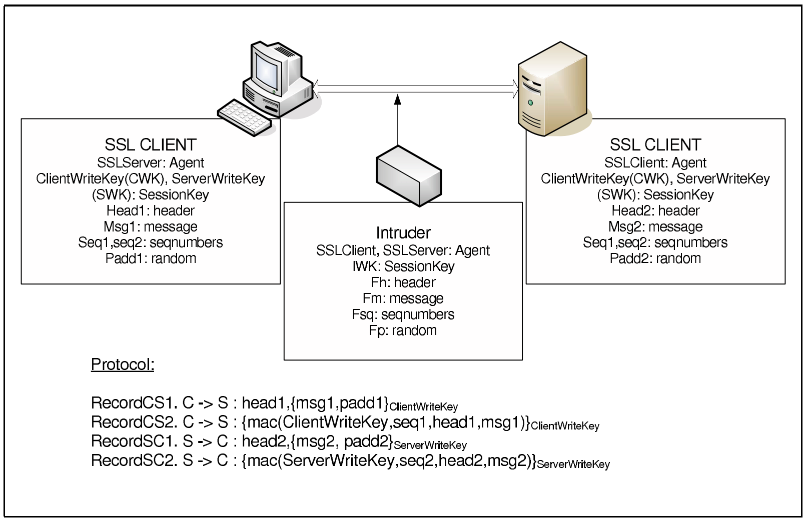

Figure 21. Representation of SSL Record protocol

5. Conclusions and Future Work

Our analysis has shown that SSL v3.0 is a secure protocol if all the features of the protocol are used. It allows a client and a server to exchange some secret data in a secure way. Thus, agents can generate secure session keys that guarantee that all data application encrypted with them is secret, and the connection is private.

SSL avoids most active attacks:

- It provides a solution against attacks on plain text: verification messages.
- It includes random data (nonces) and sequence counters which prevent "replay" attacks or repeated messages.
- Each agent can be authenticated so an intruder can not take the place of an honest agent.
- MAC allows agents to check the integrity of messages.

But there are some attacks that we cannot consider in Casper/FDR2. For instance, there are several messages, such as *ServerHelloDone* and *ChangeCipherSpec*, that are sent as clear text. A cryptanalyst could easily detect these messages and, from that moment on, it could identify each encrypted message in an SSL execution. We assume that an intruder can not perform cryptanalysis so we do not include these kinds of attacks in our analysis. This problem is described in detail in [19].

Nevertheless, SSL v3.0 is compatible with SSL v2.0, and, as we have seen above, this means that a version rollback attack is possible, i.e., the intruder may force both peers to use a weak security protocol, and take advantage of the security "holes" we have detected in SSL v2.0.

From a practical point of view, we can conclude some ideas that may be useful for end users. Even if the end user does not allow the applications to use SSL v2.0, we have seen that the basic version of the SSL Handshake v3.0 is not secure, and an intruder may take the place of a server and run the protocol with the client. We have also seen that more sophisticated versions of the protocol are not secure. In order to assure confidentiality between a client and a server, the client should:

- Verify that the server is really the server with which the client wants to communicate.
- Authenticate itself to the server using certificates and digital signatures.

At this point, it is worth noting that we have focused on the verification of the SSL specification as published in [11, 10]. In literature we may find several documented attacks over SSL v3.0, but these attacks are performed over real implementations of the protocol, not over the SSL specification [1, 2].

Our future work is concerned with extending our analysis of the SSL protocol to other security protocols and e-commerce protocols. In particular we are considering the study of the TLS protocol and its extensions to wireless networks and the study of TLS over PPP (Point to Point Protocol). With respect to e-commerce protocols, we are planning to deal with the verification of the SET protocol.

References

1. Brumley, D. and D. Boneh: 2003, 'Remote Timing Attacks Are Practical'. In: *Proc. of 12th USENIX Security Symposium*. pp. 1–14.
2. Canvel, B., A. Hiltgen, S. Vaudenay, and M. Vuagnoux: 2003, 'Password Interception in a SSL/TLS Channel'. In: *Proc. of Advances in Cryptology (CRYPT'03), LNCS 2729*. pp. 583–599.
3. Clarke, E. M., O. Grumberg, and D. A. Peled: 1999, *Model Checking*. The MIT Press.
4. Denker, G.: 2000, 'Design of a CIL Connector to Maude'. In: *Proc. of Formal Methods and Computer Security, CAV'00 Workshop*.
5. Diaz, G., F. Cuartero, V. Valero, and F. L. Pelayo: 2004, 'Automatic Verification of the TLS Handshake Protocol'. In: *Proc. of the 19th ACM Symposium on Applied Computing (SAC'04)*.
6. Dierks, T. and C. Allen: 1999, 'The TLS Protocol Version 1.0'. Internet Standards, RFC 2246. `http://www.ietf.org/rfc/rfc2222.txt`.
7. Dill, D. L.: 1996, 'The Murϕ Verification System'. In: *Proc. of 8th International Conference on Computer Aided Verification (CAV'96), LNCS 1102*. pp. 390–393.
8. Dill, D. L., A. J. Drexler, A. J. Hu, and C. H. Yang: 1992, 'Protocol Verification as a Hardware Design Aid'. In: *Proc. of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'92)*. pp. 522–525.
9. Formal Systems (Europe) Limited, 'FDR Manual'. `http://www.fsel.com/fdr2_manual.html`.
10. Freier, O. A., P. Karlton, and P. C. Kocher: 1996, 'The SSL Protocol Version 3.0'. Netscape Communications. `http://wp.netscape.com/eng/ssl3/ssl-toc.html`.
11. Hickman, K. E. B.: 1995, 'SSL 2.0 Protocol Specification'. Netscape Communications. `http://wp.netscape.com/eng/security/SSL_2.htm`.
12. Hoare, C.: 1985, *Communicating Sequential Processes*. Prentice Hall.
13. Lowe, G.: 1998, 'Casper: A Compiler for the Analysis of Security Protocols'. *Journal of Computer Security* **6**, 53–84.

14. Millen, J. K.: 2000, 'A CAPSL Connector to Athena'. In: *Proc. of Formal Methods and Computer Security, CAV'00 workshop.*
15. Millen, J. K. and G. Denker: 2002, 'CAPSL and MuCAPSL'. *Journal of Telecommunications and Information Technology* **4**, 16–27.
16. Mitchell, J. C., M. Mitchell, and U. Stern: 1997, 'Automated analysis of cryptographic protocols using Murϕ'. In: *Proc. of IEEE Symposium on Security and Privacy.* pp. 141–151.
17. Mitchell, J. C., V. Shmatikov, and U. Stern: 1998, 'Finite-State Analysis of SSL 3.0'. In: *Proc. of 7th USENIX Security Symposium.* pp. 201–216.
18. Myers, J.: 1997, 'Simple Authentication and Security Layer (SASL)'. Internet Standards, RFC 2222.
`http://www.ietf.org/rfc/rfc2246.txt`.
19. Wagner, D. and B. Schneier: 1996, 'Analysis of the SSL 3.0 Protocol'. In: *Proc. of 2nd USENIX Workshop on Electronic Commerce.* pp. 29–40.

Appendix

A. Verification of the basic protocol

Ccomplete Casper specification of the basic protocol:

```
#Free variables
C, S : Agents
SuiteC: ListCipherSuites
SuiteS: CipherSuites
LCompression: ListMC
MCompression: MC
preMasterSecret: Random
PKServer: PublicKey
SKServer: PrivateKey
InverseKeys= (PKServer, SKServer)

#Processes
Client(C,S,SuiteC, LCompression, preMasterSecret)
Server(S, SuiteS, MCompression, PKServer, SKServer)

#Protocol description
0. -> C: S
ClientHello. C -> S : C, SuiteC, LCompression
ServerHello. S -> C :  SuiteS, MCompression
Certificate. S -> C: PKServer
ClientKeyExchange. C -> S : {preMasterSecret}{PKServer}

#Specification
```

```
Secret(C, preMasterSecret,[S])
Agreement(C,S, [preMasterSecret, SuiteC, MCompression, PKServer])

#Actual variables
SSLClient, SSLServer, SSLIntruder: Agents
SC, LSI: ListCipherSuites
SS, SI: CipherSuites
LC, LCI: ListMC
MComS, MComI: MC
preMaster: Random
PKS,PKI: PublicKey
SKS, SKI: SecretKey
InverseKeys=(PKS, SKS), (PKI, SKI)

#System
Client(SSLCliente,SSLServer,SC,LC,preMaster)
Server(SSLServer,SS,MComS,PKS,SKS)

#Intruder Information
Intruder=SSLIntruder
IntruderKnowledge = {SSLClient,SSLServer,SI,LSI,LCI,MComI,SKI,PKI}
```

And this is the result of the verification. The tool has found two attacks.

```
Casper version 1.4
Parsing...
Type checking...
Consistency checking...
Compiling...
Writing output...
Output written to PA_ssl3.csp
Done

Starting FDR
Checking PA_ssl3.csp

Checking assertion SECRET_SPEC [T= SYSTEM_S
Attack found:

Top level trace:
  SSLClient believes preMaster is a secret shared with SSLServer
  The intruder knows preMaster
```

```
System level:
                0.               ->  SSLClient    : SSLServer
      ClientHello. SSLClient     ->  I_SSLServer  : SSLClient, SC, LC
      ServerHello. I_SSLServer   ->  SSLClient    : SI, MComI
      Certificate. I_SSLServer   ->  SSLClient    : PKI
ClientKeyExchange. SSLClient     ->  I_SSLServer : {preMaster}{PKI}
  The intruder knows preMaster

Checking assertion AuthenticateClientToServerAgreement_
preMasterSecret_SuiteC_MCompression_PKServer [T= SYSTEM_1
Attack found:

Top level trace:
  SSLClient believes (s)he is running the protocol, taking role Client,
  with SSLServer, using data items preMaster, SC, MComI, PKI
  SSLServer believes (s)he has completed a run of the protocol,
  taking role Server, with SSLClient, using data items preMaster,
  SC, MComS, PKS

System level:
                0.              ->  SSLClient  : SSLServer
      ClientHello. SSLClient    -> I_SSLServer : SSLClient, SC, LC
      ClientHello. I_SSLClient  ->  SSLServer  : SSLClient, SC, LC
      ServerHello. SSLServer    -> I_SSLClient : SS, MComS
      ServerHello. I_SSLServer  ->  SSLClient  : SI, MComI
      Certificate. SSLServer    -> I_SSLClient : PKS
      Certificate. I_SSLServer  ->  SSLClient  : PKI
ClientKeyExchange. SSLClient    -> I_SSLServer : {preMaster}{PKI}
ClientKeyExchange. I_SSLClient  ->  SSLServer  : {preMaster}{PKS}

Done
```

B. Verification of the SSL Record

Casper specification of the SSL Record protocol.

```
#Free variables
C, S : Agent
ClientWriteKey, ServerWriteKey: SessionKey
head1, head2: header
msg1, msg2: message
```

```
seq1,seq2: seqnumbers
padd1, padd2: random
mac: HashFunction
InverseKeys= (ClientWriteKey, ClientWriteKey),
  (ServerWriteKey,ServerWriteKey)

#Processes
Client(C,S, ClientWriteKey, ServerWriteKey,
 msg1, head1, seq1, seq2, padd1) knows mac
Server(S,C, ServerWriteKey, ClientWriteKey,
 msg2, head2, seq1, seq2, padd2) knows mac

#Protocol description
0. -> C: S
RecordCS1. C -> S : head1,{msg1,padd1}{ClientWriteKey}
RecordCS2. C -> S : {mac(ClientWriteKey,seq1,head1,msg1)}{ClientWriteKey}
RecordSC1. S -> C : head2,{msg2, padd2}{ServerWriteKey}
RecordSC2. S -> C : {mac(ServerWriteKey,seq2,head2,msg2)}{ServerWriteKey}

#Specification
Secret(C, msg1,[S])
Secret(S, msg2,[C])

#Actual variables
SSLClient, SSLServer, Intruder: Agent
CWK,SWK, IWK: SessionKey
h1,h2,fh: header
m1,m2, fm: message
p1,p2,fp: random
sq1, sq2, fsq: seqnumbers
InverseKeys=(CWK, CWK), (SWK, SWK),(IWK, IWK)

#System
Client(SSLClient, SSLServer, CWK, SWK, m1, h1, sq1, sq2,p1)
Server(SSLServer, SSLClient, SWK, CWK, m2, h2, sq1, sq2,p2)

#Intruder Information
Intruder=Intruder
IntruderKnowledge = {SSLClient, SSLServer, IWK, fh, fm, fsq, fp}
```

Results of the verification. No attacks have been found.

```
Casper version 1.4
Parsing...
```

```
Type checking...
Consistency checking...
Compiling...
Writing output...
Output written to ssl3_record2.csp
Done

Starting FDR
Checking ssl3_record2.csp

Checking assertion SECRET_SPEC [T= SYSTEM_S
No attack found

Checking assertion SECRET_SPEC [T= SYSTEM_S
No attack found

Done
```