FORMAL VERIFICATION OF THE SECURE SOCKETS LAYER PROTOCOL[*]

Llanos Tobarra, Diego Cazorla, J. José Pardo and Fernando Cuartero
Escuela Politécnica Superior de Albacete, Universidad de Castilla-La Mancha, 02071, Albacete, Spain

Keywords: Security protocols, model checking, Casper/FDR2.

Abstract: Secure Sockets Layer (SSL) has become one 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 Handshake 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.

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. 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 (Hickman, 1995), released in 1995, and SSL v3.0 (Freier et al., 1996) 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) (Dierks and Allen, 1999), 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 (Clarke et al., 1999) 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.

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. Among these tools we can find Casper/FDR2 toolbox (Lowe, 1998), ProVerif (Blanchet et al., 2005), LySA(Bodei et al., 2003) and AVISPA (Armando et al., 2005).

In this paper we present a formal verification of the SSL protocol using the Casper/FDR2 toolbox. 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 hand-

[*]This work has been supported by the spanish government with the project "Application of Formal Methods to Web Services", with reference TIN2006-15578-C02-02, and the JCCM regional project "Application of formal methods to the design and analysis of Web Services and e-commerce " (PAC06-0008-6995)

Tobarra L., Cazorla D., José Pardo J. and Cuartero F. (2008).
FORMAL VERIFICATION OF THE SECURE SOCKETS LAYER PROTOCOL.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 246-252
DOI: 10.5220/0001695202460252
Copyright © SciTePress

shake 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 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.

In literature we may find several papers that deal with the analysis of the SSL protocol. The most interesting one is (Mitchell et al., 1998) where SSL Handshake is analysed using a general purpose finite-state enumeration tool called Murϕ. The tool Murϕ (Dill, 1996) was designed for hardware verification and related analysis, but has also been used to analyse some security protocols (Mitchell et al., 1997). 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 (Mitchell et al., 1997). Our approach, using Casper, minimizes this problem.

In (Wagner and Schneier, 1996) 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.

The paper is organised as follows. In Section 2 the SSL Handshake v3.0 subprotocol is verified using the same methodology that was used in (Mitchell et al., 1998), 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. Finally in Section 3 we give our conclusions and some lines of future work.

2 VERIFICATION OF SSL HANDSHAKE v3.0

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

SSL is a security protocol based on 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.

More detailed information about SSL v3.0 can be obtained from the Netscape specification (Freier et al., 1996).

Our analysis is mainly focused on SSL Handshake subprotocol. If the peers succeed in establishing a session, the exchange of application data will be secure.

In order to analyse the protocol we have to describe the environment that we consider 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.

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.

2.1 Basic Protocol

We begin the incremental analysis with a basic protocol (see Fig. 1(a)). 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.

ClientHello.	C $\rightarrow$ S : C,SuiteC,LCompression
ServerHello.	S $\rightarrow$ C : SuiteS,MCompression
Certificate .	S $\rightarrow$ C : PKServer
ClientKeyExchange	C $\rightarrow$ S : $\{\text{preMasterSecret}\}_{PKServer}$

(a) Basic Protocol Narration

ClientHello.	C $\rightarrow$ I_S : C,SuiteC,LCompression
ServerHello.	I_S $\rightarrow$ C : *SI,MComI*
Certificate .	I_S $\rightarrow$ C : *PKIntruder*
ClientKeyExchange	C $\rightarrow$ I_S : $\{\text{preMasterSecret}\}_{PKIntruder}$

(b) Attack on basic protocol version

Figure 1: Narration of the basic protocol and the associated attack.

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. 1(b) 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.

Another attack takes place when 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. In this case, the intruder can supplant the client and can establish a session as an honest client.

2.2 Server and Client Authentication

If we want to prevent the previous described attacks 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. 2(a)). 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.

The client should be also authenticated. In this case, the server will send a request certificate message to the client (see Fig. 2(a)). 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.

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. 2(a)). 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.

At this point, the intruder can supplant neither the

ClientHello.	C $\rightarrow$ S : C,SuiteC,LCompression
	[C!=S]
ServerHello.	S $\rightarrow$ C : SuiteS,MCompression
CertificateS.	S $\rightarrow$ C : $\{$S,PKServer$\}_{CASign}$
	[ValidCertificate(S,PKServer)]
CertificateRequest.	S $\rightarrow$ C : tCert
CertificateC.	C $\rightarrow$ S : $\{$C,PKClient$\}_{CASign}$
	[ValidCertificate(C,PKClient)]
ClientKeyExchange.	C $\rightarrow$ S : $\{$preMasterSecret$\}_{PKServer}$
CertificateVerify.	C $\rightarrow$ S : $\{$MD5(preMasterSecret)$\}_{SKClient}$

(a) Basic Protocol Narration

ClientHello.	$I_C \rightarrow$ S : *C,SI,LComI*
ServerHello.	S $\rightarrow I_C$: SuiteS,MCompression
CertificateS.	S $\rightarrow I_C$: $\{$S,PKServer$\}_{CASign}$
CertificateRequest.	S $\rightarrow I_C$: tCert
ClientHello.	C $\rightarrow I_S$: C,SuiteC,LCompression
ServerHello.	$I_S \rightarrow$ C : *SI,MCompI*
CertificateS.	$I_S \rightarrow$ C : $\{$S,PKServer$\}_{CASign}$
CertificateRequest.	$I_S \rightarrow$ C : tCert
CertificateC.	C $\rightarrow I_S$: $\{$C,PKClient$\}_{CASign}$
ClientKeyExchange.	C $\rightarrow I_S$: $\{$preMasterSecret$\}_{PKServer}$
CertificateVerify.	C $\rightarrow I_S$: $\{$MD5(preMasterSecret)$\}_{SKClient}$
CertificateC.	$I_C \rightarrow$ S : $\{$C,PKClient$\}_{CASign}$
ClientKeyExchange.	$I_C \rightarrow$ S : $\{$preMasterSecret$\}_{PKServer}$
CertificateVerify.	$I_C \rightarrow$ S : $\{$MD5(preMasterSecret)$\}_{SKClient}$

(b) Attack on basic protocol version

Figure 2: Narration of the protocol with client and server authentication and the associated attack.

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. 2(b)). These attacks can desynchronise both agents or can reduce the security, as we have seen in previous cases.

2.3 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. 3), 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.

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

ClientHello.	C $\rightarrow$ S : C,SuiteC,LCompression
	[C!=S]
ServerHello.	S $\rightarrow$ C : SuiteS,MCompression
CertificateS.	S $\rightarrow$ C : $\{$S,PKServer$\}_{CASign}$
	[ValidCertificate(S,PKServer)]
CertificateRequest.	S $\rightarrow$ C : tCert
CertificateC.	C $\rightarrow$ S : $\{$C,PKClient$\}_{CASign}$
	[ValidCertificate(C,PKClient)]
ClientKeyExchange.	C $\rightarrow$ S : $\{$preMasterSecret$\}_{PKServer}$
CertificateVerify.	C $\rightarrow$ S : $\{$MD5(preMasterSecret)$\}_{SKClient}$
FinishedC.	C $\rightarrow$ S : $\{$MD5(SuiteC,LCompression,SuiteS, MCompression)$\}_{CWS(preMasterSecret)}$
FinishedC.	S $\rightarrow$ C : $\{$MD5(SuiteC,LCompression,SuiteS, MCompression)$\}_{SWS(preMasterSecret)}$

Figure 3: Narration of the protocol with verification messages.

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. Maybe the intruder cannot guest the application data but it establishes a new session and the client ignores it.

2.4 Establishing a Session

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. 4). In the complete SSL specification we need to add two new fields:

SSL agent version and session identifier.

We also add three new messages: 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.

ClientHello.	$C \rightarrow S$: C,verC,NonceC,SuiteC, LCompression *[C!=S]*
ServerHello.	$S \rightarrow C$: verS,NonceS,idSes,SuiteS, MCompression
CertificateS.	$S \rightarrow C$: $\{S,PKServer\}_{CASign}$ *[ValidCertificate(S,PKServer)]*
CertificateRequest.	$S \rightarrow C$: tCert
ServerHelloDone.	$S \rightarrow C$: HelloDone
CertificateC.	$C \rightarrow S$: $\{C,PKClient\}_{CASign}$ *[ValidCertificate(C,PKClient)]*
ClientKeyExchange.	$C \rightarrow S$: $\{preMasterSecret\}_{PKServer}$
ChangeCipherSpecC.	$C \rightarrow S$: CCS
CertificateVerify.	$C \rightarrow S$: $\{MD5(preMasterSecret,NonceC,NonceS)\}_{SKClient}$
FinishedC1.	$C \rightarrow S$: $\{MD5(SuiteC,verC)\}_{CWS(preMasterSecret,NonceC,NonceS,idSes)}$
FinishedC2.	$C \rightarrow S$: $\{MD5(SuiteS,verS,idSes)\}_{CWS(preMasterSecret,NonceC,NonceS,idSes)}$
ChangeCipherSpecS.	$S \rightarrow C$: CCS
FinishedS1.	$S \rightarrow C$: $\{MD5(SuiteC,verC)\}_{SWS(preMasterSecret,NonceC,NonceS,idSes)}$
FinishedS2.	$S \rightarrow C$: $\{MD5(SuiteS,verS,idSes)\}_{SWS(preMasterSecret,NonceC,NonceS,idSes)}$

Figure 4: Narration 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.

2.5 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. 5). 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.

ClientHello.	$C \rightarrow S$: C,verC,NonceC,SuiteC, LCompression *[C!=S]*
ServerHello.	$S \rightarrow C$: verS,NonceS,idSes,SuiteS, MCompression
CertificateVerify.	$C \rightarrow S$: $\{MD5(preMasterSecret,NonceC,NonceS)\}_{SKClient}$
ChangeCipherSpecC.	$C \rightarrow S$: CCS
FinishedC1.	$C \rightarrow S$: $\{MD5(SuiteC, verC, LCompression, NonceC)\}_{CWS(preMasterSecret,NonceC,NonceS,idSes)}$
FinishedC2.	$C \rightarrow S$: $\{MD5(SuiteS,verS,NonceS,MCompression,idSes)\}_{CWS(preMasterSecret,NonceC,NonceS,idSes)}$
ChangeCipherSpecS.	$S \rightarrow C$: CCS
FinishedS1.	$S \rightarrow C$: $\{MD5(SuiteC, verC, LCompression, NonceC)\}_{SWS(preMasterSecret,NonceC,NonceS,idSes)}$
FinishedS2.	$S \rightarrow C$: $\{MD5(SuiteS,verS,NonceS,MCompression,idSes)\}_{SWS(preMasterSecret,NonceC,NonceS,idSes)}$

Figure 5: 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.

2.6 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 (Hickman, 1995), 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.

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 : $\{idCon\%con\}_{WSC(MasterSecret,challenge,idCon)}$
	[idCon == con]
ServerVerify.	S → C : $\{challenge\%temp\}_{WSS(MasterSecret,challenge,idCon)}$
	[challenge == temp]
RequestCertificate.	S → C : $\{ChCertificate\}_{WSS(MasterSecret,challenge,idCon)}$
ClientCertificate.	C → S : $\{\{C,PKClient\}_{CASign},\{f(f(MasterSecret, challenge,idCon),ChCertificate, \{S,PKServer\}_{CASign})\}_{SKClient}\}_{WSC(MasterSecret,challenge,idcon)}$
	[TrustedCertificate(C,PKClient)]
ServerFinished.	S → C : $\{idSes\}_{WSS(MasterSecret,challenge,idCon)}$

Figure 6: Casper specification of SSL Handshake v2.0. Case 1.

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 : $\{idCon\%con\}_{WSC(MasterSecret,challenge,idCon)}$
	[idCon == con]
ServerVerify.	S → C : $\{challenge\%temp\}_{WSS(MasterSecret,challenge,idCon)}$
	[challenge == temp]
ServerFinished.	S → C : $\{idSes\}_{WSS(MasterSecret,challenge,idCon)}$

Figure 7: Casper specification of SSL Handshake v2.0. Case 2.

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 6 and 7). 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.

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.

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.

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 : $\{idCon\%con\}_{WSC(MasterSecret,challenge,idCon)}$
	[idCon == con]
ServerVerify.	S → C : $\{challenge\%temp\}_{WSS(MasterSecret,challenge,idCon)}$
	[challenge == temp]
ServerFinished.	S → C : $\{idSes\}_{WSS(MasterSecret,challenge,idCon)}$
	[idSes == ses]

Figure 8: Casper specification of SSL Handshake v2.0. Case 3.

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 : $\{idCon\%con\}_{WSC(MasterSecret,challenge,idCon)}$
	[idCon == con]
ServerVerify.	S → C : $\{challenge\%temp\}_{WSS(MasterSecret,challenge,idCon)}$
	[challenge == temp]
RequestCertificate	S → C : $\{ChCertificate\}_{WSS(MasterSecret,challenge,idCon)}$
ClientCertificate.	C → S : $\{\{C,PKClient\}_{CASign},\{f(f(MasterSecret, challenge,idCon),ChCertificate, \{S,PKServer\}_{CASign})\}_{SKClient}\}_{WSC(MasterSecret,challenge,idcon)}$
	[TrustedCertificate(C,PKClient)]
ServerFinished.	S → C : $\{idSes\}_{WSS(MasterSecret,challenge,idCon)}$
	[idSes == ses]

Figure 9: Casper specification of SSL Handshake v2.0. Case 4.

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 8 and 9).

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.

3 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.

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 (Wagner and Schneier, 1996).

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.

At this point, it is worth noting that we have focused on the verification of the SSL specification as published in (Hickman, 1995; Freier et al., 1996). 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 (Brumley and Boneh, 2003; Canvel et al., 2003).

Our future work is concerned with extending our analysis of the SSL protocol to other security protocols and e-commerce protocols. With respect to e-commerce protocols, we are planning to deal with the verification of the SET protocol.

REFERENCES

Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Hankes Drielsma, P., Heám, P.-C., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., and Vigneron, L. (2005). The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In Etessami, K. and Rajamani, S. K., editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*. Springer.

Blanchet, B., Abadi, M., and Fournet, C. (2005). Automated Verification of Selected Equivalences for Security Protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL. IEEE Computer Society.

Bodei, C., Buchholtz, M., Degano, P., Nielson, F., and Nielson, H. R. (2003). Automatic validation of protocol narration. In *Proceedings of the 16th Computer Security Foundations Workshop (CSFW 03).*, pages 126–140. IEEE Computer Society Press.

Brumley, D. and Boneh, D. (2003). Remote Timing Attacks Are Practical. In *Proc. of 12th USENIX Security Symposium*, pages 1–14. USENIX Press.

Canvel, B., Hiltgen, A., Vaudenay, S., and Vuagnoux, M. (2003). Password Interception in a SSL/TLS Channel. In *Proc. of Advances in Cryptology (CRYPT'03), LNCS 2729*, pages 583–599. Springer.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press.

Dierks, T. and Allen, C. (1999). *The TLS Protocol Version 1.0*. Internet Standards, RFC 2246. `http://www.ietf.org/rfc/rfc2222.txt`.

Dill, D. L. (1996). The Murϕ Verification System. In *Proc. of 8th International Conference on Computer Aided Verification (CAV'96), LNCS 1102*, pages 390–393. Springer.

Freier, O. A., Karlton, P., and Kocher, P. C. (1996). *The SSL Protocol Version 3.0*. Netscape Communications. `http://wp.netscape.com/eng/ssl3/ssl-toc.html`.

Hickman, K. E. B. (1995). *SSL 2.0 Protocol Specification*. Netscape Communications. `http://wp.netscape.com/eng/security/SSL_2.htm`.

Lowe, G. (1998). Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6:53–84.

Mitchell, J. C., Mitchell, M., and Stern, U. (1997). Automated analysis of cryptographic protocols using Murϕ. In *Proc. of IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society Press.

Mitchell, J. C., Shmatikov, V., and Stern, U. (1998). Finite-State Analysis of SSL 3.0. In *Proc. of 7th USENIX Security Symposium*, pages 201–216. USENIX Press.

Wagner, D. and Schneier, B. (1996). Analysis of the SSL 3.0 Protocol. In *Proc. of 2nd USENIX Workshop on Electronic Commerce*, pages 29–40. USENIX Press.