Verification of the TLS Handshake protocol

Carst Tankink (0569954), Pim Vullers (0575766)

20th May 2008

1 Introduction

In this text, we will analyse the Transport Layer Security (*TLS*) handshake protocol. This protocol is originally described in RFC 4346 of the Internet Engineering Task Force (IETF) [2] and is meant to establish a *secure* connection between a client and a server agent.

In order to carry out the analysis, we will first give a high-level description and an informal narration of the basic handshake protocol in section 2. After this, we transcribe this narration into a ProVerif [1] protocol description, which we then validate using the automated validation tool ProVerif. Analysis results will be described in section 3.

Next to the basic protocol, TLS offers several options, among which *client certification* and *session resuming*. We will describe these extensions in sections 4 and 6. Due to timing constraints, we were only able to analyse the certification extension, which is described in section 5. Finally we will describe any difficulties modelling the TLS handshake protocol in ProVerif and conclude this text in sections 7 and 8.

2 The basic protocol

In this section, we consider the basic narration that derives from Paulson [3]. It consists of six messages, and several computation steps. We explicitly note the computation steps made by the parties in between the key exchange, and give explanations of the most important steps.

2.1 Abbreviations and definitions used

A (Alice) the client

B (Bob) the server

Na, Nb fresh nonces from A resp. B

Pa, Pb the preferences for encryption and compression from A resp. B

Sid a session identifier

cert(X, Kx) a certificate containing public key Kx of X

PMS the pre-master-secret

PRF a pseudo-random-number function

M the master-secret $M = PRF(PMS, Na, Nb)$

Finished an hash of M and all preceding handshake messages $Finished = Hash(M, messages)$

2.2 Informal narration

As noted above, this informal narration is based on the one given by Paulson [3]. We have made the between-message computations explicit, and have added tags to all messages. The tagging of messages is considered good practice, and we therefore adhere to this from the start.

$$\begin{array}{rcll}
 & : & A \text{ generates fresh nonce } Na & (1) \\
 & : & A \text{ determines } Sid & (2) \\
 & : & A \text{ chooses options } Pa & (3) \\
A \rightarrow B & : & ClientHello, A, Na, Sid, Pa & (4) \\
 & : & B \text{ generates fresh nonce } Nb & (5) \\
 & : & B \text{ chooses options } Pb & (6) \\
B \rightarrow A & : & ServerHello, Nb, Sid, Pb & (7) \\
B \rightarrow A & : & ServerCertificate, cert(B, Kb) & (8) \\
 & : & A \text{ generates secret } PMS & (9) \\
A \rightarrow B & : & \{ClientKeyExchange, PMS\}_{Kb} & (10) \\
 & : & A \text{ and } B \text{ expect } PMS \text{ to be secret} & (11) \\
 & : & A \text{ calculates } M = PRF(PMS, Na, Nb) & (12) \\
 & : & A \text{ calculates } Finished & (13) \\
A \rightarrow B & : & \{Finished\}_{clientK(Na,Nb,M)} & (14) \\
 & : & B \text{ calculates } M = PRF(PMS, Na, Nb) & (15) \\
 & : & B \text{ calculates } Finished & (16) \\
 & : & B \text{ begins ServerAuth(A, B, PMS)} & (17) \\
B \rightarrow A & : & \{Finished\}_{serverK(Na,Nb,M)} & (18) \\
 & : & A \text{ verifies received } Finished & (19) \\
 & : & B \text{ verifies received } Finished & (20) \\
 & : & A \text{ ends ServerAuth(A, B, PMS)} & (21) \\
 & : & A \text{ expects } M \text{ and } Finished \text{ to be secret} & (22)
\end{array}$$

We will now clarify those steps that are non-standard in informal narration (such as nonce generation).

Determining *Sid* (step 2) In this step, A picks a session identifier. This is necessary for session resuming, where an attempt is made to resume the session identified with *Sid*. For the moment, we assume that *Sid* is randomly generated by A.

Choosing encryption options (steps 3 and 6) These steps are in the TLS specification for establishing the preferences for encryption and compression. These are not detailed here, and we assume that Pa and Pb are just some randomly generated data.

Server certificates We do not model the Public Key Infrastructure that lies behind the certificates, we just assume that the server has a certificate containing his public encryption key and his name. It is important to note that an agent cannot create this certificate himself. In the actual implementation, these certificates are provided by a Certification Authority after carefully checking the identity of the owner.

3 Analysis of the basic protocol

To analyse the TLS protocol, we have modeled it for use with the ProVerif tool [1]. The extended ProVerif model, including client certification, can be found in appendix A. For this model we used the PI-calculus notation since this is closer to the informal narration than the default Horn clauses notation used by ProVerif.

We have made several assumptions and simplifications in formalising the informal narration, which we will explain here.

Certificate distribution Instead of having a CA signing a certificate containing a name and a public key, we assume that an agent has this certificate after the initialisation phase.

Certificate verification Normally, a certificate is validated by checking the signature of the issuing CA and verifying the identity of this CA. This is done recursively until a trusted CA is found. In this formalisation, we assume that one can simply verify a certificate by confronting it with the name that is supposed to be in it. The certificate then yields the public key belonging to this name. Since agents cannot create certificates themselves, certificate forgery is not possible in this model.

In our analysis, we have looked at several attacks. First of all, we have looked at the secrecy of the pre-master secret. Since the key is based on this pre-master secret, it can be assumed that the keys are secret. However, we have also tested that the final *Finished* messages are secret. This is not the case for the message going from the server to the client. The attack trace that shows this, however, is just a normal instantiation of the protocol, in which an attacker sends the name of A in the ClientHello message. The server then assumes that he is talking to A, and proceeds in the run.

Next to these secrecy requirements, we have also looked at the injectivity of authentication between client and server, meaning that when server and client finish the protocol, the client A knows she has established a secure connection based on pre-master secret PMS with server B.

This means that the server does not know that a message encrypted under $\mathit{clientK}(Na, Nb, M)$ is actually sent by A, since an attacker could pretend to be A in the beginning of a further valid run, and control the key used after this establishment. In other words, the authentication of server and client is not mutual; the server is always authenticated to the client, but the server must trust that the client in honest. This result is also noted by Paulson [3].

For our verification we only used the built-in external attacker model of ProVerif, using `query attacker: someSecret` and `query evinj: someEvent(A, B, C) ==> evinj: someOtherEvent(A, B, C)`. Due to timing constraints we did not model a spy to verify secrecy against an internal attacker.

4 Client certification

As noted above, the basic protocol does not provide any guarantees to the server (B) about talking to the client (A).

In order to fix this, TLS provides an optional certificate sent by A. This certificate establishes that A is the owner of a signing key, that is later used to verify that she has generated the nonce Na and the pre-master secret PMS for communication with B, and that those have not been sent by an attacker impersonating A.

4.1 Informal narration

We have extended the informal narration as given in section 2.2 with the required messages and computations for client certification. Those additional steps have been marked such that they can be easily recognised.

$$
\begin{array}{rcll}
 & : & A \text{ generates fresh nonce } Na & (1) \\
 & : & A \text{ determines } Sid & (2) \\
 & : & A \text{ chooses options } Pa & (3) \\
A \rightarrow B & : & ClientHello, A, Na, Sid, Pa & (4) \\
 & : & B \text{ generates fresh nonce } Nb & (5) \\
 & : & B \text{ chooses options } Pb & (6) \\
B \rightarrow A & : & ServerHello, Nb, Sid, Pb & (7) \\
B \rightarrow A & : & ServerCertificate, cert(B, Kb) & (8) \\
\mathbf{A \rightarrow B} & : & \mathbf{ClientCertificate, cert(A, Ka)} & (9) \\
 & : & A \text{ generates secret } PMS & (10) \\
A \rightarrow B & : & \{ClientKeyExchange, PMS\}_{Kb} & (11) \\
 & : & A \textbf{ begins ClientAuth(A, B, PMS)} & (12) \\
\mathbf{A \rightarrow B} & : & \{\mathbf{CertificateVerify, Hash(Nb, B, PMS)}\}_{\mathbf{Ka^{-1}}} & (13) \\
 & : & B \textbf{ ends ClientAuth(A, B, PMS)} & (14) \\
 & : & A \text{ and } B \text{ expect } PMS \text{ to be secret} & (15) \\
 & : & A \text{ calculates } M = PRF(PMS, Na, Nb) & (16) \\
 & : & A \text{ calculates } Finished & (17) \\
A \rightarrow B & : & \{Finished\}_{clientK(Na, Nb, M)} & (18) \\
 & : & B \text{ calculates } M = PRF(PMS, Na, Nb) & (19) \\
 & : & B \text{ calculates } Finished & (20) \\
 & : & B \text{ begins ServerAuth(A, B, PMS)} & (21) \\
B \rightarrow A & : & \{Finished\}_{serverK(Na, Nb, M)} & (22) \\
 & : & A \text{ verifies received } Finished & (23) \\
 & : & B \text{ verifies received } Finished & (24) \\
 & : & A \text{ ends ServerAuth(A, B, PMS)} & (25) \\
 & : & A \textbf{ and } B \text{ expect } M \text{ and } Finished \text{ to be secret} & (26)
\end{array}
$$

We will now clarify those steps that are new in this informal narration.

Sending of ClientCertificate (step 9) informs the server of the signature verification key. Again we do not model the Public Key Infrastructure that lies behind the certificates, as described before for server certificates.

Sending of CertificateVerify (step 13) assures the authenticity of the client to the server by hashing some relevant data, and signing it.

Authenticity markers (steps 12 and 14) can be added similar to the authenticity markers for the server authentication.

Secrecy goals (step 26) now also apply for B since he now can be sure that A is not being impersonated by an attacker.

5 Analysis of client certification

We have added the client certification step to the ProVerif formalisation, leading to the model found in section A. Using this model, the authentication between client and server is provably established, and the finished messages are secret both when going from the server to the client and when going from client to server.

This verification has been done using the same model and assumptions as mentioned for the basic protocol in section 3.

6 Session resuming

The TLS protocol also provides the opportunity for an interrupted session to be resumed. The client and server are supposed to have kept the session ID *Sid* and the master secret M in some cache. At resumption a short version of the handshake protocol will be run, consisting only of *ClientHello*, *ServerHello* and *Finished* messages. The client and server random values (Na and Nb respectively) are generated anew, and with the old M new session keys are derived ($clientK(Na, Nb, M)$ and $serverK(Na, Nb, M)$). This is to prevent attacks on the keys that may have occurred in the meantime (that may be long). On the other hand, computation of the pre-master secret PMS is not repeated, as that is the most time-consuming part of the handshake. So a resumption handshake is pretty efficient.

Due to a lack of time we have not managed to model and verify session resuming in ProVerif. We refer to the work in the interactive proving tool Isabelle by Paulson [3], for a model and verification of this option of the TLS handshake protocol.

7 Modelling TLS in ProVerif

The main problem in modelling the protocol with ProVerif was that there are several data items, such as the *Finished* messages in the protocol which are computed from other items, but still meant to be secret. ProVerif cannot attack these computed objects by default, but it is possible to verify the safety of such an object by outputting a flag on the computed objects. Now, an attacker can only find the value of the flag when she has knowledge of the object.

Other difficulties mainly had to do with unreachable code, but checking for this and repairing it was straightforward.

8 Conclusion

Based on our own verification and the work done by Paulson [3] we can conclude that the TLS handshake protocol establishes a secure session key for client and server. In the basic model the only trust that can be established is from the client in the server, if a higher level of trust is required, mutual authentication should be used which is provided by the client certification option of the protocol.

References

[1] Bruno Blanchet. ProVerif: Cryptographic protocol verifier in the formal model. Webpage. `http://www.proverif.ens.fr/`, accessed on 18 May 2008.

[2] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, April 2006.

[3] Lawrence C. Paulson. Inductive analysis of the Internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.

A ProVerif formalisation of the protocol

```
(***********************************************************************
 *** The Transport Layer Security (TLS) Handshake Protocol            ***
 ***                                                                  ***
 *** Carst Tankink (0569954), Pim Vullers (0575766)                   ***
 ***********************************************************************)

(************************************************************************
 * DEFINITIONS                                                          *
 ************************************************************************)

(* A public channel *)
free net.

(* Message tags *)
free ClientHello, ClientKeyExchange, ClientCertificate,
     ServerHello, ServerCertificate, CertificateVerify.

(* Agent initialization is done over a private channel *)
private free initialClientData, initialServerData.

(* The cryptographic constructors *)
fun hash/1.          (* hashing *)
fun encrypt/2.       (* symmetric key encryption *)
fun pencrypt/2.      (* Public key encryption *)
fun sign/2.          (* Public key signing *)
fun enc/1.           (* Extracts encryption key from a keypair *)
fun dec/1.           (* Extracts decryption key from a keypair *)

(* The cryptographic destructors *)
(* decrypt/2 *)
reduc decrypt(encrypt(x, y), y) = x.

(* pdecrypt/2 *)
reduc pdecrypt(pencrypt(x, enc(y)), dec(y)) = x.

(* unsign/2 *)
reduc unsign(sign(x, enc(y)), dec(y)) = x.

(* A constructor that maps agents to their secret keypairs *)
private fun pubkeypair/1.
private fun signkeypair/1.

(* Pseudo-random-number function *)
fun PRF/1.

(* Symmetric key construction *)
fun clientK/3.
fun serverK/3.

(* Certificates *)
private fun cert/2.

(* If cert(x,y) establishes x as owner of key y, y is returned *)
reduc verify(cert(x, y), x) = y.

(************************************************************************
 * QUERIES                                                              *
 ***********************************************************************)

  (* secrecy Pre Master secret *)
    query attacker: PMS.

  (* secrecy Master secret *)
    query attacker: MSa.
    query attacker: MSb.

  (* secrecy Finished message from the client *)
    query attacker: FinishedAFlag.

  (* secrecy Finished message from the server *)
    query attacker: FinishedBFlag.

  (* authenticity of the server *)
    query evinj: endServerAuth(x, y, z) ==> evinj: beginServerAuth(x, y, z).

  (* authenticity of the client *)
    query evinj: endClientAuth(x, y, z) ==> evinj: beginClientAuth(x, y, z).

  (* Dead code check *)
    query attacker: clientFinished.
    query attacker: serverFinished.

(***********************************************************************
 * CLIENT PROCESS, this client is the initiator of the protocol.        *
 ***********************************************************************)

let client =
  (** Initial agent data from a trusted channel **)
    in(initialClientData, (A, signKey, clientCert));

  (** Replication to model arbitrary sessions **)
    !

  (** Get the server's name, perhaps as user input **)
    in (net, B);

  (* A generates fresh nonce Na *)
    new Na;

  (* A determines Sid, session is randomly generated in basic model *)
    new Sid;

  (* A chooses options Pa, options are just random data for our model *)
    new Pa;

  (* A -> B : ClientHello *)
    let CH = (ClientHello, A, Na, Sid, Pa) in out(net, CH);

  (* B -> A : ServerHello *)
    in(net, SH); let (=ServerHello, Nb, =Sid, Pb) = SH in

  (* B -> A : ServerCertificate *)
    in(net, SC); let (=ServerCertificate, serverCert) = SC in
  (* A -> B: ClientCertificate *)
    let CC = (ClientCertificate, clientCert) in out(net, CC);

  (* A generates secret PMS *)
    new PMS;

  (** begin client authentication **)
    event beginClientAuth(A, B, PMS);

  (* A -> B : ClientKeyExchange *)
    let encKey = verify(serverCert, B) in
    let CKE = pencrypt((ClientKeyExchange, PMS), encKey) in out(net, CKE);

  (* A -> B: CertificateVerify *)
    let CV = sign((CertificateVerify, hash((Nb, B, PMS))), signKey) in
    out(net, CV);

  (* A calculates the Master secret M *)
    let M = PRF((PMS, Na, Nb)) in

  (* A calculates Finished *)
    let Finished = hash((CH, SH, SC, CC, CKE, CV, M)) in

  (* A -> B : Finished *)
    out(net, encrypt(Finished, clientK(Na, Nb, M)));

  (* B -> A : Finished *)
    in(net, FB);

  (* A verifies received finished *)
    let =Finished = decrypt(FB, serverK(Na, Nb, M)) in

  (** end server authentication **)
    event endServerAuth(A, B, PMS);

  (** secrecy check on the Master secret **)
    new MSa; out (M, MSa)|

  (** secrecy check on the Finished message **)
    new FinishedAFlag; out(Finished, FinishedAFlag)|

  (** dead code check **)
    new clientFinished; out(net, clientFinished).

(***********************************************************************
 * THE SERVER PROCESS                                                  *
 ***********************************************************************)

let server =
  (** Initial agent data from a trusted channel **)
    in(initialServerData, (B, decKey, serverCert));

  (** Replication to model arbitrary sessions **)
    !

  (* A -> B : ClientHello *)
    in(net, CH); let (=ClientHello, A, Na, Sid, Pa) = CH in
  (* B generates fresh nonce Nb *)
    new Nb;

  (* B chooses options Pb, just some random data in our model *)
    new Pb;

  (* B -> A : ServerHello *)
    let SH = (ServerHello, Nb, Sid, Pb) in out(net, SH);

  (* B -> A : ServerCertificate *)
    let SC = (ServerCertificate, serverCert) in out(net, SC);

  (* A -> B : CientCertificate *)
    in (net, CC); let (=ClientCertificate, clientCert) = CC in

  (* A -> B : ClientKeyExchange *)
    in(net, CKE); let (=ClientKeyExchange, PMS) = pdecrypt(CKE, decKey) in

  (* A -> B : CertificateVerify *)
    let unsignKey = verify(clientCert, A) in
    in(net, CV); let (=CertificateVerify, cvHash) = unsign(CV, unsignKey) in

  (* B verifies client signature *)
    let =cvHash = hash((Nb, B, PMS)) in

  (** end client authentication **)
    event endClientAuth(A, B, PMS);

  (* A -> B : Finished *)
    in (net, FA);

  (* B calculates M *)
    let M = PRF((PMS, Na, Nb)) in

  (* B calculates Finished *)
    let Finished = hash((CH, SH, SC, CC, CKE, CV, M)) in

  (** server authentication **)
    event beginServerAuth(A, B, PMS);

  (* B -> A : Finished *)
    out(net, encrypt(Finished, serverK(Na, Nb, M)));

  (* B verifies received Finished *)
    let =Finished = decrypt(FA, clientK(Na, Nb, M)) in

  (** secrecy check on the Master secret **)
    new MSb; out (M, MSb)|

  (** secrecy check on the finished **)
    new FinishedBFlag; out(Finished, FinishedBFlag)|

  (** dead code check **)
    new serverFinished; out(net, serverFinished).

(*********************************************************************
 * THE INITIALIZER PROCESS                                           *
 *********************************************************************)
```

```
let initializer =
    new agent; (* Generate agent name *)
    let clientKeyPair = signkeypair(agent) in (* Generate client key *)
    let serverKeyPair = pubkeypair(agent) in (* Generate server key *)
    let clientCert = cert(agent, dec(clientKeyPair)) in
    let serverCert = cert(agent, enc(serverKeyPair)) in
    out (initialClientData, (agent, enc(clientKeyPair), clientCert));
    out (initialServerData, (agent, dec(serverKeyPair), serverCert));
    out (net, agent).

(**********************************************************************
 * THE SYSTEM                                                          *
 **********************************************************************)

process
    !initializer | !client | !server
```