Verifying MobiilID protocol with SPIN

Vilen Looga
vilen@ut.ee

MTAT.07.019 Research Seminar in Cryptography,
Institute of Computer Science,
University of Tartu
November 2009

Abstract. This paper recognizes the need to use automated ways to verify security protocols. MobiilID protocol is used to provide users of mobile devices the ability to digitally sign files, access online banks and even participate in e-voting. We chose SPIN, a LTL model checker, to create a model of the protocol and test it for possible problems.

Key words: software model checking, cryptographic protocol, mobileid

1 Introduction

The MobiilID protocol (henceforth also called as "the protocol") was devised to enable secure communication between a mobile device with a special smart card and a web service that would require the user to authenticate himself or give his digital signature. This technology enables new services, most notably e-voting, using a mobile device. Obviously, such a protocol has to be secure against malicious attacks which target either the implementation or the protocol itself. This paper will focus on the latter.

Most of the security protocols that are frequently used undergo automatic evaluation. For example, the protocol that we will examine was verified with the ProVerif[5] formal protocol verifier.

The reason, why mission critical software undergoes automatic verification, is that even the most skillful security experts cannot create and analyse all the possible situations that might arise when the protocol is exevuted. To explain that, think about the protocol as a (in)finite space of all possible states. Each new state is created by a transition from an existing state by executing a single instruction defined in the protocol. For example, in our case a transition from one state to another would be sending a single request, or assigning a new value to a variable. As one can imagine the number of possible states and transitions — also called a state space — would grow immensely, especially if the protocol is complex. Also, it would be impossible for a human being to traverse the whole state space and analyse the consequences of all possible traces within the state space. That's where software model checkers such as SPIN[1] come to help. This kind of software can create the whole state space of a protocol based on a abstract model of the protocol and check if any of the states cause the security of the

protocol to be compromized. More information about that can be found in the SPIN manual [6].

Now that we are armed with such tools we will have to implement the model of the protocol and also state security properties for that model. A security property states that something should never happen[6]. For example, in case of this protocol, a security property can state that some bit of information should never go to a third party or that some variable should never have a certain value. The model checker will tell us if there are any violations of security properties in our model. As an added bonus, the model checker can also tell if there are any possible deadlocks or livelocks that might occur due to potential problems in the specification of the protocol.

The implemented model of the security protocol is also useful in another way — it could be used to automatically create an implementation of the protocol, thus creating mathematically proven production-grade code. More on that in section 5.

There are real-world examples, where automatic protocol verification software was able to uncover problems in security protocols, that were previously undetected, even though some of the checked protocols have been available for public scrutiny for decades, like in the case of AWN[2, 3].

The main goal of this paper is to show how software model checking can be used either as an addtional measure to verify the security of a protocol or even as a tool to create protocol implementations that are mathematically proven to be secure. For people less familiar with software model checking, this paper will serve as an introduction to this field with an example practical application. For security experts, in case they were not familiar with this topic, our paper presents an alternative way to analyse cryptographic protocols. Finally, for people who are interested in the security of e-voting and using mobile phones for digital signing, this paper presents another proof of how safe (or unsafe) the system is.

The paper is divided as follows: section 2 describes the protocol in detail and section 3 explains which tools were used to implement the model of the protocol and the process of implementation. Section 4 analyses the results of the model checker output. Finally we present some ideas for future work in section 5 and finish the paper with conclusions. Appendixes A and B contain the Promela source code and the output of the Spin model checker.

2 MobiilID Protocol

We will verify one of the protocols that is described in MobiilID documentation — the "Mobile authentication in asynchronous Cilent-Server mode" protocol (Figure 1) as descibed in section 5.3.1 of the DigiDocService document[4].

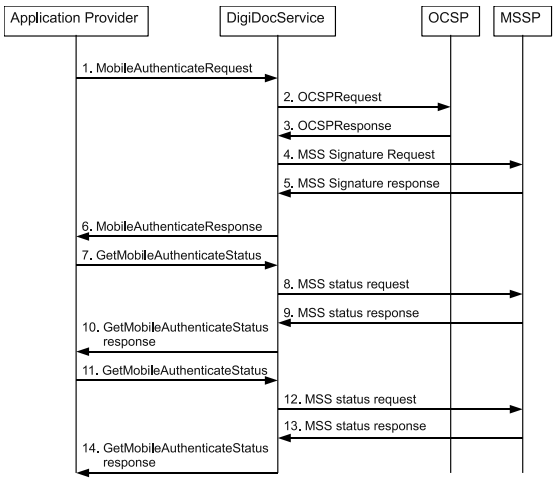

Fig. 1. MobiilID authentication in asynchronous Cilent-Server mode

3 Verification

3.1 Implementation

We will create a model based on the authentication MobiilID protocol, which was described in section 2.

Although the MobiilID protocol is not complex, compared to some other security protocols, we still have to use some abstraction when creating a model. It means that we have to make some assumptions and carefully select which nuances of the possible protocol interactions should be modeled and which should be not. If we would try model all possible attack vectors, it would take too much effort to implement and the resulting state space would grow exponentially. In conclusion, the goal of our model is to find flaws in the way different parties interact with each other through those protocol, and how an adversary could potentially use it for his advance.

When creating our model we relied on following assumptions, which are common among model checkers specifically tailored for security protocols (as stated by Armando[3]). These assumptions allow us to create a simplified initial model of the protocol:

1. **Perfect Cryptography (PC)** — it is only possible to decrypt a cryptotext with the correct key. It means that in our model we do not question the cryptographic primitives used in the protocol.
2. **Dalev Yao Intruder (DY)** — means that the intruder, who is present in our model, could interact with any other party; he is capable of eavesdropping, forging or intercepting messages.

3. **Honest Principals (HP)** — parties participating in an interaction that answer to only specified messages and only in a matter specified by the protocol
4. **Security Goals (SG)** — security goals are invariants, meaning that we can only state that something should not happen. Expressing reachability or liveness properties is generally problematic with LTL model checkers, unless we can express and invariant of a reachability property (which would equal to a security property). More on that in [6].

Those assumptions have been criticized by Armando for various reasons. Notably, he believes that it is not possible to successfully analyze some classes of protocols, because they do not follow these assumptions (or follow them only partially). More on that in [3]. In our work we tried to consider those problems and address them if possible.

The Promela code will model interactiosn between two parties: the DigiDoc service, which is used to authenticate users, and AppProvider, an entity requiring authentication of a user. Both of them follow the protocol as specified in the documentation. Third party in this model will be the intruder who is able to intercept all messages and does not have to follow the protocol.

The Promela code can be found in appendix A.

4 Results

The output of the SPIN model checker is in appendix B. After executing the protocol model, we discovered two potential attacks:

1. **Denial of Service attack** — it means that the intruder could intercept all messages coming from AppProvider or DigiDocService and therefore disrupt the service. However, the attacker would not be able to gain any additional information from the intercepted messages.
2. **False public key** — in this case the attacker would somehow convince one (or both) of the parties that his public key should be used for message encryption. All the messages sent will be readable only by the attacker. The implications of that would be severe.

The first attack is a threat to almost any protocol and MobiilID is no exception. The implications of the second attack are severe, however, it should be noted that there is no (known) inherit flaw in the protocol that would force one of the communication parties to use the public key of the attacker. This situation was modeled on purpose in the protocol to test a situation where either the AppProvider or the DigiDoc service are dishonest.

As a result of the verification we can state that although the requests used in the protocol are quite complex, if we focus only on the parts of the protocol that make it secure — end-to-end encryption and partner nonces — the model of interactions becomes quite simple. Therefore, there is a less chance that some potential attack could be hidden in the complexity of the protocol, which is what

our model has shown. In conclusion, both of the attacks discoved by the model were already known, and most of security protocols are vulnerable to them.

5 Future work

The usual steps of creating mission critical software is firstly, specifications that create the abstract model of how the software would work, and secondly implementation, which is written according to the specifications. The specifications undergo evaluation, for example an abstract model of a protocol would be examined by security experts and model checking software. Similarly, the code of the implementation would be reviewed and put through automated tests (also specified by experts).

The inherit problem with such an approach is that there is a human factor between the transition from specification to the implementation of the software. We believe that this is problematic, since it is impossible for a human being to verify with absolute certainty that the implementation follows the specification. For example, if a specification of a protocol has been verified mathematically to not have flaws, the implementation of the same protcol migh not be as secure due to faulty human translation of specifications into code.

One solution for such a problem would be using automatic generation of program code from the mathematically proven specifications. The model of the protocol would be specified and verified mathematically in a model chercker, such as Spin or Proverif. After that the same model would be used to automatically generate native code, which, after compilation, would give us an implementation with the same mathematically proven security as the specifications.

There are tools that enable automatic implementation generation such as PIG[8] and [7].

6 Conclusions

This paper explained how model checking could be used to model a protocol and check it for possible flaws that compromize security of communicationg parties. The example protocol that we verified using SPIN — MobiilID — turned out to be secure and we were not able to discover new unexpected attacks, besides the two that were already known. We also discussed how automatic implementation generation could decrease the likelyhood of flaws in security software.

References

1. Spin: On-the-fly LTL Model Checker. http://spinroot.com/spin/whatispin.html.
2. Alessandro Armando, Roberto Carbone, and Luca Compagna. LTL Model Checking for Security Protocols. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 385–396, Washington, DC, USA, 2007. IEEE Computer Society.

3. Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps. In *FMSE '08: Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 1–10, New York, NY, USA, 2008. ACM.
4. AS Sertifitseerimiskeskus. *DigiDocService Specification*, 2.122 edition, April 2007. http://www.sk.ee/files/DigiDocService_spec_eng.pdf.
5. Bruno Blanchet. ProVerif: Cryptographic protocol verifier in the formal model. http://www.proverif.ens.fr.
6. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2004.
7. Chung-Shyan Liu. A program generator for object-based implementation of communication protocol software. *Autonomous Decentralized Systems, International Symposium on*, 0:384, 1999.
8. Jose Quaresma. PIG - Protocol Implementation Generator.

A SPIN Source Code for MobiilID authentication protocol

```
/*
* Author: Vilen Looga <vilen AT ut DOT ee>
* This is the Promela model of MobiilId authentication protocol
*/

/* Communication parties */
mtype = {app_provider, digidocservice, intruder, key_app, key_intr, key_digidoc};

/* Some messages */
mtype = { SESSIONID, OK, USER_AUTHENTICATED };

/* Requests used in the protocol. Since a request consists of many variables, it is
necessary to define new types to be able to send the whole request as a single message *
/* MobileAuthenticate request */
typedef MARQ {
mtype IDCode, Country, PhoneNo, Language, ServiceName, MessageToDisplay, SPChallenge,
MessagingMode, AsyncConfiguration;
bool ReturnCertData, ReturnRevocationData;
mtype key; /* Other parties publickey that is used to encrypt the message */
}

/* MobileAuthenticate response */
typedef MARS {
mtype Sesscode, Status, UserIDCode, UserGivenName, UserSurname, UserCountry,
UserCN, CertificateData, ChallengeID, Challenge, RevocationData;
mtype key;
}
```

```
/* GetMobileAuthenticateStatus request */
typedef GMASRQ {
mtype Sesscode, WaitSignature;
mtype key;
}

/* GetMobileAuthenticateStatus response */
typedef GMASRS {
mtype Status, Signature;
mtype key;
}

/* Channels to send messages (partner, messagetype) */
/* Since SPIN does not support universal channels,
each message type has to have its own channel */
chan chan_MARQ = [0] of { mtype, MARQ };
chan chan_MARS = [0] of { mtype, MARS };
chan chan_GMASRQ = [0] of { mtype, GMASRQ };
chan chan_GMASRS = [0] of { mtype, GMASRS };


active proctype AppProvider()
{
/* Partner */
mtype partner;
mtype partner_key;

/* Status bytes */
byte session_status = 0; /* 0 - session not established or pending,
  1 - session request pending, 2 - session established */
byte authreq_status = 0; /* 0 - authentication request not sent,
  1 - authentication request pending */

/* Messages */
MARQ ma_rqst;
MARS ma_resp;
GMASRQ gmas_rqst;
GMASRS gmas_resp;

start:
/* Random selection of partner */
/* Each "::" clause has an equal chance to be selected*/
if
::partner = digidocservice; partner_key = key_digidoc;
```

```
::partner = intruder; partner_key = key_intr;
fi;

/* Select which phase to commit */
/* Each "::" clause where the start condition evaluates to TRUE
   has an equal chance to be selected. Code inside assert() must
   evaluate to TRUE otherwise we have reached an unwanted state */
if
::(session_status == 0) -> goto phase0;
::(session_status == 1) -> goto phase1;
::(session_status == 2) && (authreq_status == 0) -> goto phase2;
::(session_status == 2) && (authreq_status == 1)-> goto phase3;
::assert((session_status == 0) && (authreq_status == 1));
::assert((session_status == 1) && (authreq_status == 1));
::(session_status == 0) -> goto stop;
::(session_status == 2) -> goto stop;
fi;

/* Sending MobileAuthenticate request */
phase0: /* This kind of labels are used as bookmarks in code and can be jumped to from
   any place in the code. Must not be used inside atomic() clauses */
ma_rqst.key = partner_key;
chan_MARQ ! partner, ma_rqst; /* "channel ! msg" is syntax for sending msg into channel*
session_status = 1;
goto start;

/* Receiving MobileAuthenticate response */
phase1:
if
::chan_MARS ? app_provider, ma_resp;
::timeout -> assert(false);
fi;

gmas_rqst.Sesscode = ma_resp.Sesscode;

if
/* d_step means that all the code is executed without creating intermediate states */
::d_step { ma_resp.key == key_app && ma_resp.Status == OK } ->
session_status = 2; goto start; /* Session established */
::else -> assert(false); /* Session not established, protocol fails */
fi;

/* Sending GetMobileAuthenticateStatus request*/
phase2:
gmas_rqst.key = partner_key;
```

```
chan_GMASRQ ! partner, gmas_rqst;
authreq_status = 1 -> goto start; /* Authentication request sent*/

/* Receiving GetMobileAuthenticateStatus response */
phase3:
if
::chan_GMASRS ? app_provider, gmas_resp;
::timeout -> assert(false);
fi;

if
::d_step { (gmas_resp.key == key_app)
&& (gmas_resp.Status == USER_AUTHENTICATED) } -> goto stop;
::else -> assert(false); /* Bad response, protocol fails */
fi;

stop:
skip;
}

active proctype DigiDocService()
{
/* Partner */
mtype partner;
mtype partner_key;

/* Messages */
MARQ ma_rqst;
MARS ma_resp;
GMASRQ gmas_rqst;
GMASRS gmas_resp;

start:
/* Random selection of partner */
if
::partner = app_provider; partner_key = key_app;
::partner = intruder; partner_key = key_intr;
fi;

phase0:
/* Receiving MobileAuthenticate request */
chan_MARQ ? digidocservice, ma_rqst;
if
::ma_rqst.key == key_digidoc -> goto phase1;
::else -> goto start;
```

```
fi;

phase1:
/* Sending MobileAuthenticate response */
ma_resp.key = partner_key;
ma_resp.Sesscode = SESSIONID;
ma_resp.Status = OK;

if
::chan_MARS ! partner, ma_resp -> goto phase2;
::timeout -> goto start;
fi;

phase2:
/* Receiving GetMobileAuthenticateStatus request */
chan_GMASRQ ? digidocservice, gmas_rqst;
if
::d_step { gmas_rqst.key == key_digidoc
      && gmas_rqst.Sesscode == SESSIONID } -> goto phase3;
::else -> goto start;
fi;

phase3:
/* Sending GetMobileAuthenticateStatus response */
gmas_resp.key = partner_key;
gmas_resp.Status = USER_AUTHENTICATED;
if
::chan_GMASRS ! partner, gmas_resp -> goto stop;
::timeout -> goto start;
fi;

stop:
skip;

}

active proctype Intruder()
{
/* Partner */
mtype partner;

/* Messages */
MARQ ma_rqst;
MARS ma_resp;
GMASRQ gmas_rqst;
```

```
GMASRS gmas_resp;

/* Receiving data */
do
/* Intercept message */
::chan_MARQ ? _, ma_rqst ->
if
::(ma_rqst.key == key_intr) -> skip;
::skip; /* Dump message*/
fi;
::chan_MARS ? _, ma_resp ->
if
::skip;
::skip;
fi;
::chan_GMASRQ ? _, gmas_rqst  ->
if
::skip;
fi;
::chan_GMASRS ? _, gmas_rqst  ->
if
::skip;
fi;
od;
}
```

B SPIN Output

```
pan: assertion violated 0 (at depth 11)
pan: wrote mobiilid_auth.pml.trail

(Spin Version 5.2.2 -- 7 September 2009)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          - (none specified)
assertion violations +
acceptance   cycles  - (not selected)
invalid end states +

State-vector 180 byte, depth reached 11, errors: 1
       11 states, stored
        0 states, matched
       11 transitions (= stored+matched)
```

```
        0 atomic steps
hash conflicts:         0 (resolved)

    4.653 memory usage (Mbyte)


pan: elapsed time 0 seconds
1:  DigiDocService(1):[partner = app_provider]
2: AppProvider(0):[partner = digidocservice]
3: AppProvider(0):[((session_status==0))]
4: AppProvider(0):[ma_rqst.key = partner_key]
5: AppProvider(0):[chan_MARQ!partner,ma_rqst.IDCode,ma_rqst.Country,ma_rqst.PhoneNo,ma_r
6:  Intruder(2):[chan_MARQ?_,ma_rqst.IDCode,ma_rqst.Country,ma_rqst.PhoneNo,ma_rqst.Lang
7:  Intruder(2):[(1)]
8: AppProvider(0):[session_status = 1]
9: AppProvider(0):[partner = digidocservice]
10: AppProvider(0):[((session_status==1))]
11: AppProvider(0):[(timeout)]
pan: assertion violated 0 (at depth 12)
spin: trail ends after 12 steps
#processes 3:
 12: proc 0 (AppProvider)  line  94 (state 31) (invalid end state)
assert(0)
 12: proc 1 (DigiDocService)  line 151 (state  7) (invalid end state)
chan_MARQ?digidocservice,ma_rqst.IDCode,ma_rqst.Country,ma_rqst.PhoneNo,ma_rqst.Language
 12: proc 2 (Intruder)  line 200 (state 20) (invalid end state)
chan_MARQ?_,ma_rqst.IDCode,ma_rqst.Country,ma_rqst.PhoneNo,ma_rqst.Language,ma_rqst.Serv
chan_MARS?_,ma_resp.Sesscode,ma_resp.Status,ma_resp.UserIDCode,ma_resp.UserGivenName,ma_
chan_GMASRQ?_,gmas_rqst.Sesscode,gmas_rqst.WaitSignature,gmas_rqst.key
chan_GMASRS?_,gmas_rqst.Sesscode,gmas_rqst.WaitSignature,gmas_rqst.key
global vars:
chan chan_MARQ (=1): len 0:
chan chan_MARS (=2): len 0:
chan chan_GMASRQ (=3): len 0:
chan chan_GMASRS (=4): len 0:
local vars proc 0 (AppProvider):
byte   session_status: 1
byte   authreq_status: 0
(struct ma_rqst)
bit    ReturnCertData: 0
bit    ReturnRevocationData: 0
(struct ma_resp)
(struct gmas_rqst)
(struct gmas_resp)
local vars proc 1 (DigiDocService):
```

```
(struct ma_rqst)
bit    ReturnCertData: 0
bit    ReturnRevocationData: 0
(struct ma_resp)
(struct gmas_rqst)
(struct gmas_resp)
local vars proc 2 (Intruder):
(struct ma_rqst)
bit    ReturnCertData: 0
bit    ReturnRevocationData: 0
(struct ma_resp)
(struct gmas_rqst)
(struct gmas_resp)
```