A Formal Security Analysis of an OSA/Parlay Authentication Interface

R. Corin[1], G. Di Caprio[3], S. Etalle[1], S. Gnesi[2], G. Lenzini[4,5], and C. Moiso[3]

[1] Department of Computer Science, University of Twente
7500 AE Enschede, The Netherlands
{coring,etalle,lenzinig}@cs.utwente.nl

[2] Istituto di Scienza e Tecnologie dell'Informazione, ISTI-CNR
Area della Ricerca di Pisa, Via G. Moruzzi 1, 56124 Pisa, Italy
gnesi@isti.cnr.it

[3] Telecom Italia Lab
Via G. Reiss Romolo 274, 1048 Torino, Italy
{corrado.moiso,gaetano.dicaprio}@tlab.com

[4] Telematica Instituut
Brouwerijstraat 1 – 7523 XC Enschede, The Netherlands
gabriele.lenzini@telin.nl

[5] Istituto di Informatica e Telematica, IIT-CNR
Area della Ricerca di Pisa, Via G. Moruzzi 1, 56124 Pisa, Italy
gabriele.lenzini@iit.cnr.it

Abstract. We report on an experience in analyzing the security of the Trust and Security Management (TSM) protocol, an authentication procedure within the OSA/Parlay Application Program Interfaces (APIs) of the Open Service Access and Parlay Group. The experience has been conducted jointly by research institutes, experienced in security, and an industry experts in telecommunication networking. OSA/Parlay APIs are designed to enable the creation of telecommunication applications outside the traditional network space and business model. Network operators consider the OSA/Parlay architecture promising for stimulating the development of web-service applications by third party providers which are not necessarily expert in telecommunication and security. The TSM protocol is executed by the gateways to OSA/Parlay networks; its role is to authenticate client applications trying to access the interfaces of some object representing an offered network capability. For this reason, potential security flaws in the TSM authentication strategy can cause the unauthorized use of network with evident damages to the operator and to the quality of services. This paper reports the rigorous formal analysis of the TSM specification originally given in UML; the design activity of the formal model, the tool-aided verification performed, and the security flaws discovered.

Keywords: Formal Verification of Security, OSA/Parlay API, Industrial Test Case.

1 Introduction

OSA/Parlay [1] Application Program Interfaces (APIs) [9] are designed for an easy interaction between traditional IT applications and telecommunication networks. OSA/Parlay APIs are abstract building blocks of network capabilities that developers, not necessarily expert in telecommunications but perhaps with more expertise in the enterprise market, can quickly comprehend and use to generate new applications. Concisely, OSA/Parlay APIs proposes an attractive framework where programmers can develop innovative resources or design new services.

An example of such a service is the retrieval and purchase of goods via a mobile phone. The service could be provided by a third party, different from the mobile operator. In this case, the provider could develop the service by assembling components that control network capabilities and functions, for example, sending/receiving a SMS. These components (in particular, their APIs) are provided by the telecoms operator. For example, the sending/receiving of a SMS could be realized in the following SOAP body that, in XML notation where namespace and encoding descriptors are omitted, appears as follows:

```
<sendSMS>
 <dest_address>
   tel:1234567
 </dest_address>
 <send_address>
   tel:0123456
 </send_address>
 <message>
   Could you please reserve
   two seats for 9 o'clock?
 </message>
</sendSMS>
```

OSA/Parlay APIs can also be used in the development of new web-based services. To this end, the Parlay community has designed specific APIs, called Parlay X APIs, based on web service principles and oriented to the Internet community.

When network resources are broadly accessible, it becomes crucial to define and enforce appropriate access rules between entities offering network capabilities and service suppliers, so that an operator can maintain full control over the usage of her resources and on the quality of service. For instance, it is important that the use of services is guided by a set of rules defining the supply conditions and the reciprocal obligations between the client and the network operator. Service Level Agreements (SLAs) are commonly used to formalize a detailed description of all the aspect of the deal. To avoid that unauthorized entities can sign an agreement and use the network illegally, on-line authentication checks are of primary importance.

[1] See `http://www.parlay.org`

Authentication in a distributed setting is usually achieved by the use of cryptographic protocols. Experience teaches that these protocols need to be carefully checked, before being fielded (e.g., [2, 5, 8, 11, 12, 15, 16]), and nowadays developers have access to libraries of reliable protocols for different security goals. For example the Secure Socket Layer (SSL) by Netscape, is widely used to ensure authenticity and secrecy in Internet transactions. Unfortunately, the use of reliable, plugged-in, protocols is not sufficient to ensure security, just like the use of reliable cryptography is not sufficient to ensure secrecy in a communication. As we shall see formal methods can help to validate the correct use of security procedures.

In this paper we discuss the validation of the authentication mechanism in the Trust and Security Management (TSM) protocol in OSA/Parlay APIs [1]. This protocol is designed to protect telecommunication capabilities from unauthorized access and it implements an authentication procedure. TSM is specified in the UML [14], where its composing messages, its interfaces towards the client and the server, and the methods implementing security-critical procedures, are described at different levels of abstraction. The formal validation experiment, conducted within a joint project between research Institutes and Telecom Italia Lab, has revealed some security flaws of the authentication mechanism. From the analysis of the traces showing the attacks, we were able to suggest possible solutions to fix the security weaknesses discovered, and to state a general principle of prudent engineering (in the style of [4]) for improving the security in web-service applications.

2 The OSA/Parlay Architecture

The OSA/Parlay architecture enables service application developers to make use of network functionality through an open standardized interface. OSA/Parlay APIs [1] provide an abstract and coherent view of heterogeneous network capabilities, and they allow a developer to interface its applications via distributed processing mechanisms. The OSA/Parlay architecture, shown in Figure 1, consists of:

- a set of *Client Applications* accessing the network resources;
- a set of *Service Interfaces*, or Service Capability Features (SCFs), that represent interfaces for controlling the network capabilities provided by network resources (*e.g.,* controlling the routing of voice calls, sending/receiving SMSs, locating a terminal, etc.);
- a *Framework*, that provides a modular and "controlled" access to the SCFs.
- *Network Resources*, in the telecommunication network, implementing the network capabilities.

A *Parlay Gateway* includes the framework functions and the Service Capability Services (SCSs), that is the modules implementing the SCFs: it is a logical entity that can be implemented in a distributed way across several systems.

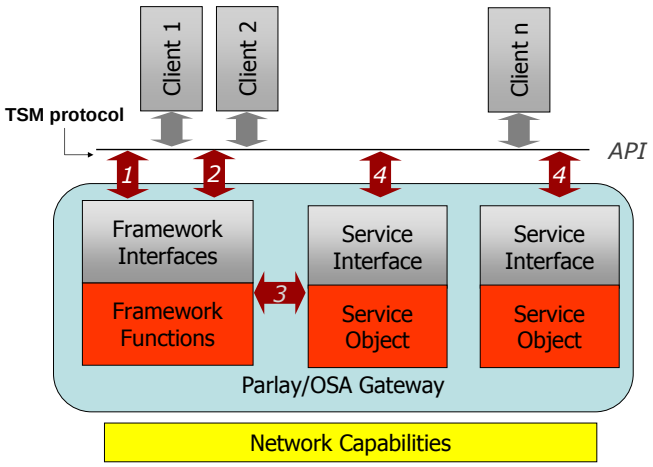

Fig. 1. The OSA/Parlay Architecture. The Trust and Security Management protocol runs between the Framework Interfaces and the Clients.

Since the target applications could be deployed in an administrative domain different from the one of the Parlay Gateway, the secure and controlled access to the SCFs is a predominant aspect for the Parlay architecture. To get the references of the required SCFs, an application must interact several times with the framework interfaces. For example, the application must carry out an authentication phase before selecting the SCFs required, as described in Section 2.1. In this phase the framework verifies whether the application is authorized to use the SCFs, according to a subscription profile. Finally, an agreement is digitally signed, and the framework gives to the application the references to the required SCFs (*e.g.,* as CORBA interface reference). These references are valid only for a single session of the application. When the framework has to return an SCF reference to an application, it contacts the SCS which implements it, by passing all the configuration parameters, for instance the Service Level Agreement conditions, stored in the subscription profile of the application. The SCS creates a new instance of the SCF, configured with the received parameters, and returns its reference to the framework. Each time the application invokes a method on the SCF instance, the SCS executes it by taking into account the configuration parameters received at instantiation time.

Gateways based on the OSA/Parlay framework here presented have been implemented by, for instance, Ericsson, Alcatel, Lucent, AePONA, and Incomit (though they have not been deployed yet).

2.1 Trust and Security Management protocol

One of the critical steps for guaranteeing controlled access to the SCFs is the authentication phase between the gateway and the application. It is supported by the protocol implemented by the Trust and Security Management (TSM) API. We focuses on the analysis of the properties of this security protocol, whose behavior is summarized by the message sequence chart in Figure 2. The main steps of the protocol are:

- Initiate Authentication: the client invokes "`initiateAuthenticationWithVersion`" on the framework's *public* interface (*e.g.,* an URL) to initiate the authentication process. Both the client and the framework provide a reference to their own access interfaces.
- Select Authentication Mechanism: the client invokes "`selectAuthenticationMechanism`" on the framework authentication interface, to negotiate which hash function will be used in the authentication steps.
- The client and the framework authenticate each other. The framework could authenticate the client before (or after) the client authenticates the framework, or the two authentication processes could be interleaved. However, the client shall respond immediately to any challenge issued by the framework, as the framework might not respond to any challenge issued by the client until the framework has successfully authenticated the client. Each authentication step is performed following a one-way Challenge Handshake Authentication Protocol (CHAP) [10], that is by issuing a challenge in the "challenge" method, and checking if the partner returns the correct response. An invocation of the method "`authenticationsucceeded`" signals the success of the challenge.
- Request an access session: when authenticated by the framework, the client is permitted to invoke "`requestAccess`" to start an access session. The client provides a reference to its own Access interface, and the framework returns a reference to Access interface, unique for this client.
- The access interface is used to negotiate the signing algorithm to be used in the session and to obtain the references to other framework interfaces (we will call them, *service framework interfaces*), such as service discovery and service agreement management.

Having obtained the reference to a service framework interface the TSM finishes. Note that the references to the interfaces must remain secret: if an intruder got hold of them, it would be able to (abusively) access the services. For this reason our analysis will mainly concentrate on the secrecy of these references.

In fact, after the TSM ends, the client selects the required SCFs by invoking the "`selectService`" method on the service agreement management interface. The client obtains a service token, which can be signed as part of the service agreement by the client and the framework, through the "`signServiceAgreement`" and the "`signAppServiceAgreement`" methods. Generally the service token has a limited lifetime: if the lifetime of the service token expires, a method

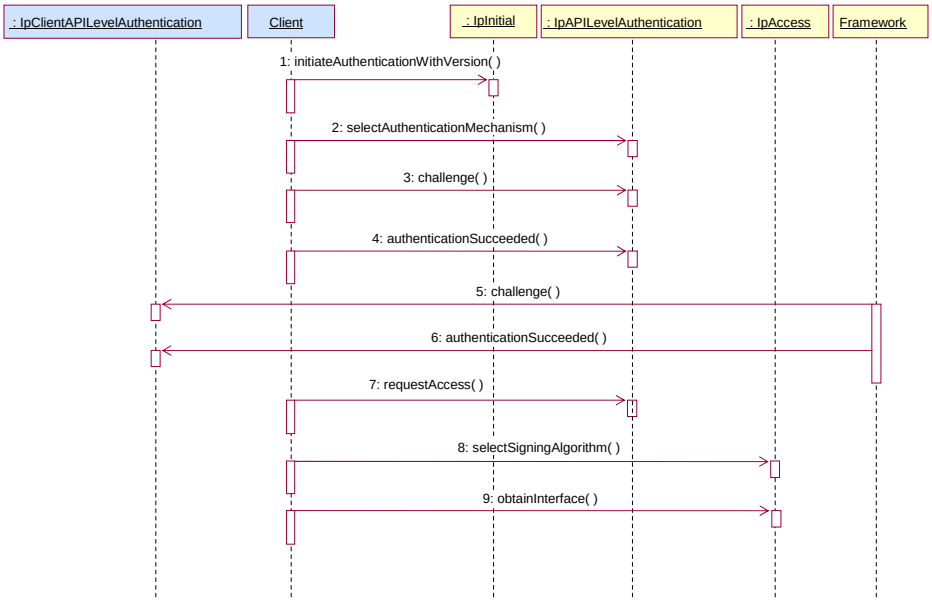

Fig. 2. Message sequence chart describing the steps of the TSM protocol [1]

receiving the service token will return an error code. If the sign service agreement phase succeeds, the framework returns to the client a reference to the selected SCF, personalized with the client configuration parameters.

3 Security Formal Analysis

This section explains in detail the formal analysis of the security of the TSM protocol that we have done. To carry out the verification phase we used CoProVe [6] a constraint-based system for the verification of cryptographic protocols [2]. CoProVe has been developed at the University of Twente (NL); it is an improved version of the system designed by Millen and Shmatikov [13]. CoProVe is based on the strand spaces model [17]; it enjoys an efficient implementation, a monotonic behavior which allows to detect flaws associated to partial runs, and an expressive syntax in which a principal may also perform explicit checks for deciding whether to continue or not with the execution. All these features make

[2] Freely accessible via the web at `http://wwwes.cs.utwente.nl/24cqet/coprove.html`

CoProVe quite efficient in practice. The intruder model is that of Dolev-Yao [7], where the malicious entity is identified with the communication infrastructure. Protocols are written in Prolog-lake style, and properties are expressed as reachability predicates. In case a security flaw is discovered, CoProVe can show one or all the traces showing the attack.

3.1 Modeling Choices

One of the challenges in applying tools of automatic analysis to industrial architectures lies in translating the (usually less formal) specification into a rigorous formal model. In our experience, translating a complex system design into a formal protocol specification involves many non-trivial steps: software technology concepts such as method invocation and object interfaces have to be "encoded" into an algebraic protocol specification. This encoding phase also forces the engineer to reason about the security implication of using these constructs.

The OSA/Parlay framework APIs specification consists of many pages of UML specification; at this level of abstraction it is difficult to have a good overview of its security mechanisms. In the APIs specification, for instance, there is no explicit transmission of messages: the exchange of one (sometimes even more) messages happens exclusively by the mechanism "invocation of a method over an object interface". Moreover, different levels of abstraction are mixed: for example, the same mechanism of "method invocation" is used both to describe, in one step, the whole set of critical steps of the CHAP handshake and the single message starting of the protocol. More critically, "method invocation" does not specify the confidentiality of the input/output parameters involved. Innocent acknowledgment messages are treated in the same way as references to confidential object interfaces.

The application of clear modeling choices encourages the design of a formal model without the previous ambiguities. In translating the TSM specification in a model we define and apply the following modeling choices.

Modeling Choice 1 *A reference to a (new) private interface, F, is modeled by a (new) shared encryption key, KF.*

Choice 1 reflects the fact that an intruder who does not know the private interface reference cannot infer anything from any method invocation over that interface. This simple, but essential observation will make our security analysis straightforward, as we explain in Section 3.

Modeling Choice 2 *Calling a method, with parameter M, over a private interface F is modeled as sending the message $\{M\}_{KF}$* i.e., *, M encrypted with KF. Dually, getting the result is translated as receiving a message encrypted with the same KF;*

In Choice 2 we treat a reference to an object interface as a communication port; consequently calling a method equates transmitting a message through that port. Moreover, we model the transmission of a message through F, as

the transit of a message encrypted with the key KF. In other words, calling a method over an interface is modeled as a communication encrypted with the interface key. This choice reminds of an observation by Abadi and Gordon [3], who suggest the use of cryptographic keys to model mobility. Our situation is indeed much simpler: the only form of "mobility" we have, is the dynamic creation of a "channel", that is an interface reference.

3.2 Formal Models

We apply Choices 1 and 2 to design the TSM formal *abstract* model written in the usual representation of cryptographic protocols. The obtained model is as follows:

* initiate *	* request access *
step 1. $C \longrightarrow F : C, KC$	step 8. $C \longrightarrow F : \{req\}_{KF}$
step 2. $F \longrightarrow C : KF$	step 9. $F \longrightarrow C : \{KA/fail\}_{KF}$
* select authentication methods *	* select signing methods *
step 3. $C \longrightarrow F : \{[h, h', h'']\}_{KF}$	step 10. $C \longrightarrow F : \{[s, s', s'']\}_{KA}$
step 4. $F \longrightarrow C : \{h\}_{KF}$	step 11. $F \longrightarrow C : \{s\}_{KA}$
* challenge *	* request for service interface *
step 5. $F \longrightarrow C : \{F, N\}_{KC}$	step 12. $C \longrightarrow F : \{req'\}_{KA}$
step 6. $C \longrightarrow F : \{C, h(N, SCF)\}_{KC}$	step 13. $F \longrightarrow C : \{KS/fail\}_{KA}$
step 7. $F \longrightarrow C : \{ok/fail\}_{KC}$	

In this abstract model, C represents a client and F the framework, while $C \longrightarrow F : M$ denotes C sending message M to F. With $\{M\}_K$ we indicate the plain-text M encrypted with a key K, while $h(M)$ denotes the result of applying a hash function h to M. In step 1 the client initiates the protocol over the public interface of the framework, by providing its name and a reference to its interface, KC. In step 2 the framework replies by sending a reference, KF, to its own interface.

Remark 1. It may seem odd that despite modelling choice we transmit references to interfaces (represented as keys) in clear. The expectation here is that the challenge response protocol of steps 5-7 would avoid intrusion anyway.

In steps 3 and 4 the client asks the framework to choose an authentication method among h, h' and h''. In steps 5 and 6 the actual CHAP protocol is carried out, using the hash function selected in step 4. Here, SCF represents a shared secret between C and F, required by CHAP [10]. Indeed the UML specification did not provide the details about the CHAP implementation; here we use the version of CHAP where the client and the framework already share the secret SCF. In steps 8 and 9 the client asks for an interface where to invoke

```
% Initiator role specification
client(C,F,Kc,Kf,N,Req,Ka,Scf,[
   send([C,Kc]),
   recv(Kf),
   recv([F,N]+Kc),
   send([C,sha([N,Scf])]+Kc),
   send(Req+Kf),
   recv(Ka+Kf)]).

% Responder role specification
framewk(C,F,Kc,Kf,N,Req,Ka,Scf,[
   recv([C,Kc]),
   send(Kf),
   send([F,N]+Kc),
   recv([C,sha([N,Scf])]+Kc),
   recv(Req+Kf),
   send(Ka+Kf)]).

% Secrecy check
%(it is a singleton role)
secrecy(N, [ recv(N) ] ).
```

```
% scenario specification
% pairs [name, Name]
% [label for the role; actual role]
scenario
([[c,Client1],
  [f,Framew1],
  [sec,Secr1]]):-

client(c,f,kc,_,_,req,_,scf,Client1),
framewk(c,f,_,kf,n,_,ka,scf,Framew1),
secrecy(ka, Secr1).

% The initial intruder knowledge
initial_intruder_knowledge([c,f,e]).

% specify which roles we want
% to force to finish
%(only sec in this example)
has_to_finish([sec]).
```

Fig. 3. The "CoProVe" specification (in two columns) used to check the secrecy of KA. To reduce the search space here we implemented only steps 1-2, 5-6 and 8-9. In other words we assumed: (a) a constant hashing function h; (b) that the framework does not reply (instead of replying "false") if the client answer wrongly to the CHAP challenge.

the request access for a service. In steps 10 and 11 the framework chooses the interface. Finally in steps 12 and 13 the client sends a request for a service and receives back the reference to the relative framework interface.

The abstract model has been translated into the language required by CoProVe. The result of this translation is a *concrete* formal model; in addition, we encode (in the language of CoProVe) the security properties that we want to check. In Figure 3 we report one of the concrete models we used for checking whether KA remains secret or not.

The specification in Figure 3 involves three principals: one client (`c`), one framework (`f`) and eavesdropping agent (`sec`). Each role is specified by a sequence of send or receive actions that mimic exactly the steps of the abstract model. Symbol "`+`" is used to denote symmetric encryption using shared keys. Formal parameters (*e.g.,* in the client role`C,F,Kc,Kf,N,Req,Ka,Scf`) are used to denote all the objects used in the role specification. In a scenario these parameters are instantiated with actual constants representing real objects (*i.e.,* `c,f,_,kf,n,_,ka,scf`). Here "`_`" is used when no instantiation is required, that is when a free variable is involved. The intruder is assumed to know only the

client and framework names plus its own name "`e`". Verification of secrecy consists in asking if there is a trace leading the eavesdropper to know a secret.

3.3 Formal Analysis and Detected Weakness

The analysis performed on the model of TSM protocol, pointed out weaknesses in the security mechanism. In the following we will describe the flaws discovered as a commented list of items. Where significant, we show the output produced by CoProVe and we interpret the output.

Flaw 1. An intruder can impersonate a client and start an authentication challenge with the framework.

An intruder can obtain the reference to the interface used by the client to start the authentication challenge (key `kf`). This happens, unsurprisingly, because the reference `kf` is transmitted in clear, as the following trace of CoProVe confirms:

```
1.  [c,send([c,kc])]
1'. [f,recv([c,kc])]
2.  [c,recv(_h325)]
2'. [f,send(kf)]
```

Each row represents a communication action. For example, `c,send[c,kc]` represents the action "send" that "`c`" executes with message "`[c,kc]`"; `c,recv(_h325)` represents the results of a "receive" where the client "`c`" receives the name (in this case generated by the intruder) " `_h325`". The sequence of actions reveal the attack. It can be visualized in the conventional notation of security protocol (where, we also write `_h325` as KE, the intruder key, because this is its understood meaning.):

$$\begin{array}{llll} 1. & C \longrightarrow I(F) : C, KC & 2. & I(F) \longrightarrow C : KE \\ 1'. & I(C) \longrightarrow F : C, KC & 2'. & F \longrightarrow I(C) : KF \end{array}$$

This run comprises two parallel runs of the protocol, in which the intruder plays, respectively, the role of the client against the framework ($I(C)$ in *steps* $1'$ and $2'$) and the framework against the client ($I(F)$ in steps 1 and 2).

This flaws is not serious in itself (provided the authentication procedure is able to detect an intruder and close the communication), but it becomes serious when combined with the next weaknesses in the security; by knowing *kf* an intruder is able to grab other confidential information.

Flaw 2. An intruder can impersonate a client, authenticate itself to the framework and obtain the reference to the interface used to request access to a service (key *ka*).

This is a serious flaw that compromises the main goal of the protocol itself. Informally, a malicious application can pass the authentication phase instead of an honest client, and it can obtain a reference to the interface used to request a service (key *ka*). The study of the output of CoProVe (here depicted in two columns) shows the existence of an "oracle" attack, where the intruder uses the client to get the right answer to the challenge:

```
1.  [c,send([c,kc])]          6.  [c,send([c,sha([n,scf])] + kc)]
1'. [f,recv([c,kc])]          6'. [f,recv([c,sha([n,scf])] + kc)]
2.  [c,recv(_h325)]           8.  [c,send(req + _h325)]
2'. [f,send(kf)]              9.  [c,recv(_h391 + _h325)]
5'. [f,send([f,n] + kc)]      8'. [f,recv(req + kf)]
5.  [c,recv([f,n] + kc)]      9'. [f,send(ka + kf)]
                                  [sec,recv(ka)]
```

Using the standard informal notation for describing protocols, the above trace is read as follows:

$$
\begin{array}{lll}
1. & C \longrightarrow I(F) : C, KC & 6. \quad C \longrightarrow I(F) : \{C, h(N, SCF)\}_{KC} \\
1'. & I(C) \longrightarrow F : C, KC & 6' \quad I(C) \longrightarrow F : \{C, h(N, SCF)\}_{KC} \\
2. & I(F) \longrightarrow C : KE & 8. \quad C \longrightarrow I(F) : \{req\}_{KE} \\
2'. & F \longrightarrow I(C) : KF & 9. \quad I(F) \longrightarrow C : \{fail\}_{KE} \\
5'. & F \longrightarrow I(C) : \{F, N\}_{KC} & 8'. \quad I(C) \longrightarrow F : \{req\}_{KF} \\
5. & I(F) \longrightarrow C : \{F, N\}_{KC} & 9'. \quad F \longrightarrow I(C) : \{KA\}_{KF}
\end{array}
$$

This run comprises two parallel runs of the protocol, in which the intruder plays, respectively, the role of the framework against the client and the role of the client against the framework. Searching among the set of attacks returned by CoProVe, we find also the following, straightforward, man-in-the-middle, attack:

```
1.  [c,send([c,kc])]          6.  [c,send([c,sha([n,scf])] + kc)]
1'. [f,recv([c,kc])]          6'. [f,recv([c,sha([n,scf])] + kc)]
2'. [f,send(kf)]              8.  [c,send(req + kf)]
2.  [c,recv(kf)]              8'. [f,recv(req + kf)]
5'. [f,send([f,n] + kc)]      9'. [f,send(ka + kf)]
5.  [c,recv([f,n] + kc)]      9.  [c,recv(_h325)]
                                  [sec,recv(ka)]
```

This trace shows that the intruder can eavesdrop first the key `kf`, passed in clear, and then steal the message `ka+kf`. At this point key `ka` can be obtained by a simple decryption. This attack is obviously straightforward at this point of the analysis, but it became clear as soon as we applied Choice 1.

Flaw 3. An intruder can impersonate a client, authenticate itself to the framework, send a request for a service and obtain the reference to a service framework interface (key *ks*).

This is also a serious flaw that compromises the main goal of the protocol. An intruder can obtain the reference to a service framework interface (key *ks*). It is easy to understand, that this is possible, for example, as a consequence of flaw 1 and 2: once an intruder has authenticated itself instead of the client, it can easily obtain the reference.

```
% Initiator role specification
client(C,F,Kc,Kf,N,Req,Scf,
       Ka,A1,A2,A,[
   recv([C,F]),
   send([C,Kc]),
   recv(Kf),
   send([A1,A2]+Kf),
   recv([A,A]+Kf),
   recv([F,N]+Kc),
   send([C,sha([N,Scf])]+Kc)
   send(Req+Kf),
   recv(Ka+Kf)]).

% Responder role specification
framewk(C,F,Kc,Kf,N,Req,Scf,
        Ka,A1,A2,[
   recv([C,Kc]),
   send(Kf),
   recv([A1,A2]+Kf),
   send([A1,A1]+Kf),
   send([F,N]+Kc),
   recv([C,sha([N,Scf])]+Kc),
   recv(Req+Kf),
   send(Ka+Kf)]).

framewk2(C,F,Kc,Kf,N,Req,
          Ka,A1,A2,[
   recv([C,Kc]),
   send(Kf),
   recv([A1,A2]+Kf),
   send([A2,A2]+Kf),
   recv(Req+Kf),
   send(Ka+Kf)]).
```

```
% secrecy check (singleton role)
secrecy(N, [ recv(N) ] ).

% Scenario
scenario([
    [c,Client1],
    [f,Framew1],
    [f,Framew2],
    [sec,Secr1]
         ]) :-
client(c,f,kc,_,_,req,scf,_,a1,
        a2,a1,Client1),

framewk(c,f,_,kf,n,_,scf,ka,_,_,
        Framew1),

framewk2(c,f,_,kf2,n2,_,ka2,_,_,
        Framew2),

secrecy(ka2, Secr1).

% Set up the intruder knowledge
initial_intruder_knowledge([c,f,e]).

% specify which roles we want
% to force to finish
% (only sec in this example)
has_to_finish([sec]).
```

Fig. 4. The "CoProVe" code used to discover flaw 4 (in two columns). The model of the framework includes the "select authentication method" phases of the abstract model and implements steps 1–9 of the abstract model. Step 7 is omitted, *i.e.,* the framework does not reply (instead of sending "fail") in case of failure of the challenge phase. The second instance of the framework models only steps 1–4 and steps 8–9, that is those steps strictly necessary to discover the attack.

Further checks with CoProVe, show that the intruder can even retrieve this reference with a man-in-the-middle attack, for instance, by listening to the communication between the client and the framework and stealing the reference when it is passed in clear. In our model this attack can be explained as follows: the intruder intercepts, by eavesdropping, the message $\{KS\}_{KA}$ and it decrypts it. This is possible because the encryption key KF is passed in clear and, by eavesdropping, the intruder can easily obtain $\{KA\}_{KF}$, and hence KA (flaw 2).

Flaw 4. An intruder can force the framework to use an authentication mechanism of her choice.

This flaw has been discovered using the specification in Figure 4, with two instances of the framework. When a client offers a list of authentication methods, the first instance selects the first method at the head of a list (here consisting of only two items), whereas the second instance choses the second. In this way we model different choices made by the framework.

The attack is shown by the following CoProVe trace; an intruder can force the framework to select a particular authentication mechanism, by the use of a replay attack.

```
a.1.  [c,send([c,kc])]                  a.6'. [f,recv([c,sha([n,scf])] + kc)
a.1'. [f,recv([c,kc])]                  a.8.  [c,send(req + _h320)]
a.2.  [c,recv(_h320)]                   a.9.  [c,recv(req + _h320)]
a.2'. [f,send(kf)]                      a.8'. [f,recv(_h404 + kf)]
a.3.  [c,send([a1,a2] + _h320)]         a.9'. [f,send(ka + kf)]
a.3'. [f,recv([a1,a2] + kf)]            b.1'. [f,recv([c,_h487])]
a.4'. [f,send([a1,a1] + kf)]            b.2'. [f,send(kf2)]
a.4.  [c,recv([a1,a1] + _h320)]         b.3'. [f,recv([a1,a1] + kf2)]
a.5'. [f,send([f,n] + kc)]              b.4'. [f,send([a1,a1] + kf2)]
a.5.  [c,recv([f,n] + kc)               b.8'. [f,recv(_h488 + kf2)]
a.6.  [c,send([c,sha([n,scf])] + kc)    b.9'. [f,send(ka2 + kf2)]
                                              [sec,recv(ka2)]
```

The attack can represented in the following abstract steps:

a.1	$C \longrightarrow I(F1) : C, KC$	*a.4'*	$F1 \longrightarrow I(C) : \{[a1]\}_{KF}$
a.1'	$I(C) \longrightarrow F1 : C, KC$		$[\ldots]$
a.2	$I(F) \longrightarrow C : KE$	b.1'	$I(C) \longrightarrow F2 : C, KE$
a.2'	$F1 \longrightarrow I(C) : KF$	b.2'	$F2 \longrightarrow I(C) : KF2$
a.3	$C \longrightarrow I(F1) : \{[a1, a2]\}_{KE}$	b.3'	$I(C) \longrightarrow F2 : \{[a1, a1]\}_{KF2}$
a.3'	$I(C) \longrightarrow F1 : \{[a1, a2]\}_{KF}$	b.4'	$F2 \longrightarrow I(C) : \{[a1]\}_{KF2}$

In the trace the intruder acts as a men-in-the-middle in a communication between the client and the first instance of the framework $F1$ and it learns what method the framework is able to use (sequences $a.i$). In the second run, the intruder acts as a client, and it offers to the second instance of the framework $F2$ the choice that the framework is able to accept (sequences $b.i$). The structure of the attack

is such that it can be applied also for forcing the selection of a signing methods, that is steps 10 and 11 of the abstract model.

4 Discussion

The analysis performed so far shows some weaknesses of the protocol, and gives also useful indications on how to improve the robustness of the protocol. This section discusses the weaknesses here presented, and suggests possible solutions to increase the overall security. We start with some preliminary considerations.

The security is weak is because some references to interfaces are passed in the clear. This is because the role of those references has been misunderstood, or under-evaluated, or more probably not recognized in the UML, high-level, object specification. A rigorous, synthetic, formal specification and precise modeling choices helps in giving each object its right role. In our case we were able to identify in the role of some references to object interface the same role that session keys have. This observation can be quoted as a principle:

> Independently of their high-level representation, data that directly or indirectly gives access to a secret, must be thought of (hence, modeled) as encryption keys.

This principle plays a role also in fixing the protocol. In fact, the common practice in protocol engineering [4] suggests the use of (other) session keys to protect the confidentiality of sensitive information, which in the case of TSM are the references to interfaces. According to the TSM model, session keys are indeed missing completely from the present implementation[3] while their use could prevent the intruder from gaining a reference to an interface (as shown, by a men-in-the-middle attack).

An additional point of discussion concerns the correct use of a CHAP-based authentication. From the OSA/Parlay documentation ([1] page 19) we read that *security can be ensured if the "challenge" is frequently invoked by the framework to authenticate the client that, in turn, must reply "immediately"*:

Our analysis proves that not only the intruder can act as a client with respect to the framework, but also that it can passively observe, as man-in-the-middle, the framework and a client authenticating each other as many times as they want, and then steal the reference to the service framework interfaces when they are transmitted in clear. At this point the intruder can substitute itself for the client.

Generally speaking TSM confidentiality improves if the framework encrypts all the messages containing a reference to an interface. Encryption requires that the framework authenticates the client, and later that it agrees upon a session key with the authenticated client. This can be done, for example by "running a Secure Sockets Layer (SSL) protocol at the beginning of the TSM session. The

[3] Do not confuse them with the session keys that appear in the abstract model. Those are part of the model and represent private references to interfaces.

SSL allows two entities, a client and a server, to authenticate each other and to establish session keys. Session keys are then used to ensure confidentiality and integrity in any, next, exchange of messages. As a consequence, the SSL can substitute the CHAP authentication procedure required by the TSM specification. The common use of the SSL sees the client to authenticate the server (*i.e.,* , the framework in our case); in the context of the TSM security, is mandatory that the server authenticates the client as well.

Flaw 4 is different in nature, and it teaches that particular care must be paid to the choice of the encryption algorithms or digital signature procedures offered by the framework: for example, the intruder can force the system to use the encryption algorithm that is easier to crack.

5 Conclusions

This paper discusses an industrial experience of formal analysis applied to the security aspects of the OSA/Parlay Trust and Security Management protocol. The protocol is devised to authenticate clients before granting them access to network services. Our experience confirms that formal methods are an invaluable tool for discovering serious security flaws which may be overlooked otherwise. This is true in two respects. First, the use of a formal model, where only the relevant security features are expressed, helps at pointing out what are the critical security component. In an informal description, on the other hand, this information is usually dispersed and difficult to gather. Second, the use of an automatic tool allows one to identify dangerous man-in-the middle attacks, which are notoriously difficult to see on high-level specifications.

From this experience, conducted within a joint project between industry and research institutes, we state a general principle for security in web-services: it is essential to identify clearly the security role of each object involved in service specification. It is vital especially for those objects that abstractly represent encryption keys. This principle helps at simplifying the security analysis. With the application of this principle we discover serious weaknesses more easily, and we are able to discuss how the security of the TSM protocol can be generally improved.

The results of this work has been presented to the join standardization group 3GPP/ETSI/Parlay. They have decided to open a study on how to strengthen the security of OSA/Parlay in the next future.

6 Acknowledgment

S. Gnesi and G. Di Caprio, and C. Moiso were supported by the MIUR-CNR Project SP4. R. Corin was supported by the IOP GenCom project PAW; S. Etalle was partially supported by the BSIK project BRICKS; G. Lenzini was supported by SP4, PAW and by the IIT-CNR project “Trusted e-services for Dynamic and Mobile Coalitions”.

References

[1] *Open Service Access (OSA) - Application Programming Interface (API) Mapping for OSA*. http://www.3gpp.org/ftp/Specs/archive/29_series. Release 5.

[2] M. Abadi and A. D. Gordon. Reasoning about Cryptographic Protocols in the Spi Calculus. In A. W. Mazurkiewicz and J. Winkowski, editors, *Proc. of 8th Int. Conf. on Concurrency Theory (CONCUR 97), Warsaw, Poland, July 1997*, *LNCS* 1243, pages 59–73. Springer-Verlag, 1997.

[3] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols. The Spi Calculus. TR 149, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, USA 1998.

[4] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996. IEEE Computer Society.

[5] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Trans. Software Engineering and Methodology*, 9(4):443–487, 2000. ACM .

[6] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In M. Hermenegildo and G. Puebla, editors, *Proc. of the International Static Analysis Symposium (SAS), Madrid, Spain, Sep. 2002*, *LNCS* 2477, pages 326–341. Springer-Verlag, 2002.

[7] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Trans. Information Theory*, 29(2):198–208, 1983. IEEE Computer Society.

[8] R. Gorrieri, F. Martinelli, M. Petroocchi, and A. Vaccarelli. Formal analysis of some timed security properties in wireless protocols. In *Proc. of the 6th IFIP WG 6.1 Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003) Paris, France, Nov. 2003*, *LNCS* 2884, pages 139–154. Springer Verlag, 2003.

[9] Parlay X Working Group. Parlay apis 4.0: Parlay x web services - white paper. The Parlay Group, 2002. http://www.parlay.org.

[10] G. Leduc. Verification of two versions of the challenge handshake authentication protocol (CHAP). *Annals of Telecommunications*, 55(1-2):18–30, 2000. Hermes-Lavoisier.

[11] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. *Software Concepts and Tools*, 3(17):93–102, 1997. Springer-Verlag.

[12] C. A. Meadows. Formal verification of cryptographic protocols: A survey. In J. Pieprzyk and R. Safavi-Naini, editors, *Proc. of the Int. Conf. on the Theory and Application of Cryptology Advances in Cryptology and Information Security, (ASIACRYPT 94)*, *LNCS* 917, pages 135–150. Springer-Verlag, 1994.

[13] J. Millen and V. Shmatikov. Constraint Solving for Bounded-Process Cryptographic Protocol Analysis. In P. Samarati, editor, *Proc. of the 8th ACM Conf. on Computer and Communication Security*, pages 166–175. ACM , 2001.

[14] UML Resource Page. Unified Modeling Language. http://www.uml.org.

[15] A. W. Roscoe. Modelling and Verifying Key-Exchange Protocols using CSP and FDR. In *Proc. of The 8th Computer Security Foundations Workshop (CSFW 95), Kenmare, Ireland, Mar. 1995*, pages 98–107. IEEE Computer Society, 1995

[16] S. Schneider. Verifying Authentication Protocols in CSP. *IEEE Trans. Sofware Engineering*, 24(8):743–758, 1998. IEEE Computer Society .

[17] J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. of the 19th IEEE Computer Society Symposium on Research in Security and Privacy (SSP 98), Oakland, CA, USA, May 1998*, pages 160–171. IEEE Computer Society, 1998.