# Application of Formal Methods to the Analysis of Web Services Security*

Llanos Tobarra, Diego Cazorla, Fernando Cuartero, and Gregorio Díaz

Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha. 02071 Albacete, Spain.
{mtobarra, dcazorla, fernando, gregorio}@info-ab.uclm.es

**Abstract.** Web Services technologies have introduced a new challenge for security protocols. Traditional security protocols cannot handle intermediaries and the flexibility of Web Services bindings. Thus, several proposals for introducing security in Web Services have been presented. One of these is *Web Services Security*. In this paper we illustrate how this protocol works, with an example, and analyse whether it is a good option guaranteeing the security of Web Services.

**Keywords:** Protocols and standards for WS, Security of WS, Secure Electronic Commerce.

## 1 Introduction

The rapid development of the World Wide Web in recent years has dramatically increased the exchange of information between clients and companies, and has also boosted electronic commerce transactions. Traditionally, the environment where electronic transactions occur consists of a web server that offers 'services' to human clients who use a web browser to select the information or products they wish to obtain. Nowadays this view is changing; companies wish to offer and use services automatically, i.e., they want to 'live' in a world where interoperability between various software applications running on separate platforms is possible, and, for example, Java can talk with Perl, and Windows applications can talk to UNIX applications.

A technology that has emerged recently and offers these kinds of transactions is Web Services [27]. Web Services implements a new Service Oriented Architecture (SOA), which is based on loosely coupled services. In a Web Services environment we find the following components:

- *Service providers*: they implement the web service and, in most cases, publish the service interface and the service registry information.
- *Service brokers*: they allow clients to access the service interface and the implementation information.

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

– *Service clients*: they look for a service in a broker registry and then connect to the service provider in order to use it.

One of the most important issues in Web Services development is that each functional block should be platform or programming language-independent, and accessible for everybody. Thus, each block has to be described using an internet standard. The most important internet standards related to web services are the following:

– *XML* (eXtensible Markup Language)[26] is a markup language which underlies most of the specifications related to Web Services. XML is actually a metalanguage (a language for describing other languages) which lets you design your own customised markup languages for unlimited different types of documents
– *SOAP* (Simple Object Access Protocol)[22] is an XML-based messaging protocol used to encode the information in Web Services request and response messages before sending them over a network. SOAP messages are independent of any operating system or protocol and may be transported using a variety of Internet protocols, including SMTP, MIME and HTTP.
– *WSDL* (Web Services Description Language)[23] is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. Service providers use a WSDL document to specify available services. It also contains service access information.
– *UDDI* (Universal Description, Discovery and Integration)[19] is the client-side API and a server implementation based on SOAP. It stores and retrieves information about service providers and web services.

In order to use a web service, a client obtains a WSDL file in which a particular web service is described. If the client knows where the web service is located, they can retrieve it directly. Otherwise they can search for it using the UDDI protocol. Then, they prepare a SOAP request, which is an XML document that follows an XML schema. Each SOAP message is composed of a main element called an *envelope*. Each envelope has two main parts: payload data, included into the body, and one or more headers that contain control data such as addressing data, security items or quality options. The client sends the request through a transport protocol, usually HTTP. When a web service receives a request, it executes the requested actions and responds to the client with a SOAP response message. This message includes the result of the actions.

One of the main problems in using web services is that they are exposed to security attacks. Traditional security protocols, such as SSL [7], TLS [1] and IPsec[13], are used to protect communication between two agents in a network. Nevertheless, these are point-to-point technologies, whereas web services need end-to-end level security because the information does not travel straight to the endpoint; usually information needs to pass through several intermediate agents that may need to use or alter some parts of the information. Moreover, traditional security protocols secure communication at transport rather than message level. Therefore, messages are protected only while in transit on the wire.

In order to introduce security in web services, we can consider (among others) the following protocols: Web Services Security [11], Security Assertions Markup Language (SAML) [21], Identity Federation [18] and Extensible Access Control Markup Language (XACML) [20].

In this paper we will focus on Web Services Security (WS-Security), a protocol that extends SOAP in order to implement *message integrity* and *confidentiality*. While SOAP provides a flexible technique for structuring messages, it does not directly address how to secure these messages. WS-Security builds on the SOAP specification, structuring the use of essential security capabilities. WS-Security uses binary tokens for authentication, digital signatures for integrity and content-level encryption for confidentiality. It is based on XML Encryption [24] and XML Signature [25] and guarantees the integrity and confidentiality of information. Several companies such as IBM, Microsoft and Sun have adapted its development tools for Web Services in order to support this security protocol.

In line with the development of e-commerce and security protocols, some techniques have also been developed to model a system and check its properties on it. One of the most promising techniques of this type is *model checking*. Model checking [5] is a formal methods based technique for verifying finite-state-concurrent systems, and has been implemented in several tools. One of the main advantages of this technique is that it is automatic and allows us to see if a system works properly or not. In case the system does not work as expected, the model checking tool provides a trace that leads to the source of the error.

In this paper we present a formal verification of WS-Security using the Casper/FDR2 toolbox [15,6]. Casper is a compiler that accepts a syntax very similar to the syntax used to specify protocols, and translates a model into CSP [9] code which is verified using the model checker FDR2. Although Casper uses a general purpose model checker, the translation to CSP code is done automatically and transparently to the user, and the results of the verification are presented in terms of the Casper model.

We have modelled and analysed a web service application (license server) previously developed by using Microsoft .NET 2003 and Microsoft Web Services Enhancement (WSE) [16]. We have abstracted the application code in order to build a simple Casper model which deals with the exchange of sensitive (encrypted and signed) information (SOAP messages) between client and server.

Several papers [12,8,17] can be found which use formal methods to analyse Web Services, and in which formal methods are used to analyse the behaviour and the performance of web services standards; these analyses help designers to correct any errors. More related to our work, however, are [14] and [4]. The first maps out of SOAP messages (including WS-Security information) to Casper input is presented. The second one presents a scripting language called *TulaFale* for specifying SOAP security protocols and verifying attacks. TulaFale is based on Pi-Calculus and some extensions for symbolic cryptographic operations. They verified the WS-Security protocol family in [2] with this tool. These same group of researchers have developed an implementation of an automatic tool, described in [3], that translates security policy files into a TulaFale script and verifies some properties by invoking a theorem prover.

The paper is organised as follows. In Section 2 we describe our web service application (software license server) and the SOAP messages that client and server exchange. In Section 3 we build the Casper specification that models the system, describe the security requirements that the system should achieve, and verify the model against these requirements. Finally, in Section 4 we give our conclusions and some outlines for future work.

## 2 A Web Services example: a software license server

In this section we illustrate how WS-Security works, through an example. A software company offers on its web site a set of applications for its clients. When a client wants to use one of these programs they register through a web service and pay for several licenses through a payment service. Then, they can download the programmes and get several product keys using another web service (the license server). If they do not register they will not be able to install the application because during the installation process they will be asked for the product key. A general overview of the system is depicted in fig. 1.

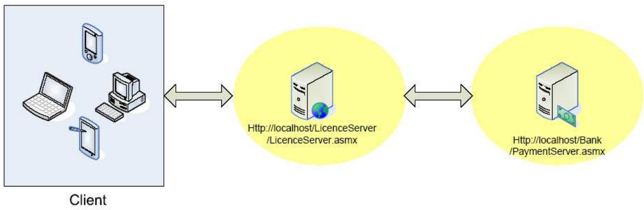


**Fig. 1.** General overview of the system

We shall look at two web services:

1. A web service for registering clients, where a client gives their financial data, a set of selected applications and a number of licenses for each application.
2. A web service for obtaining product keys. This web service generates a new product key if the client is registered. The web service accepts as parameters a username, password and product identifier.

First, a client who wants to use several software products must be registered in the system. In order to register, the client connects to the register web service and send their data. Then, the license server processes the information and generates an invoice that is sent to the payment service that carries out the purchase order. When the payment service responds to the server with a payment authorisation code, the server sends this back to the client together with information on the result of the transaction.

Afterwards, a registered client will need a product key in order to install the application. So, the client sends a request to the license server. The server checks if the client is allowed to use that program and the status of the client subscription. If everything is right, the server generates a new product key for the request and sends it to the client.

Although we have described two web services, in the rest of the paper we will focus on the second, i.e., the license server.

A registered client calls on the web service, which allows it to get a new product key for an application (see fig. 2). The client gets the keys through a small program that asks for the username, the corresponding password and an application code. This program generates a request, which is a SOAP message that follows the specification of the WSDL protocol. If the client has been registered and has the corresponding permissions, the server sends a SOAP message response that includes the newly generated product key. An overview of the system and the two exchanged SOAP messages is depicted in fig. 2.

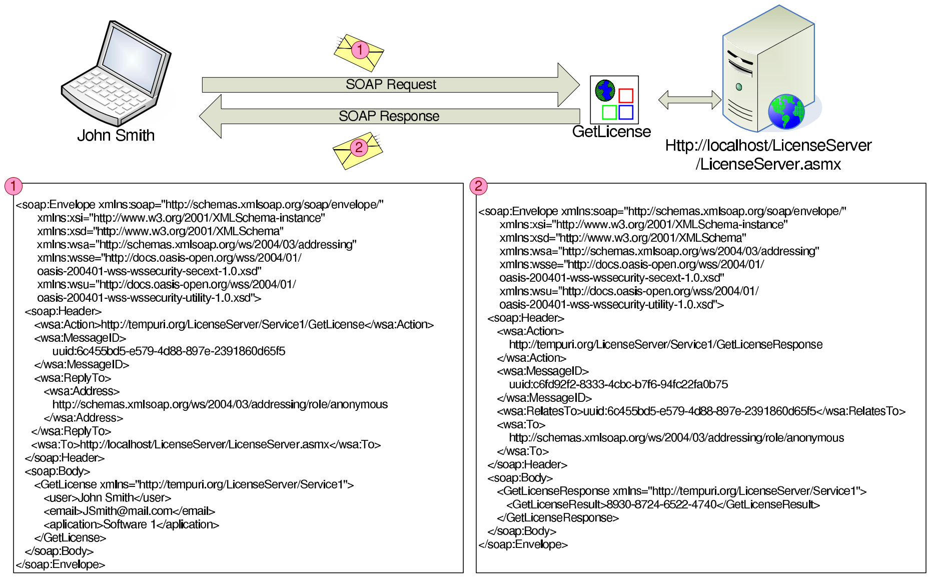


**Fig. 2.** SOAP messages

It is clear that this web service needs to guarantee several security properties, such as the confidentiality of the client authorisation data items. First, a registered client proves that they can be trusted, with their username and password. If an intruder guesses these variables they will be able to access the product keys. We must be sure that the username and passwords are secrets shared between server and client. WS-Security also allows us to cipher data to guarantee its confidentiality, thus, we must encrypt the request message body.

We must cipher the response body too. In it, the server sends the product key. If an intruder can guess it, they will be able to install the program without paying for it. It is the client who pays instead of the intruder in this case. The server must guarantee the confidentiality of the product key, which must be a secret shared with the client.

When the server generates a product key, this key is valid for only one kind of application. If the client asks for a key to a program and tries to install a different one, they will not be able to use the received key in the installation. An intruder can alter the request message and change the selected application code, meaning the client receives an incorrect key and that, therefore, they cannot install the program. Thus, the integrity of the message must be checked in order to avoid these kinds of attacks. WS-Security offers the possibility of signing a message; this way, both agents, server and client, can verify the integrity of the signed parts.

WS-Security adds a new header element called *Security* to each SOAP message (see fig. 3). In it, we include the necessary security tokens for the message. The first XML element is usually a BinarySecurityToken, which represents a binary token as a X.509 certificate, or a Kerberos ticket. It is used to sign message parts and to cipher symmetric keys. In this case we only sign the message with it. The request message will contain the client certificate, and the response message the server certificate.

WS-Security allows clients to first sign and then cipher messages or vice versa. The order of these two operations is determined by the order of XML Signature elements and XML Encryption elements in the Security header. We will consider the case of when a message is ciphered and then signed, i.e., when XML Encryption elements appear after the Signature element, as represented in fig. 3. If the message were ciphered *after* being signed the order of the elements would be different.

The Signature element contains a digital signature computed from several parts of the message. It is divided into two main parts: information about how the signature was computed, represented by the SignedInfo element, and the signature value in the SignatureValue element. The first part deals with the canonicalisation method and the signature algorithm. It includes a list of references in the References element. For each signed part in the message there is a Reference element that points to a signed part of the message, that in turn informs us of the digest algorithm and the digest result. There is also a KeyInfo element that refers to the certificate used in the signature.

The EncryptedKey element represents a symmetric key that is ciphered with the other public agent key. The EncryptionMethod element indicates which algorithm is used to cipher the key. Then, there is a KeyInfo element that makes a reference to the certificate used to secure the key. If the certificate is unknown to the receiver, the producer must include it in a BinarySecurityToken. After the KeyInfo item there is a CipherValue element which includes the ciphered value of

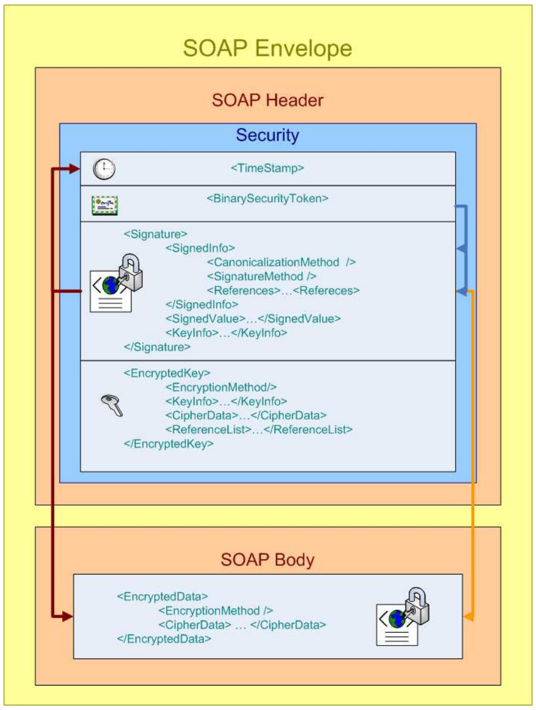


**Fig. 3.** Abstract structure of a SOAP Message with WS-Security

the key. If the shared key has been exchanged before, it will contain a reference to that key instead of its ciphered value. In the last subelement of EncryptedKey, the ReferenceList element, is a list of references in which each reference points to a ciphered message part. When a message part is encrypted, it is replaced by a EncryptedData, where there is a reference to the key, and the result of the cipher process.

If we look at encryption *before* signature, the protocol works as follows:

- First, the client creates a SOAP request message without any security measures.
- Then, they include a security header and generate a symmetric key that ciphers the message with the server public key. The key is added to the message.
- The client ciphers their username, their password and the selected application code with the newly generated symmetric key. The body element is replaced by an EncryptedData element as a result of this step.
- The client computes a digital signature from the ciphered body of the message. The client adds all the security elements corresponding to the digital signature and sends the message to the server.
- The server receives the message and verifies the digital signature, it then decrypts the message body and checks the client data with the function

*registered*. This function returns *true* if the username and password are correct. Otherwise it returns *false*.
- If the client is registered, the server will generate a product key.
- Finally, the server prepares the SOAP response message following the same steps as the client and sends it to the client.

## 3 Verification of the Web Service

In this section we will analyse, using the model checking tool Casper, whether the web service described in the previous section allows a registered client to connect to a server and obtain a valid key for a software product. In order to achieve this client aim, WS-Security must guarantee the following results:

- Confidentiality of username and password in the request message, and the product key or code element in the response message.
- Integrity of the full SOAP envelope.

In our model, we will consider an intruder who can perform the following actions:

- Overhear and intercept all the messages over the network.
- Modify the messages. The intruder can 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 in the system.

We will assume that the intruder cannot perform any cryptanalysis.

We use Casper syntax to represent a SOAP message (see fig. 4). First, we suppose that all digest functions, encryption methods and signature algorithms are secure. In WS-Security there is only one hash function, SHA-1, which is represented by the hash type function `sha` in our protocol description. Computation of the digest of several variables is represented as `sha(v1,...,vn)`. If we wish to represent a message `m` that is ciphered with the key `K`, we do so with the expression $\{\texttt{m}\}_{\texttt{K}}$.

We will suppose that the clients and the web service server know each other's certificates. We represent certificates by means of two functions: *PK(A)*, which returns the public key of the agent A, and *SK(A)*, which returns the private key of the agent A. This allows us to represent binary tokens as X.509 certificates and Kerberos tickets.

We described WS-Security messages in the previous section. Firstly, there is a certificate that, as we have just explained is replaced by two functions. Then, there is an EncryptedKey element, a symmetric key ciphered with the receiver public key, represented by $\{\texttt{Key}\}_{\texttt{PK(Receiver)}}$. Only the owner of the private key partnering the public one can decipher it and find out the symmetric key. After the symmetric key there is a digital signature. A digital signature is a digest value ciphered with the signer's private key. The sender signs the message body after it has been encrypted, so we have $\{\texttt{sha}(\{\texttt{Body}\}_{\texttt{Key}})\}_{\texttt{SK(Agent)}}$.

In the request message we represent the body with three important variables: username, password and application code. In the response message we represent the body with the variable *code* which represents the product key. The body of each message is ciphered with the corresponding symmetric key, included in the EncryptedKey.

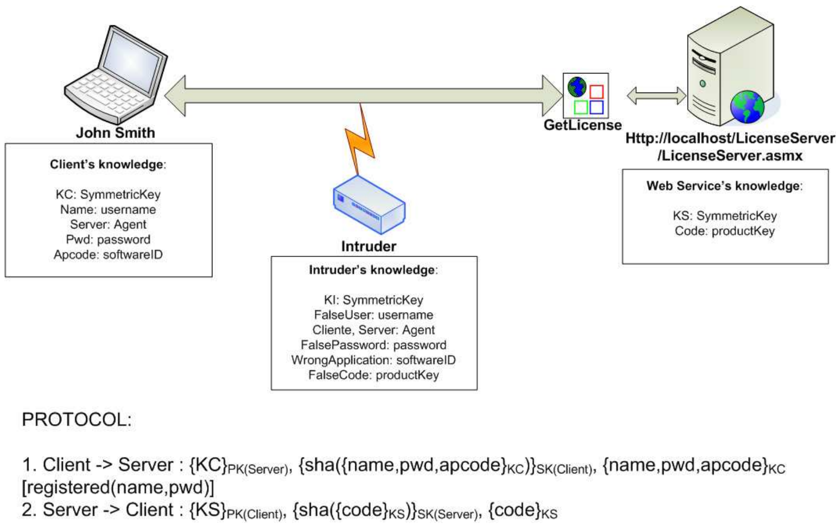


**Fig. 4.** Representation of the analysed system and the resulting protocol

We now check the following properties:

1. Aliveness(Client,Server) and Aliveness(Server,Client). Both properties check that the client executes the protocol with the server, and vice versa. So, the intruder cannot take the place of the client nor the server.
2. StrongSecret(Client,name,[server]) and StrongSecret(Client, pwd,[server]). These search a trace where the intruder can guess the value of the username *name* and the password *pwd*.
3. Secret(Client, KC, [Server]). This verifies that the client only shares a symmetric key, KC, with the server and that the intruder cannot guess it. So, if the intruder does not know it, they will not be able to decrypt the message body. The difference between Secret and StrongSecret is that the latter checks all traces, even though the protocol has not been executed completely.
4. Agreement(Client,Server,[KC, name, pwd, apcode]). This verifies that the client executes a complete run of the protocol with the server and that both share the same value for the variables username, password and application code.
5. StrongSecret(Server, code, [Client]). This confirms that the protocol guarantees the confidentiality of the variable code, which represents the product key.

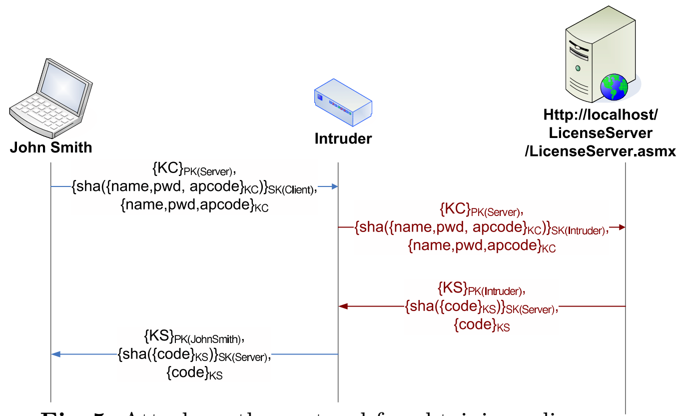


**Fig. 5.** Attack on the protocol for obtaining a license

6. Secret(Server, KS, [Client]). This checks that the server only shares a symmetric key, KS, with the client and that the intruder cannot guess it. So, if the intruder does not know it, they will not be able to decrypt the message body.
7. Agreement(Server,Client,[KS,name,pwd,apcode,KC,code]). This checks that the server executes a complete run of the protocol with the client, and that they share the same value for the following variables: KS, name, pwd, apcode, KC and code. Thus, it verifies the integrity of the two messages.

We check all these properties with Casper and our system specification. We look at two different situations. In the first, a client only wants a product key for a single program and runs the protocol once. In the second, the client runs the protocol twice because they want to install two different software programs, *sw1* and *sw2*.

In the first case, Casper finds an attack (see fig. 5). The intruder knows the client public key, so can find out the value of $\mathsf{sha}(\{\mathsf{name,pwd,apcode}\}_{\mathsf{KC}})$. From this information, the intruder can generate a correct request message and is then able to take the place of the client. The intruder executes the protocol, with the server as an intermediary. The problem is that the username and the password are not associated with the agent identifier. Therefore we must include the agent identifier in the message. A possible option is to use WS-Addressing, which allows us to include *sender identifier* and *receiver identifier* amongst other information. We include these identifiers and we sign them. The resulting protocol is

1. Client $\rightarrow$ Server: $\{\text{KC}\}_{\text{PK(Server)}}$,
   $\{\text{sha}(\{\text{name, pwd, apcode}\}_{\text{KC}}, \mathit{Server}, \mathit{Client})\}_{\text{SK(Client)}}$,
   $\{\text{name, pwd, apcode}\}_{\text{KC}}$
   **[registered(name,pwd)]**
2. Server$\rightarrow$ Client: $\{\text{KS}\}_{\text{PK(Client)}}$,
   $\{\text{sha}(\{\text{code}\}_{\text{KC}}, \mathit{Server}, \mathit{Client})\}_{\text{SK(Server)}}$,
   $\{\text{code}\}_{\text{KS}}$

We check all the properties again with this version; Casper does not find any attack. Then, we check what happens if a client gets a product key and then tries to get another. If we have a variable $v$ that is used in both executions of the protocol, we will use $v_i$ to denote the value of $v$ in every run of the protocol. For example, $KC_1$ is the client symmetric key for the first execution of the protocol, and $KC_2$ is the same value but for the second. We suppose these variables are refreshed in each execution.

We find two new attacks as result of our verification of the two-executions version of the protocol. As we can see in figures 6 and 7, the intruder behaves as a intermediate and records all the messages. After one execution, they can replace one of the honest agents and cheat the other. In the first attack, the intruder takes the place of the client and gets a product key. A client can request a fixed number of product keys, thus, the intruder reduces the number of requests by the client and so the client loses money. In the second attack, the client is deceived by the intruder and gets a wrong product key for the selected application. Both attacks share the feature that the intruder uses previously recorded messages (known as *replay attack*).

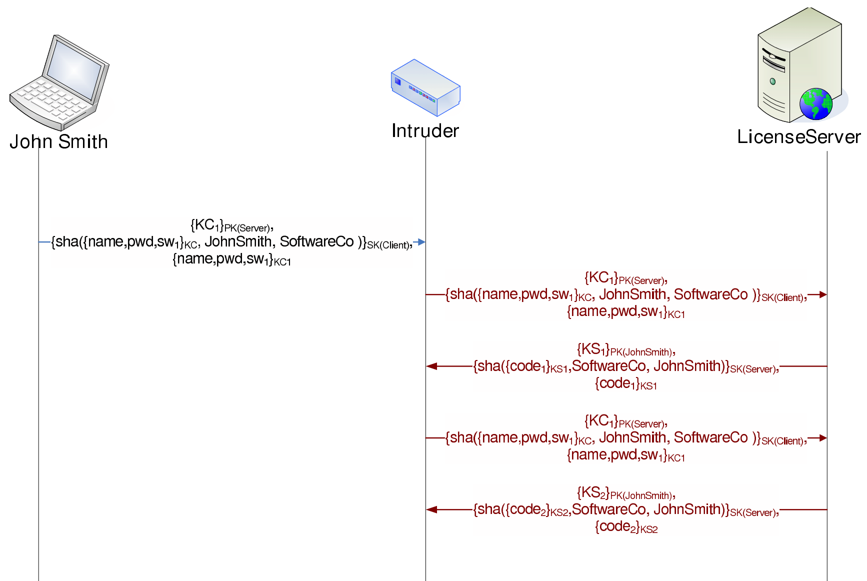


**Fig. 6.** First attack on the protocol for obtaining a license with two executions

To avoid replay attacks, as in the both cases explained above, we need to introduce some information to each message to make it unique, e.g., a time-stamp or a message identifier.

WS-Security allows us to include time-stamps in a message. A timestamp consist of a creation time and an expiry time. Clients should select a short time validity period, because otherwise, an intruder could use the message in a replay

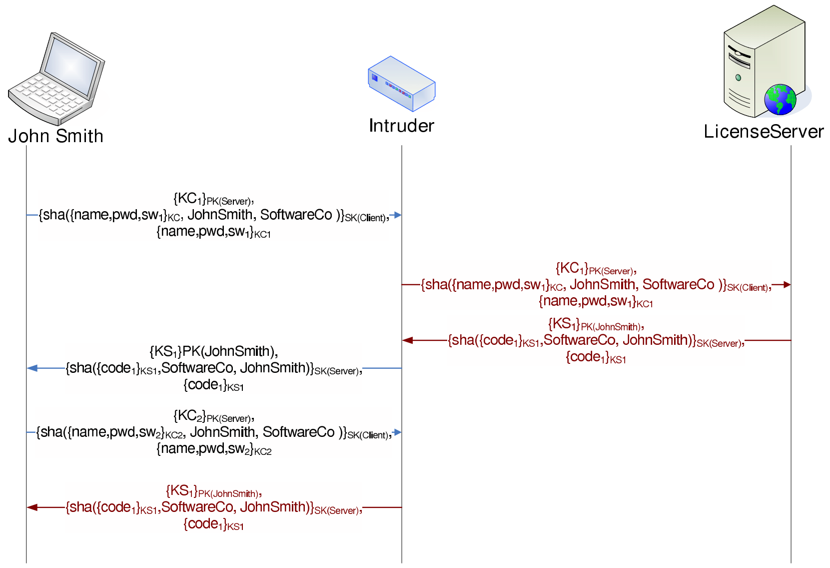


**Fig. 7.** Second attack on the protocol for obtaining a license with two executions

attack before it expires. WS-Security time-stamps may be modelled in Casper by using a special kind of variable. With this variable we can represent the creation time but not the expiry time; thus, we have to include a condition in the time-stamp which checks how many time units have passed since the message was sent. Taking time-stamps into account, the protocol specification is as follows:

1. Client $\rightarrow$ Server:
   Clock,$\{\text{KC}\}_{\text{PK(Server)}}$,
   $\{\text{sha}(\{\text{name, pwd, apcode }\}_{\text{KC}}, Server, Client, Clock)\}_{\text{SK(Client)}}$,
   $\{\text{name, pwd, apcode}\}_{\text{KC}}$
   **[registered(name,pwd) and (Clock==now or Clock==now-1)]**
2. Server $\rightarrow$ Client:
   Clock,$\{\text{KS}\}_{\text{PK(Client)}}$, $\{\text{sha}(\{\text{code}\}_{\text{KC}}, Server, Client, Clock)\}_{\text{SK(Server)}}$,
   $\{\text{code}\}_{\text{KS}}$
   **[Clock==now or Clock==now-1]**

Casper time-stamps and protocol time-stamps nevertheless have a different semantics. While Casper only gives consideration to local clocks, i.e., each agent (process) has an internal clock, represented by an integer, which starts from zero when the agent sends the first message, WS-Security time-stamps are obtained from global clock, i.e., each time-stamp is a date includes the day, month, year, hour, minutes and seconds. Thus, in Casper it is possible for two different messages to have the same time-stamp because they were sent by different agents, while in WS-Security it is very difficult for two messages to have the same time-stamp.

Although we have found an attack on the Casper model including time-stamps, we cannot conclude that WS-Security suffers from this attack, because in fact, the model is not representing the protocol.

The second solution considered above to avoid replay attacks was to introduce a unique message identifier. In this case, the server can store every message identifier and check whether a new message is actually new or is being replayed. The message identifier should be signed in order to guarantee its integrity. The protocol model could be as follows:

1. Client → Server:

   MessageID1,$\{\text{KC}\}_{\text{PK(Server)}}$,

   $\{\text{sha}(\{\text{name, pwd, apcode }\}_{\text{KC}}, Server, Client, MessageID1)\}_{\text{SK(Client)}}$,

   $\{\text{name, pwd, apcode}\}_{\text{KC}}$

   **[registered(name,pwd)]**

2. Server → Client:

   MessageID2,$\{\text{KS}\}_{\text{PK(Client)}}$,

   $\{\text{sha}(\{\text{code}\}_{\text{KC}}, Server, Client, MessageID2)\}_{\text{SK(Server)}}$, $\{\text{code}\}_{\text{KS}}$

Despite this, in Casper we cannot represent the server cache of message identifiers, so we cannot check if this version guarantees all the security properties.

## 4 Conclusions and Future Work

In this paper we have presented a web service application (a software license server) that studies security issues by means of WS-Security. In order to analyse whether this web service is secure, we have built the SOAP messages exchanged by the client and the server, and then translated these messages into Casper syntax in order to verify some security properties. We have looked at two scenarios: in the first, a client wishes to obtain a single product key, whereas in the second the client wishes to obtain two different product keys.

Taking into account the results of the verification, we can put forward the following conclusions:

– Even in the most simple case, an intruder could supplant the client and obtain a product key. This means that, besides WS-Security another mechanism is needed which allows agents to identify themselves. One possibility is to use WS-Addressing which allows us to insert in a SOAP message information about the participants.
– Even if we use WS-Addressing, it is possible that an intruder could obtain a product key if we consider a more sophisticated environment (the two run version). In this case, the problem is which the intruder can record messages that can be used to perform a replay attack.

The problem in the second case is that a message can be stored and replayed as many times as the intruder needs. As mentioned before, a unique message identifier, or a time-stamp with a short validity period, could be a solution to this attack. Unfortunately, Casper failed to model SOAP messages that include time-stamps and/or messages identifiers, so we cannot say whether this model is secure or not. We could use another feature of WS-Security in order to avoid replay attacks. We could add a *UsernameToken* element to a SOAP message signed by the client. This element contains a username, the corresponding password and a nonce (a fresh random value), which guarantees the uniqueness of the message. The username and password are secrets shared between the client and the server. Thus, we should only have to add the nonce to our representation of the protocol.

Another solution could be to identify each message as part of a flow or a conversation. Some protocols, such as WS-ReliableMessaging, include in each message a unique identifier for each sequence of messages. It also includes the number of the message in the sequence. Each time a client starts a new sequence the identifier is renewed. When an agent wants to establish a sequence it sends a request for a new sequence identifier to the other end. Then it can send its message. The other end should send an acknowledgement of the received message. When an agent does not want to send more messages, it must send a finished sequence message. All these control messages are an overhead for a simple protocol, as with the one we have analysed.

Our future work is concerned with building a model of the system which allows us to prove that WS-Security (working together with other protocols such as WS-Addressing and/or WS-ReliableMessaging) is a good option in guaranteeing the security of Web Services (including replay attacks).

Finally, it is worth pointing out that WS-Security does not provide a full security solution by itself. It should be used with other protocols. In a request/response conversation a client and a server do not negotiate any security options. So it is recommended that developers use WS-Policy to establish the security requirements of clients and servers. Currently, a set of complementary protocols is being developed to complement the security level of WS-Security. These include WS-Trust and WS-Secure Language [10].

## References


1. T. Dierks; C. Allen. *"The TLS Protocol. Version 1.0 ". RFC 2246.* Standards track, Network Working Group, January 1999.
2. K. Bhargavan, C. Fournet, A.D. Gordon, and R.Corin. *"Secure Sessions for Web Services"*. August 2004. At `http://research.microsoft.com/projects/samoa/secure-sessions-with-scripts.pdf`.
3. K. Bhargavan, C. Fournet, and A.D. Gordon. Verifying policy-based security for web services. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 268–277, New York, NY, USA, 2004. ACM Press.
4. K. Bhargavan, C. Fournet, and A.D. Gordon, and R. Pucella. Tulafale: A security tool for web services. In *Formal Methods for Components and Objects: Second International Symposium, FMCO 2003*, volume 3188 of *Lecture Notes in Computer Science*, pages 197 – 222. Springer, November 2003.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

6. Formal Systems (Europe) Limited. FDR Manual. http://www.fsel.com/fdr2_manual.html.
7. O. Alan Freier, Philip Karlton, and Paul C. Kocher. *"The SSL Protocol Version 3.0"*. Internet draft, Netscape, March 1996.
8. G.Díaz, J. Pardo, E. Cambronero, V. Valero, and F. Cuartero. *"Verification of Web Services with Timed Automata". 1st Int'l Workshop on Automated Specification and Verification of Web Sites*, 2005.
9. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
10. IBM and Microsoft. *"Security in a Web Services World: a proposed architecture and roadmap".* *http://www.-106.ibm.com/developerworks/library/ws-secmap/*, April 2002.
11. IBM, Microsoft, and VeriSign. *"Web Services Security (WS-Security). Version 1.0".* April 2002.
12. J.E. Johnson, D.E. Langworthy, L. Lamport, and F.H.Vogt. *"Formal Specification of a Web Services Protocol". Electronic Notes in Theoretical Computer Science 105 (2004) 147-158*, February 2004.
13. S. Kent and K. Seo. Security architecture for the internet protocol. Internet Draft, october 2004.
14. E. Kleiner and A.W.Roscoe. *"Web Services Security: a preliminary study using Casper and FDR". Proceedings of the Workshop on Automated Reasoning for Security Protocol Analysis (ARSPA 2004)*, June 2004.
15. G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6:53–84, 1998.
16. Microsoft. Microsoft Web Services Enhacements (WSE) 2.0. http://http://msdn.microsoft.com/webservices/building/wse/.
17. S. Nakijima. *"On verifying Web Services Flows". Proc. SAINT 2002 Workshop*, pages 223–224, January 2002.
18. OASIS. *"Identity Federation. Liberty Alliance Project "*, 2004.
19. OASIS. *"UDDI Version 3.0.2".* *http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3*, October 2004.
20. OASIS. *"eXtensible Access Control Markup Language (XACML) v2.0"*, February 2005.
21. OASIS. *"Security Assertion Markup Language (SAML) v2.0 "*, April 2005.
22. W3C. *"Simple Object Access Protocol (SOAP) 1.1".* *http://www.w3c.org/TR/2000/NOTE-SOAP-20000508*, May 2000.
23. W3C. *"Web Services Description Language (WSDL) 1.1 ".* *http://www.w3.org/TR/wsdl*, March 2001.
24. W3C. *"XML Encryption Syntax and Processing".* *http://www.w3c.org/TR/2002/REC-xmldend-core-20021210*, December 2002.
25. W3C. *"XML Signature Syntax and Processing".* *http://www.w3c.org/TR/2002/REC-xmldsig-core-20020212*, February 2002.
26. W3C. *"Extensible Markup Language (XML)1.1".* *http://www.w3.org/TR/2004/REC-xml11-20040204/*, April 2004.
27. W3C. *"Web Services Architecture".* *http://www.w3c.org/TR/2004/NOTE-ws-arch-20040211*, February 2004.