On secure implementation of an IHE XUA-based protocol for authenticating healthcare professionals [⋆]

Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi

Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
masi@math.unifi.it, {pugliese,tiezzi}@dsi.unifi.it

Abstract. The importance of the Electronic Health Record (EHR) has been addressed in recent years by governments and institutions. Many large scale projects have been funded with the aim to allow healthcare professionals to consult patients data. Properties such as confidentiality, authentication and authorization are the key for the success for these projects. The Integrating the Healthcare Enterprise (IHE) initiative promotes the coordinated use of established standards for authenticated and secure EHR exchanges among clinics and hospitals. In particular, the IHE integration profile named XUA permits to attest user identities by relying on SAML assertions, i.e. XML documents containing authentication statements. In this paper, we provide a formal model for the secure issuance of such an assertion. We first specify the scenario using the process calculus COWS and then analyse it using the model checker CMC. Our analysis reveals a potential flaw in the XUA profile when using a SAML assertion in an unprotected network. We then suggest a solution for this flaw, and model check and implement this solution to show that it is secure and feasible.

1 Introduction

In recent years, the exchange of Electronic Health Records (EHRs) among clinics and hospitals has become an interesting field of research and study for academia and the industry. An EHR is a set of sensitive data containing all healthcare history of a patient (e.g. medical exams or prescriptions).

Two important concepts in EHR management are security and interoperability: the content of an EHR cannot be disclosed to unauthorized people without an explicit patient consent and has to be accessible by heterogeneous systems. These requirements impose that any software participating in an EHR exchange must adhere to common specifications.

Integrating the Healthcare Enterprise (IHE) [1] is a worldwide initiative founded for promoting the coordinated use of established standards to improve information sharing in an healthcare scenario. To achieve security and interoperability, many profiles for integrating different systems have been proposed by IHE. These profiles can be combined for building healthcare applications by using a Service Oriented Computing (SOC) approach and OASIS standards such as SAML [2], ebXML [3], and WS-Trust [4].

[⋆] This work has been supported by the EU project Sensoria, IST-2005-016004.

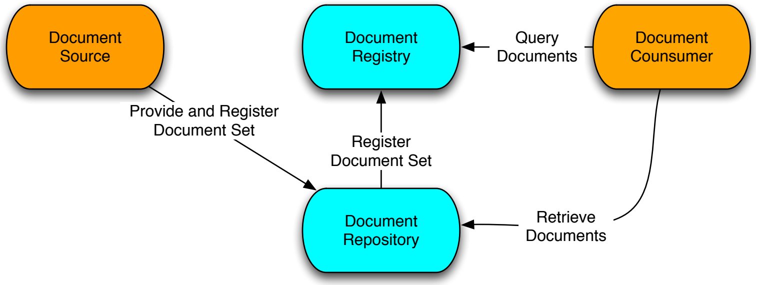

Fig. 1. The XDS model

IHE specifications are by now used to build nationwide projects with the aim of sharing patient healthcare data, such as the French GIP-DMP [5] or the Austrian ARGE-ELGA [6] EHR projects.

A typical EHR transmission is made by exploiting an ebXML registry/repository model (called in IHE jargon Cross Enterprise Document Sharing, XDS), as depicted in Figure 1. A document source (typically a medical device) provides and registers documents for a given patient to a repository that extrapolates metadata and feeds a registry. A document consumer (a workstation used by an healthcare professional) queries the registry for documents related to the patient. The registry searches in its metadata and replies with a set of links. These links are used by the consumer for retrieving documents from the repository.

Confidentiality and auditing is achieved using Transport Layer Security (TLS) [7] and logging as defined in the Audit Trail and Node Authentication (ATNA) profile [1]. Any node participating in ATNA owns an host X.509 certificate for attesting machine's identity. Requisites of each profile can be merged (i.e. *grouped*) together for building a complete infrastructure. For instance, XDS grouped with ATNA provides a secure and audited data exchange through TLS channels using a registry/repository model.

Healthcare professionals authentication is one of the basic requirements for the access of person related health data at regional, national and also multinational level. Authentication is defined by IHE in the Cross Enterprise User Assertion (XUA) integration profile. The XUA specification covers the use of a SAML authentication assertion issued by an identity provider to be injected using WS-Security [8] during the documents queries. Due to local government complexities where each nation / hospital / clinic have its own authentication method, the assertion issuance process is leaved open. The WS-Trust standard is only suggested, but not proposing a specific profile or a set of messages to be exchanged potentially leads to weak implementations.

Because of the impact that the IHE specifications are having, formal models of protocols and standards are needed. A large body of work has been already made on analyzing WS-Trust protocols, see e.g. [9–12], where message-level authentication [13] properties are verified. By relying on them, in this paper we aim at formalizing and implementing a protocol combining WS-Trust and IHE profiles. More specifically, our protocol is built on an XDS transaction grouped with ATNA and authenticated by an XUA SAML assertion. To our best knowledge, this is the first tentative to formalize protocols derived from IHE specifications.

```
<saml:Assertion><saml:Issuer> issuer-identity </saml:Issuer>
    <ds:Signature> ... </ds:Signature>
    <saml:Subject><saml:NameID> username </saml:NameID>
        <saml:SubjectConfirmation Method="#bearer">
            <saml:SubjectConfirmationData> ...</saml:SubjectConfirmationData>
        </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Conditions NotBefore="ts1" NotOnOrAfter="ts2">
        <saml:AudienceRestriction><saml:Audience> registry-address </saml:Audience>
        </saml:AudienceRestriction>
    </saml:Conditions>
    <saml:AuthnStatement  AuthnInstant="ts3"> ...
    </saml:AuthnStatement>
    <saml:AttributeStatement> ... </saml:AttributeStatement>
</saml:Assertion>
```

Fig. 2. Excerpt of a sample SAML token (using the *bearer* method)

The process for issuing a SAML token is a delicate task: if an assertion is stolen, a malicious attacker can re-use it and have access to unauthorized healthcare data. One could suggest to use TLS for authenticating channels during the issuance. In fact, IHE supports TLS by means of ATNA for compatibility with legacy non-WS standards such as Dicom [14] and Health Level 7 version 2 [15]. However, given the possibility by XUA to choose any issuance process, the use of TLS should be discouraged in favor of WS-Security. Moreover, as argued in [11], if a secure transport layer in web service communications is used, intermediaries cannot manipulate the messages on their way; this does not comply with the requirements of SOC. For these reasons, our proposal does not rely on TLS.

It is worth noticing that in the IHE security model, applications should also avoid heavy use of encryption, because the impact on performance of the current encryption algorithms is excessive [1]. Indeed, IHE applications can even run on medical devices with a reduced computational power.

The work presented in this paper consists of three main contributions. First, we fill the gap leaved open by XUA by proposing a protocol (Section 2) for issuing the SAML token according to the IHE and OASIS dictates. Second, we formally specify the protocol (Section 3) using the calculus COWS [16]. We then analyze (Section 4) the formal model with the model checker CMC and show that a potentially severe security flaw exists in the SAML assertion format specified by XUA. Third, we provide an implementation of the protocol with our revised assertion format (the implementation is only sketched in Section 5, the interested reader is referred to [17]). We conclude by touching upon comparisons with related work and directions for future work (Section 5).

2 An XUA-Based protocol

As previously discussed, XUA does not address the authentication mechanisms of the local network. Instead, it leverages on the abstraction layer introduced by SAML. The SAML OASIS standard is a set of specification documents defining *assertions* (or tokens) and a *protocol* to exchange them. A SAML authentication assertion is an XML document issued by a *Security Token Service (STS)*[1] that contains statements about

[1] For the sake of simplicity, we assume an STS that is directly able to authenticate users, i.e. it plays also the role of the identity provider.

$C \rightarrow STS$	:	$C, \mathit{msgId1}, STS, \mathrm{UT}(\mathit{user}, \mathit{salt}, \mathit{int}), \mathit{ts1}, \mathrm{RST}(REG)$	(1)
$STS \rightarrow C$	:	$STS, C, \mathit{msgId2}, \mathit{msgId1}, \mathrm{RSTR}(\mathit{ctx}, \{STS, n, ts, \mathit{ctx}\}_{dKey})$	(2)
$C \rightarrow STS$	:	$\mathit{msgId3}, \mathit{msgId2}, STS, \mathit{ts2}, \mathrm{RSTR}(\mathit{ctx}, \{n+1, C, \mathit{msgId3}, \mathit{msgId2}, \mathit{ctx}\}_{K^{+}_{STS}})$	(3)
$STS \rightarrow C$	:	$C, STS, \mathit{msgId4}, \mathit{msgId3}, \mathrm{RSTRC}(\mathrm{RSTR}(\{[STS, ts', \mathit{user}, REG]\}_{K^{-}_{STS}}))$	(4)
$C \rightarrow REG$	:	$C, REG, \mathit{msgId5}, \{[STS, ts', \mathit{user}, REG]\}_{K^{-}_{STS}}$, '*Susan*'	(5)
$REG \rightarrow C$	:	$REG, C, \mathit{msgId6}, \mathit{msgId5}, \mathit{docLinks}$	(6)

Table 1. The proposed XUA protocol

an authentication procedure performed by an underlying authentication mechanism (such as Kerberos) for a *subject*. An example is shown in Figure 2. The SAML token is then used by the service requester to interact with the services listed in the `AudienceRestriction` element.

The contacted *service provider* uses the assertion for authenticating the requester by verifying the digital signature of the trusted issuer. SAML subjects can be confirmed with the method listed in the `SubjectConfirmation` element. Here, we are interested in two methods named *bearer* [2] and *holder-of-key (HoK)* [18]. The bearer subject confirmation method tells the service provider that the subject of the assertion is the presenter (i.e. the bearer) of the assertion. In the holder-of-key method, *STS* binds an identity for the subject (or for the requester) as X.509 data. By this means, we set the subject of the assertion as the healthcare professional with confirmation data as the ATNA certificate of the requesting machine. The service provider can compare such data with the X.509 identity carried in the TLS transaction.

By means of the formal investigation presented in Section 4, we discovered a security flaw due to the format of the SAML assertion. XUA explicitly says that the bearer subject confirmation method shall be supported. However, in a large scale network as described before, it is unrealistic to assume that each node is trusted. Compromised nodes may exist and if one is able to obtain a SAML assertion issued for another, authorized node, with the bearer method it can re-use the assertion to gain access to secret resources. In fact, the service provider has no knowledge if the presenter of the assertion was the original requester. With the holder-of-key method, requester identity is bound as subject confirmation data and digitally signed by *STS*. The service provider can now detect if the bearer is the node which the assertion was intended for by checking if the identity set by *STS* matches the one presented in the communication channel by means of ATNA.

In [11], the feeling of the authors is that it looks like impossible to authenticate correctly the request for a security token issue in a two step protocol as it is instead suggested in the WS-Trust specification. Since our aim is to propose a secure and authenticated holder-of-key assertion issuance, we designed a challenge-response WS-Trust protocol in four message exchanges. Our model involves an XDS transaction grouped with ATNA and XUA for retrieving documents for a patient with id *Susan*. The protocol that we propose, written in a notation commonly used for describing security protocols, is shown in Table 1 and is graphically depicted in Figure 3.

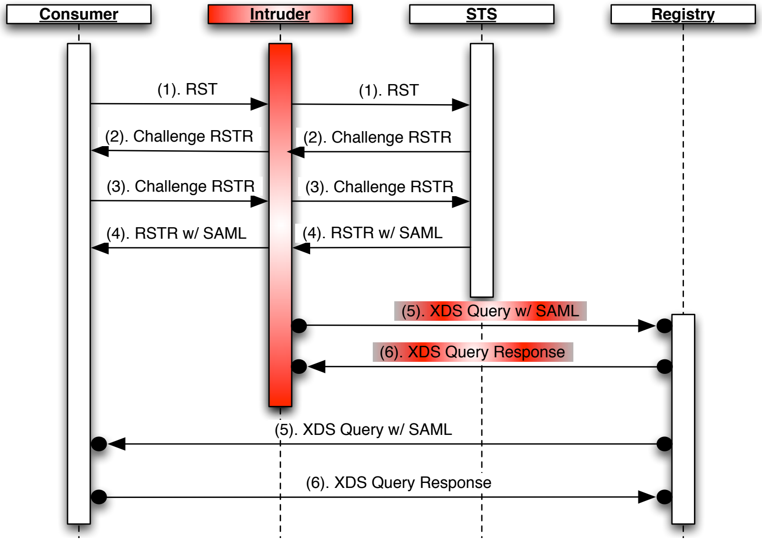

Fig. 3. The WS-Trust protocol for SAML token issuance. Messages (5) and (6) are over TLS channels. An intruder can steal the SAML token in message (4) and, if the subject confirmation method is *bearer*, can perform an unauthorized authenticated query.

Notation $\{M\}_{dKey}$ stands for the symmetric encryption of message M using the derived key *dKey*, $\{M\}_{K^{+}_{STS}}$ for the encryption of M using the public key of *STS* and $\{[M]\}_{K^{-}_{STS}}$ for the signature of M using *STS*'s private key (where [M] is the hash code of M). *ts*, *ts′*, *ts1* and *ts2* are timestamps.

The consumer *C* initiates the protocol by sending the message (1) for requesting a token to *STS*. It sends its identity *C*, a unique message identifier *msgId1*, using WS-Addressing [19], and the identity of the Security Token Service *STS*. Notation UT(*user*, *salt*, *int*) stands for the WS-Security Username Token Profile 1.1 [20] and contains the username, a random number which acts as a cryptographic salt, and an integer, respectively. RST(*REG*) is the WS-Trust 1.3 *Request Security Token* where the registry address *REG* is the ultimate recipient of the token.

Once received the message, *STS* unpacks the value of the username token, unpacks the RST(*REG*) element (*REG* must be in the *STS*'s list of valid assertion targets) and computes the derived key *dKey*. The key is computed by *STS* by concatenating the password of the user (which is given as input by the real human sitting in front of the workstation and is known by *STS* by means of the underlying authentication mechanism) with the salt and then hashed using the SHA-1 algorithm. The result of this operation is also hashed using SHA-1. This process is repeated until the total number of hash operations equals the iteration count *int*. Then, *STS* encrypts the challenge composed by its identity, a nonce *n*, a new timestamp *ts* and the WS-Trust `context` element *ctx* of the challenge (i.e. an identifier defined by WS-Trust used for correlating the messages involved in the token issuance). Indeed, *STS* challenges the requester in order to be sure on its identity and attesting its availability. RSTR is the WS-Trust *Request Security Token Response* element that contains the challenge data.

When message (2) is received by *C*, it computes *dKey* using the same algorithm as the *STS* and decrypts the message (indeed, it is the only participant able to do it). *C* performs the WS-Addressing checks: message (2) must contain the identifier *msgId1* indicating that (2) is in response to (1). It also checks if the request comes from a participants whose identity is included in the RSTR, by means of TLS mutual authentication, for instance. *C* now trusts that the challenge really comes from *STS*. Then, it adds 1 to the nonce and encrypts it, together with the message identifiers and the context, using the *STS* public key. The reply is in message (3).

After receiving the message, *STS* decrypts the content of the RSTR, checks if the nonce is equal to the one that it sent (plus one) and if the context is the same. If it is able to perform all these operations, then it can attest the identity of the user sitting in front of *C*. Thus, it issues the SAML assertion (it is signed by *STS* according to the SAML Signature profile, as enveloped signature) and sends it to *C*, via message (4). The assertion is:

$$\{[STS, ts', user, REG]\}_{K^-_{STS}}$$

where the confirmation method is *bearer*. In fact, if we would have used the *holder-of-key* method, the assertion would be as follows:

$$\{[C, STS, ts', user, REG]\}_{K^-_{STS}}$$

The assertion then contains the requester's identity as ATNA X.509 certificate, here simply represented by *C*, the issuer identity, a timestamp, the user name and the audience restriction list. We omit for simplicity all the details introduced by the SAML specification (e.g. the assertion time range validity).

Once *C* has obtained a security token, it can finally query the registry *REG* to retrieve the links to the repositories containing the EHR data that it is looking for. The query is message (5), which contains the SAML assertion.

Finally, once received message (5), *REG* validates the token. Using the *STS*'s public key it verifies the signature and, if it is valid, delivers the requested resource (i.e. the links *docLinks*) to *C* via message (6).

3 COWS specification of the protocol

In this section, first we report the syntax and the informal semantics of COWS[2], then we present the COWS specification of the XUA protocol in Section 2. Our specification reflects many real-world implementation details. Algorithms, field names and message flows are taken from OASIS standards.

3.1 COWS syntax and informal semantics

COWS [16] is a formalism specifically devised for modelling (and analysing) service-oriented applications; in fact, its design has been influenced by the principles underlying

[2] For the sake of simplicity, we present here a fragment of COWS without linguistic constructs for dealing with forced termination, since such primitives have not been used in the protocol specification. We refer the interested reader to [16, 21] for the presentation of the full language and for many examples illustrating COWS peculiarities and expressiveness.

$s ::=$	(services)
$\quad \texttt{nil} \mid u\,.\,u!\,{<}u,\ldots,u{>} \mid \texttt{p}\,.\,\texttt{o}?\,{<}u,\ldots,u{>}\,.\,s$	(empty activity, invoke, receive)
$\mid s_1 + s_2 \mid s_1 \mid s_2 \mid [\texttt{n}\sharp]\,s \mid [\texttt{X}]\,s$	(choice, parallel, name & var. delim.)
$\mid * s \mid \texttt{A}(u,\ldots,u) \mid \texttt{let}\ \texttt{A}(u,\ldots,u) = s\ \texttt{in}\ s'\ \texttt{end}$	(replication, call, let definition)

Table 2. COWS syntax

the OASIS standard for orchestration of web services WS-BPEL [22]. The syntax of COWS, written in the 'machine readable' format accepted by the interpreter and the model checker CMC [23] that we use for the analysis, is presented in Table 2. It is defined using the following notational conventions: *variables* (ranged over by X, Y, ...) start with capital letters; *names* (ranged over by n, m, ..., p, p', ..., o, o', ...) start with digits or lower case letters; *identifiers* (ranged over by u, u_1, u_2, ... and used as non-terminal symbol only) are either variables or names; *service identifiers* (ranged over by A, A', ...) start with capital letters and each of them has a fixed non-negative arity. Names are used to represent communicable values, partners and operations.

Invoke and *receive* are the basic communication activities provided by COWS. Besides input and output parameters, both activities indicate an *endpoint*, i.e. a pair composed of a partner name p and an operation name o, through which communication should occur. An endpoint p.o can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p. An invoke $\texttt{p.o}!\,{<}u_1,\ldots,u_n{>}$ can proceed as soon as all arguments $u_1, \ldots, u_n$ are names (i.e. have been evaluated). A receive $\texttt{p.o}?\,{<}u_1,\ldots,u_n{>}\,.s$ offers an invocable operation o along a given partner name p. Partner and operation names can be exchanged in communication (although dynamically received names cannot form the endpoints used to receive further invocations). This makes it easier to model many service interaction and reconfiguration patterns.

A *choice* can be used to pick out one of a set of receive activities (in fact, the arguments of a choice are constrained to start with a receive activity) that are enabled for execution.

Execution of *parallel* terms is interleaved, except when a communication can be performed. Indeed, this must ensure that, if more than one matching receive is ready to process a given invoke, only one of the receives with greater priority (i.e. the receives that generate the substitution with 'smaller' domain, see [16, 21] for further details) is allowed to progress.

The *delimitation* operators are the *only* binders of the calculus: $[\texttt{n}\sharp]\,s$ and $[\texttt{X}]\,s$ bind n and X, respectively, in the scope s. Name delimitation can be used to generate 'fresh' private names (like the restriction operator of π-calculus), while variable delimitation can be used to regulate the range of application of the substitution generated by an inter-service communication. This takes place when the arguments of a receive and of a concurrent invoke along the same endpoint match and causes each variable argument of the receive to be replaced by the corresponding name argument of the invoke within the whole scope of variable's declaration. In fact, to enable parallel terms to share the state (or part of it), receive activities in COWS do *not* bind variables (which is different from most process calculi).

The *replication* operator $* s$ permits to spawn in parallel as many copies of s as necessary. This, for example, is exploited to model persistent services, i.e. services which can create multiple instances to serve several requests simultaneously.

Finally, the *let* construct permits to re-use the same 'service code', thus allowing to define services in a modular style; `let` $\mathtt{A}(u, \ldots, u) = s$ `in` s' `end` behaves like s', where calls to `A` can occur. A service *call* $\mathtt{A}(u'_1, \ldots, u'_n)$ occurring in the body s' of a construct `let` $\mathtt{A}(u_1, \ldots, u_n) = s$ `in` s' `end` behaves like the service obtained from s by replacing the formal parameters $u_1, \ldots, u_n$ with the corresponding actual parameters $u'_1, \ldots, u'_n$.

3.2 Protocol specification

Due to lack of space we only present the relevant part of the COWS specification of the XUA-based protocol and refer the interested reader to [17] for the overall specification.

To effectively take part to the protocol, each participant has to be able to call some internal functions, defined in some basic libraries provided by the programming language used to specify the service. These functions implement algorithms, such as SHA for hashing, RSA for public-key cryptography and AES for symmetric key cryptography, necessary to properly manage the data to be sent and received. An internal function can be rendered in COWS as a term of the following form[3]:

```
*( p.req?<inputData1> . p.resp!<inputData1,outputData1>
   + p.req?<inputData2> . p.resp!<inputData2,outputData2>
   + ... + p.req?<inputDatan> . p.resp!<inputDatan,outputDatan> )
```

where `p` indicates the partner name of the considered participant, while `req` and `resp` indicate the operations used to call the function and to receive the result, respectively. To guarantee that the result *outputData$_i$* is properly delivered to the caller, it is sent back together with the correlated *inputData$_i$*. In this way, if the same function $f(\cdot)$ is concurrently called, then the results will not be mixed up. Thus, in the example below

```
(p.req!<100> | [X] p.resp?<100,X> . s1)  |  (p.req!<250> | [Y] p.resp?<250,Y> . s2)
```

where we have two calls, the pattern-matching-based communication of COWS ensures that, irrespective of the execution order, the occurrences of variable `X` in s_1 will be replaced by $f(\mathtt{100})$, while the occurrences of `Y` in s_2 will be replaced by $f(\mathtt{250})$.

Each protocol participant `P` is rendered in COWS as a pair of service definitions of the form `A`(...) = `P` within a `let` construct:

```
P(p,... ) =
      [hashReq♯] [hashResp♯] [encReq♯] [encResp♯] [decReq♯] [decResp♯] ...
      ( sha1(p, hashReq, hashResp)
        | rsa1_5_PublicKey(p,encReq,encResp,decReq,decResp)
        | ...  other internal functions  ...
        | P_behaviour(p,hashReq,hashResp,encReq,...) )

   P_behaviour(p,hashReq,hashResp,encReq,...) = sP
```

[3] These COWS terms play a role similar to that of functions in the applied π-calculus [9, 24]

where p is the participant partner name and s_{P} is the COWS term modelling the participant's behaviour. Name delimitations are used here to make the functions sha1, rsa1_5_PublicKey, ... internal by declaring that hashReq, hashResp, encReq, ... are private operation names known to P_behaviour and to the internal functions, and only to them.

The term representing the consumer's behaviour is[4]

```
sts.rst!<c,msgId1,sts,user,salt,1000,timestamp1,uri,rst_req>
| [MsgId2] [Challenge] [Y] (
c.rstr?<Y,c,MsgId2,msgId1,Challenge>. c.fault!<Y,differentFrom,sts>
+ c.rstr?<sts,c,MsgId2,msgId1,Challenge>.
( -- Calculate the aes128 key based on his password
  c.hashReq!<pwd,salt,1000>
  | [DKey] c.hashResp?<pwd,salt,1000,DKey>.
  ( -- Decrypt the Challenge
    c.decReq!<DKey,Challenge>
    | [Nonce] [Created] [Context] [X](
      c.decResp?<DKey,Challenge,X,Nonce,Created,Context>.
        c.fault!<X,differentFrom,sts,for,Context>
      +  c.decResp?<DKey,Challenge,sts,Nonce,Created,Context>.
        ( -- Encode the response
          c.encReq!<gen_key,Nonce,1,c,msgId3,MsgId2,Context>
          | [EncData] c.encResp?<gen_key,Nonce,1,c,msgId3,MsgId2,Context,EncData>.
            ( -- Encode the generated key with sts public key
              c.encReq!<stsPubKey,gen_key>
              | [EncKey]  c.encResp?<stsPubKey,gen_key,EncKey>.
                ( -- Send the response to sts
                  sts.rstrr!<msgId3,MsgId2,sts,timestamp2,EncKey,EncData>
                  | [MsgId4] [SAMLTimestamp] [Signature]
                    -- Receive token back
                    c.rstrc?<c,sts,msgId3,MsgId4,SAMLTimestamp,user,uri,Signature>.
                    ( -- Query reg for the resource identified by uri
                      reg.storedQuery!<c,reg,sts,msgId5,SAMLTimestamp,user,uri,
                                        Signature,"Susan"> ) ) ) ) ) ) ) )
```

As expected, the consumer starts by invoking STS, by executing the invoke activity along the endpoint sts.rst and by sending the request security token data. This invocation corresponds to message (1) in Table 1, where the iteration number *int* is 1000 and the registry address specified in the RST is uri. Then, the consumer waits for message (2), by means of the two receive activities along c.rstr. Notice that, in accordance with the WS-Addressing standard and due to the pattern-matching mechanism regulating the COWS communication, only messages that carry the name msgId1 can be accepted by the consumer. Moreover, the identity of STS, i.e. sts, must be contained in the message, otherwise a fault is raised (represented by the invoke activity along the endpoint c.fault)[5]. Once message (2) is received, the consumer calculates the derived key by exploiting its internal hashing function (using operation hashReq and hashResp) and, similarly, decrypts the challenge (using operation decReq and decResp). Then, pattern-matching and the choice operator are used again to check the presence of the STS's identity within the challenge. Now, the consumer can prepare the response for STS, by

[4] The string -- indicates that the rest of the line is a comment (and is ignored by CMC).

[5] Notice that if both receives along c.rstr match an incoming message, hence the first argument is sts, due to the prioritized semantics of COWS only the second receive (which generates a smaller substitution) can progress.

encrypting the challenge data, where the nonce has been incremented by 1 (this is represented by the couple Nonce, 1). Differently from the abstract description of message (3) shown in Table 1, the COWS specification follows the concrete approach used in the implementation (based on XML encryption): thus, the AES algorithm is used to encrypt the data rather than RSA. The used symmetric key gen_key, supposed to be calculated by the consumer, is in its turn encrypted with RSA by using the STS's public key and attached to the message. Finally, when the message containing the token arrives (receive along c.rstrc), the consumer invokes the storedQuery operation (the XDS feature for querying Susan's documents) provided by the registry.

The term representing the *STS* behaviour is

```
* [C] [MsgId1] [User] [Salt] [Iteration] [Timestamp1] [URI] [RST]
  sts.rst?<C,MsgId1,sts,User,Salt,Iteration,Timestamp1,URI,RST>.
  ( -- Retrieve the User's password
    sts.getPwd!<User>| [Pwd] sts.getPwdResp?<User,Pwd>.
    ( -- Calculate the derived key
      sts.hashReq!<Pwd,Salt,Iteration>
      | [DKey] sts.hashResp?<Pwd,Salt,Iteration,DKey>.
        ( -- Create the challenge
          sts.encReq!<DKey,sts,nonce1,created1,contextId>
          | [Challenge] sts.encResp?<DKey,sts,nonce1,created1,contextId,Challenge>.
            ( -- Send the challenge to the consumer
              C.rstr!<sts,C,msgId2,MsgId1,Challenge>
              | -- Receive the challenge response
              [MsgId3] [Timestamp2] [EncKey] [EncData]
              sts.rstrr?<MsgId3,msgId2,sts,Timestamp2,EncKey,EncData>.
              ( -- Decrypt the encoded key
                sts.decReq!<stsPrivateKey,EncKey>
                | [Gen_key] sts.decResp?<stsPrivateKey,EncKey,Gen_key>.
                  ( -- Decrypt the encoded data
                    sts.decReq!<Gen_key,EncData>
                    | [MsgId3] sts.decResp?<Gen_key,EncData,nonce1,1,C,MsgId3,
                                   msgId2,contextId>.
                      -- Now, the consumer is authenticated
                      ( -- Create a token SAML
                        sts.hashReq!<sts,samlTimestamp,User,URI>
                        | [SAMLhash] sts.hashResp?<sts,samlTimestamp,User,URI,SAMLhash>.
                          ( -- Sign the hash code
                            sts.sign!<stsPrivateKey,SAMLhash>
                            | [Signature] sts.signResp?<SAMLhash,Signature>.
                              ( -- Send the token
                                C.rstrc!<C,sts,MsgId3,msgId4,samlTimestamp,
                                         User,URI,Signature> ) ) ) ) ) ) ) ) )
```

The replication operator $*$ at the beginning of the term specifies that STS is a persistent service, i.e. it is capable of creating multiple instances to serve several requests simultaneously. Thus, when it receives a message along the endpoint sts.rst, corresponding to message (1) of the protocol, it creates an instance initialized with the received data. The instance, by means of operations getPwd and getPwdResp, retrieves the user's password from a private database. Using the password, it can derive a symmetric key to encrypt the challenge, by exploiting again its internal functions. Invoke along C.rstr and the subsequent receive along sts.rstrr permit sending and receiving message (2) and (3), respectively. Now, by using stsPrivateKey, STS can decipher the symmetric key generated by the consumer, which is then used to decrypt the challenge response.

Notice that, pattern-matching in the communication along sts.decResp permits checking that the response contains the incremented nonce and the context; this guarantees that the sender of the message is really the consumer acting on behalf of the authorized user. Therefore, STS creates the token, by exploiting its internal functions, and sends it to the consumer.

Finally, the term representing the registry's behaviour is

```
* [Cust] [STS] [MsgId5] [TS] [User] [Uri] [Signature]
  reg.storedQuery?<Cust,reg,STS,MsgId5,TS,User,Uri,Signature,"Susan">.
  -- Validate the token
  ( -- Calculate the hash code of the token data
    reg.hashReq!<STS,TS,User,Uri>
    | [CalculatedHash] reg.hashResp?<STS,TS,User,Uri,CalculatedHash>.
      (  -- Retrieve the STS's public key
         reg.getKey!<STS>
         | [PubKey] reg.getKeyResp?<STS,PubKey>.
           ( -- Check the signature by using PubKey
             reg.check!<PubKey,Signature>
             | [Hash] reg.checkResp?<Signature,Hash>.
               [compare♯]
               ( -- Compare the hash codes
                 reg.compare!<CalculatedHash>
                 | [X] ( reg.compare?<X>. reg.attackDetected!<Cust>
                         + reg.compare?<Hash>. reg.deliveringResource!<Cust> ) ) ) ) )
```

When the registry receives a consumer's query, by means of the receive activity along the endpoint reg.storedQuery, it validates the token within the message. To this purpose, we assume that the registry has a private database storing the public keys of all trusted STSs, and can interact with it by calling the operations getKey and getKeyResp. Instead, to check the validity of the signature, it calls function signChecker by means of check and checkResp. After calling such function, the registry obtains the hash code of the signature (and stores it in the variable Hash); by comparing it with the re-calculated hash code (stored in the variable CalculatedHash) using the private operation compare, it can either detect that an attack has been performed (this is signaled by the activity reg.deliveringResource! < Cust >) or state that the token is valid. In this last case, the activity reg.deliveringResource! < Cust > is used to signal that the registry is ready to deliver the resource to the consumer. In fact, we do not model here message (6), since the flaw we are interested to capture concerns the previous message exchanges.

4 Protocol analysis

As shown in Figure 3 we have to deal with two types of communication channels: TLS protected channels for communicating with the registry and untrusted channels for communicating with the STS. We assume the intruder as any authorized user in the network (i.e. it owns an ATNA host certificate). Therefore, it can start any mutual authenticated TLS transaction with the registry and it can look in any message exchanged by STS. Basically, we consider the intruder model introduced by [25] for TLS channels and the well-known Dolev-Yao model [26] as regards the communication with STS along untrusted channels. We focus on an intruder that intercepts the message sent by STS containing the SAML token issued for the consumer (message (4)) and re-uses the token (without modifying it) for an its own query to the registry (message (5), sent by the intruder). This is rendered in COWS as

```
Intruder(i, c, sts, user, uri, reg) =
  [MsgId4] [TS] [Signature]
  c.rstrc?<c,sts,msgId3,MsgId4,TS,user,uri,Signature> .
    ( i.underAttack!<>
      | --Forwards the message to the consumer
        c.rstrc!<c,sts,msgId3,MsgId4,TS,user,uri,Signature>
      | --Performs the attack
        reg.storedQuery!<i,reg,sts,msgId5,TS,user,uri,Signature,"Susan"> )
```

Once the intruder has caught message (4) (receive activity along `c.rstrc`), besides forwarding the message to the consumer and querying the registry, it enables the invoke activity `i.underAttack`! <>. This activity is only used during the analysis to signal that the system is under attack. Notably, the intruder's query differs from the consumer's one for the first argument only, which is `i` instead of `c`.

The analysis of the protocol is carried out by exploiting CMC [23], a software tool that permits model checking SocL formulae over COWS specifications. SocL [27] is an action- and state-based, branching time, temporal logic specifically designed to express properties of service-oriented systems. Here, we are interested to look for the presence of security flaws in the protocol, which can be expressed in SocL as follows:

```
AG [request(samlToken,requestedBy,c)]
   not EF (systemUnderAttack(i) and deliveringResource(to,i))
```

This formula means that it holds *globally* (operator `AG`) that *if* (operator [·]) a SAML token has been requested by the consumer (action `request`(`samlToken`, `requestedBy`, `c`)), then it does *not* (operator `not`) hold that *eventually* (operator `EF`) the system will be under attack by intruder `i` (predicate `systemUnderAttack`(`i`)) and, at the same time, the registry will deliver the resource to `i` (predicate `deliveringResource`(`to`, `i`)).

The previous formula is stated in terms of *abstract* actions and predicates, meaning that, e.g., a token is requested or a resource is ready to be delivered, while the COWS specification is stated in terms of *concrete* actions, i.e. communication of data tuples along endpoints. To verify an abstract property over a concrete specification, CMC permits to specify a set of *transformation rules*, such as

```
Action *.rst<$requestor,*,*,*,*,*,*,*,*>
            -> request(samlToken,requestedBy,$requestor)
 State $attacker.underAttack! -> systemUnderAttack($attacker)
 State *.deliveringResource! <$X> -> deliveringResource(to, $X)
```

The first rule maps a concrete action involving the operation `rst` to the abstract action `request`(`samlToken`, `requestedBy`, `$requestor`), where the (meta-)variable `$requestor` will be replaced with the actual requestor during the on-the-fly model checking process, while the symbol $*$ is a wildcard. Similarly, the second and third rules map the actions involving operations `underAttack` and `deliveringResource` to the corresponding state predicates. We refer the interested reader to [27] for a complete account of abstraction rules.

As already mentioned in the Introduction, CMC returns FALSE when checking the above SocL formula over the abstracted COWS specification. In fact, the system can perform the following sequence of (abstract) actions:

```
request(samlToken,requestedBy,c); internal actions; challenge(samlToken);
   internal actions; challengeResp(samlToken); internal actions;
      response(samlToken,requestedBy,c);
         request(registryQuery,requestedBy,i); internal actions
```

and reach a state where both predicates systemUnderAttack(i) and deliveringResource(to, i) hold.

Now, let us modify the COWS specification to model the use of the *holder-of-key* confirmation method rather than the *bearer* method. With respect to the specification presented in Section 3, the main difference is that in the new STS specification the invoke sts.hashReq! < sts, samlTimestamp, User, URI >, used to generate the hash code of the SAML token data, is replaced with sts.hashReq! < sts, samlTimestamp, User, C, URI >. This time the result returned by CMC when checking the previous formula over the protocol specification is TRUE. In fact, the registry can detect that the intruder's query is fake by comparing the intruder's identity with the identity contained in the SAML token by means of ATNA credentials.

5 Concluding remarks

We have presented a formal model and analysis of a Web Service security protocol, for obtaining a XUA SAML authentication assertion, using the WS-Trust OASIS standard. To the best of our knowledge, our work is the first tentative to provide a formal study for IHE specifications. This kind of protocols are obtaining an ever increasing relevance since they are used to exchange patients' healthcare data and are widely adopted.

We have revealed a potential flaw in the specification and we have also proposed a solution. Afterwards, we have implemented the 'revised' protocol using WS-Trust 1.3, SAML 2.0, WS-Security and the WS-Security Username Token Profile 1.1. We have also used the Axis2 library (available at http://ws.apache.org/axis2) and the JBoss application server (http://www.jboss.org). Our Java implementation consists of four services: the *Document Consumer* and *Document Registry*, a *Document Repository* and a *Security Token Service*. All the XDS services are given as a courtesy of the Tiani "Spirit" company located in Vienna, Austria (http://www.tiani-spirit.com). The modified STS is available as Axis2 service at http://office.tiani-spirit.com:41081/SpiritIdentityProvider/services/STS09. A more detailed account of the implementation together with the COWS sources can be found in [17].

Related work. Microsoft Research proposes the TulaFale specification language [10, 9] for security analysis of web services. TulaFale uses CryptoVerif [28] as model checking engine. The main focus is on SOAP Message Rewrite attacks that we do not consider in our work since our signatures are defined by the SAML standard. In [10] the authors analyze WS-Trust for a secure exchange of a Security Context Token (SCT) while we consider WS-Trust for issuing a SAML token.

The SAML 1.0 and 2.0 specifications have been studied e.g. in [12, 29, 30]. However, they concentrate on the SAML Protocol and Profiles [31] to obtain SAML Authentication assertion, while we focus on WS-Trust. The work closest to ours is [12] where the SAML-based Single Sign-On for Google Apps is analyzed with the AVISPA [32] tool. A flaw in the Google implementation is found, where a fake Service Provider can potentially access a Google resource without the password of the user. Similarly to our scenario, the flaw discovered is in the format of the SAML assertion, that lacks the `Audience` list. In XUA, the `Audience` list must be contained in the assertion and refer to the registry, hence this kind of attack cannot occur.

Future work. As the above mentioned works and ours witness, to simply adopt WS-Security and WS-Trust does not guarantee absence of security flaws. Due to the widespread diffusion of such standards, especially in EHR, it is then worthwhile pursuing this line of research. Therefore, in the near future we plan to study the correctness of IHE security protocols for authentication and authorization, such as CCOW [33] and Patient Identifier Cross Referencing (PIX) [1].

References

1. The IHE Initiative: IT Infrastructure Technical Framework (2009) http://www.ihe.net.
2. OASIS Security Services TC: Assertions and protocols for the OASIS security assertion markup language (SAML) v2.02 (2005) http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.
3. OASIS/ebXML Registry Technical Committee: ebXML business process specification schema technical specification v2.0.4 (2006) http://www.ebxml.org.
4. OASIS Web Services Security TC: WS-Trust 1.3 (2007) http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.pdf.
5. GIP DMP: Dossier Médical Personnel A French Project, http://www.d-m-p.org.
6. ARGE-ELGA: Die österreich elektronische gesundheitsakte http://www.arge-elga.at.
7. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. Technical Report RFC 5246, IETF (August 2008)
8. OASIS Web Services Security TC: Web service security: SOAP message security (2006) http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.
9. Bhargavan, K., Fournet, C., Gordon, A., Pucella, R.: Tulafale: A security tool for web services. CoRR **abs/cs/0412044** (2004)
10. Bhargavan, K., Corin, R., Fournet, C., Gordon, A.: Secure sessions for web services. In: SWS, ACM (2004) 56–66
11. Kleiner, E., Roscoe, A.W.: On the relationship between web services security and traditional protocols. In: Mathematical Foundations of Programming Semantics (MFPS XXI. (2005)
12. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Abad, L.T.: Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In: the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008), Hilton Alexandria Mark Center, Virginia, USA, ACM Press (2008)
13. Lowe, G.: A hierarchy of authentication specifications, IEEE Computer Society Press (1997) 31–43
14. ACR-NEMA: Digital imaging and communications in medicine (dicom) (1995)
15. Health Level Seven organization: Hl7 standards (2009) http://www.hl7.org.

16. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: ESOP. Volume 4421 of LNCS., Springer (2007) 33–47
17. Masi, M., Pugliese, R., Tiezzi, F.: On secure implementation of an IHE XUA-based protocol for authenticating healthcare professionals (full version). Available at http://rap.dsi.unifi.it/cows/.
18. OASIS Security Services TC: SAML V2.0 Holder-of-Key Assertion Profile (March 2009) http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml2-holder-of-key-cd-01.pdf.
19. Gudgin, M., Hadley, M., Rogers, T.: Web Services Addressing 1.0 - Core. Technical report, W3C (May 2006) W3C Recommendation.
20. OASIS Web Services Security TC: Username token profile v1.1 (2006) http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf.
21. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze (2008) http://rap.dsi.unifi.it/cows.
22. OASIS WSBPEL TC: Web Services Business Process Execution Language Version 2.0. (2007) http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
23. ter Beek, M., Gnesi, S., Mazzanti, F.: CMC-UMC: A framework for the verification of abstract service-oriented properties. In: SAC, ACM (2009) To appear.
24. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL. (2001) 104–115
25. Broadfoot, P., Lowe, G.: On distributed security transactions that use secure transport protocols. Computer Security Foundations Workshop, IEEE **0** (2003) 141
26. Dolev, D., Yao, A.: On the security of public key protocols. Information Theory, IEEE Transactions on **29**(2) (1983) 198–208
27. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A model checking approach for verifying COWS specifications. In: FASE. Volume 4961 of LNCS., Springer (2008) 230–245
28. Blanchet, B.: CryptoVerif: : Computationally sound mechanized prover for cryptographic protocols. In: Dagstuhl seminar ”Formal Protocol Verification Applied”. (October 2007)
29. Groß, T.: Security analysis of the saml single sign-on browser/artifact profile. In: ACSAC, IEEE Computer Society (2003) 298–307
30. Hansen, S., Skriver, J., Nielson, H.: Using static analysis to validate the saml single sign-on protocol. In: WITS, ACM (2005) 27–40
31. OASIS Security Services TC: Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0 (2005) http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf.
32. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P.H., Heám, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The avispa tool for the automated validation of internet security protocols and applications. In: Proceedings of CAV'2005. LNCS 3576. Springer-Verlag (2005) 281–285
33. Hl7: Clinical context object workgroup, ccow (2001) http://www.hl7.org.au/CCOW.htm.