Model-checking Driven Security Testing of Web-based Applications

Alessandro Armando, Roberto Carbone
DIST, University of Genova
Genova, Italy
{*armando, carbone*}*@dist.unige.it*

Luca Compagna, Keqin Li, Giancarlo Pellegrino
SAP Research
Mougins, France
{*luca.compagna, keqin.li, giancarlo.pellegrino*}*@sap.com*

***Abstract*—Model checking and security testing are two verification techniques available to help finding flaws in security-sensitive, distributed applications. In this paper, we present an approach to security testing of web-based applications in which test cases are automatically derived from counterexamples found through model checking. We illustrate our approach by discussing its application against of the SAML-based Single Sign-On for Google Apps.**

***Keywords*-model checking; security testing; web-based applications;**

I. Introduction

We are witnessing a major paradigm shift in the way ICT systems and applications are designed, implemented and deployed: systems and applications are no longer the result of programming components in the traditional sense, but are built by composing services that are distributed over the network and aggregated in a demand-driven and flexible way. However, the new opportunities opened by this paradigm shift will only materialize if concepts, techniques, and tools for security and trust will be provisioned to ensure trustworthiness. A number of verification techniques are already available to help finding flaws in security-sensitive, distributed applications at the different phases of the service life-cycle:

- *Model checking* and related automated reasoning techniques have proved effective to detect subtle flaws in the logic of distributed applications. They can be fully automatic, but since the analysis is carried out on a formal model of the system (as opposed to the actual system) their applicability is usually confined to the design phase.
- *Security testing*, unlike model checking, can be used to check the behavior of the actual system. It has been successfully applied to authorization and similar application-level policies. A special form of security testing, namely penetration testing, is effective in finding low-level vulnerabilities in on-line applications (e.g., cross-site scripting), but heavily relies on the guidance and expertise of the user. Security testing is normally applied in later stages of the service life-cycle, i.e., during the deployment or even the consumption phase.

(We do not consider techniques based on code analysis here as the availability of source code cannot be always assumed.) These techniques are already routinely used to unveil serious vulnerabilities and are therefore going to play a central role in improving the security of web-based applications. However, there is an enormous potential in using these technologies in combination rather than in isolation. In fact, state-of-the-art security verification technologies, if used in isolation, do not provide automated support to the discovery of important vulnerabilities and of the associated exploits that are already plaguing complex, web-based, security-sensitive applications. On the one hand, while model checking is a key to the discovery of the subtle vulnerabilities due to unexpected interleavings of service executions, it provides no support to testing the actual services. On the other hand, penetration testing tools—by supporting the analysis of a single service at a time—lack the global view and the automated reasoning capabilities necessary to discover the kind of vulnerabilities found by model checkers, but provide both infrastructure and repertoires of testing techniques that are very useful to find exploits related to the high-level vulnerabilities found by model checkers. Also, the work on application-level security testing has been so far focused on access and authorization policies rather than on security properties in their generality. Therefore, security testing approaches, though they may serve as a good starting point, are not directly applicable to test key security properties (e.g., authentication) of web-applications.

In this paper we present an approach to security testing of web-based applications in which test cases are automatically derived from counterexamples found through model checking. Given a description of *(i)* the protocol used by the web-application to coordinate the component services and of the expected security properties and *(ii)* the testing environment including the specification of the System Under Test (SUT), our approach (cf. Figure 1) consists of the following steps:

1) *Modeling.* An abstract model, amenable to formal analysis, of the protocol is formulated, and message mapping information is specified.
2) *Model Checking.* The abstract model is automatically analyzed via model checking. If one of the expected security properties is violated, a counterexample is dis-

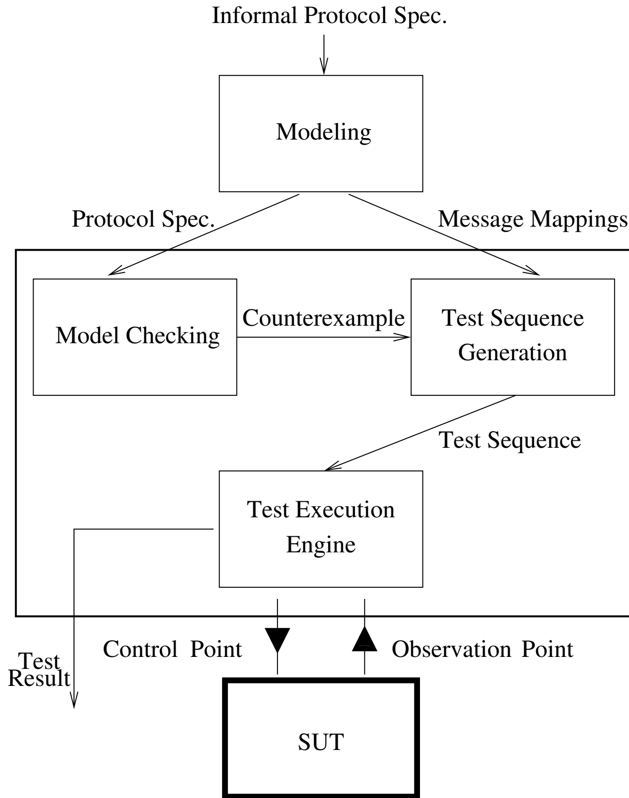


Figure 1. Approach Overview

covered and returned. If no counterexample is found, the procedure terminates.

3) *Test Sequence Generation.* An abstract test sequence is generated from the counterexample.
4) *Test Execution.* A concrete test sequence is generated from the abstract test sequence and then used to probe the SUT. The feedbacks from SUT are observed. Finally, test verdict is obtained.

We illustrate the applicability of the approach through its application on the SAML-based Single Sign-On (SSO) for Google Apps (http://www.google.com/apps/). As a case study we consider the version of the protocol used by Google until June 2008. This is particularly interesting as in May 2008 we found (with the help of a model checker) that the protocol was vulnerable to a serious authentication flaw that allowed a malicious service provider to impersonate a user on the Google Apps [1].[1] At that time we verified that the vulnerability could be exploited by manually probing the SSO service offered by Google. The approach presented in this paper shows that also this testing activity can be automated.

[1]We promptly informed both Google and US-CERT (http://www.kb.cert.org/vuls/id/612636) of the problem and Google released a new, patched version of the protocol in June 2008 (http://groups.google.com/group/google-apps-apis/browse_thread/thread/8183040d7980a2e0).

Structure of the paper. The SAML-based SSO for Google Apps is briefly presented in the next section (Section II). In Section III we describe the modeling activity. The model checking activity is presented in Section IV and the generation of abstract test sequences from the counterexamples returned by the model checker is illustrated in Section V. In Section VI, we discuss how concrete test sequences are obtained and executed against SUT. The related work is discussed in Section VII. We conclude in Section VIII with some final remarks.

II. Case Study: SAML-based SSO for Google Apps

Our case study is inspired by a real-world situation in which a Hospital takes advantage of the popular web-based Google Apps to handle its IT basic services like email, calendar, etc. The Hospital wants to keep the control of its identity management and not to add burden on its employees when they are using these services. The SAML-based SSO for Google Apps provides a solution to these requirements by allowing the Hospital to provide their employees a direct and transparent access to external services. As the name itself suggests, the SAML-based SSO for Google Apps is based on the SAML 2.0 Web Browser SSO Profile [2] (SAML SSO, for short), the de-facto standard for SSO for web-based applications.

The Hospital is required by health-care privacy regulations and directives (e.g., Data Protection Directive, 95/46/EC, supplemented with the Directive on privacy and electronic communications, 2002/58/EC) to ensure that patients' medical and data records are not disclosed to unauthorized entities. This high-level requirement is obviously violated if a Doctor's email or calendar can be accessed by some non authorized user by exploiting a flaw in the SSO mechanism.

A. SAML-based SSO services for Google Apps

SAML SSO defines an XML-based format for encoding security assertions as well as a number of protocols and bindings that prescribe how assertions should be exchanged in a variety of applications and/or deployment scenarios.

Many large software vendors have designed and developed SSO services based on SAML SSO. Hereafter we focus on the SAML-based SSO services released by Google for its Google Apps until June 2008. This version deviates from the SAML SSO standard in a few, but critical points that are detailed at the end of this section. The SAML-based SSO for Google Apps used the authentication protocol of Figure 2. Three roles take part in the protocol: a client (C), an identity provider (IdP) and a service provider (SP). C, typically a web browser guided by a user, aims at getting access to a service or a resource provided by SP. IdP authenticates C and issues corresponding authentication assertions. Finally, SP uses the assertions generated by IdP to give C access to the requested service.

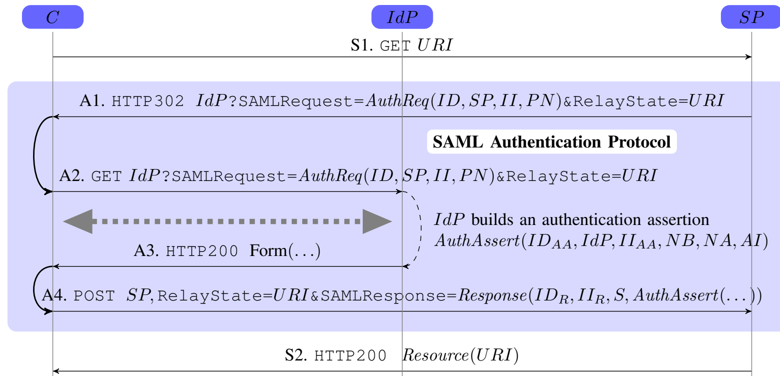


Figure 2. SAML-based SSO for Google Apps

Initially, C asks SP to provide the resource located at the address URI. SP then initiates the *SAML Authentication Protocol* by sending C a redirect response of the form

```
HTTP/1.1 302 Object MovedLocation: IdP?SAMLRequest=
      AuthReq(ID, SP, II, PN)&RelayState=URI
```

where $AuthReq(ID, SP, II, PN)$ abbreviates the XML expression:

```
<samlp:AuthnRequest
 xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
 ID="ID"
 Version="2.0"
 IssueInstant="II"
 ProtocolBinding="urn:oasis:names.tc:SAML:2.0:bindings:
      HTTP-Redirect"
 ProviderName="PN"
 AssertionConsumerServiceURL="SP"
/>
```

Here ID is a string uniquely identifying the request, II is the instant in which the authentication request is generated (e.g., "2008-05-02T08:49:40Z"), and PN is the service provider name (e.g., "google.com") that may slightly differ from the assertion consumer service URL (e.g., "https://www.google.com/a/example.com/acs") that is the most important entity at the service provider side and is indicated by SP.

IdP then challenges C to provide valid credentials and, if the authentication succeeds, IdP builds an authentication assertion $AuthAssert(ID_{AA}, IdP, C, II_{AA}, NB, NA, AI)$ of the form:

```
<Assertion ID="ID_AA"
  IssueInstant="II_AA"
  Version="2.0">
  <Issuer>IdP</Issuer>
  <Subject>
   <NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-
        format:emailAddress">C</NameID>
   <SubjectConfirmation Method="urn:oasis:names:tc:SAML
        :2.0:cm:bearer"/>
  </Subject>
  <Conditions NotBefore="NB" NotOnOrAfter="NA">
  </Conditions>
  <AuthnStatement AuthnInstant="AI">
    <AuthnContext>
     <AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:
          classes:Password
     </AuthnContextClassRef>
    </AuthnContext>
  </AuthnStatement>
</Assertion>
```

where ID_{AA} is a string uniquely identifying the authentication assertion, II_{AA} and AI are the instants in which the authentication assertion is generated and the authentication with C is completed respectively, NA and NB are timestamps establishing the validity of the authentication assertion.

IdP places the authentication assertion into a response message and then makes C (usually through client-side scripting) send this message to SP:

```
POST SP
HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xyz
RelayState = URI & SAMLResponse=
      Response(ID_R, II_R, S, AuthAssert(...))
```

where $Response(ID_R, II_R, S, AuthAssert(\ldots))$ is the result of encoding the following XML expression:

```
<samlp:Response
 xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
 xmlns="urn:oasis:names:tc:SAML:2.0:assertion"
 xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
 ID="ID_R"
 IssueInstant="II_R"
 Version="2.0">
 <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>...</SignedInfo>
  <SignatureValue>S</SignatureValue>
  <KeyInfo>...</KeyInfo>
 </Signature>
 <samlp:Status>
  <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:
       status:Success"/>
 </samlp:Status>
 AuthAssert(ID_AA, IdP, C, II_AA, NB, NA, AI)
</samlp:Response>
```

Here ID_R is a string uniquely identifying the response, II_R is the instant in which the response is generated, and S is the digital signature of the authentication assertion made by IdP, denoted by $\{AuthAssert(\ldots)\}_{K^{-1}_{IdP}}$.

The above protocol deviates from the standard SAML SSO protocol as follows:

- ID and SP are not included in the authentication assertion, and

- ID, SP and IdP are not included in the authentication response.

The first one is particularly dangerous as it makes authentication assertions generated by the IdP usable for other resources on different SPs [1].

B. The Hospital's federated environment

Figure 3 depicts the federated environment the Hospital would like to deploy. The Hospital acts as IdP authenticating and issuing authentication assertions for its Doctors (which play as Cs). These assertions are consumed by various service providers: besides Google that provides its Google Apps services, other SPs can take part in this landscape. For instance, we can imagine a Medical Insurance service provider and many others.

It suffices that the Hospital deploys the SAML-based SSO IdP service provided by Google and its Doctors could start taking advantage of SSO: they could thus authenticate once with the Hospital and get transparent access to all the services offered by the available providers. (In Figure 3 this is represented by the Doctor opening the Hospital's door with his key to get all the other doors opened.)

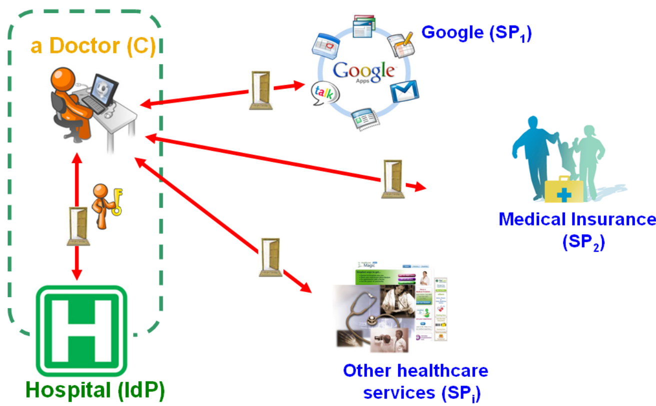


Figure 3. Hospital outsources basic IT services through federation.

III. Modeling

A. Protocol Specification

In order to specify the protocol we use HLPSL [3], the specification language of the AVISPA Tool [4]. HLPSL is a role-based language, meaning that the actions of each kind of participant are specified in a module, called a basic role. Basic roles are then instantiated and "glued" together into a composed role called `session`.

An excerpt of the HLPSL specification of our case study is given in Figure 4. The specification contains three basic roles: `client`, `serviceProvider`, and `identityProvider`. Notice that `SP2C_1`, `C2SP_1`, `SP2C_2`, and `C2SP_2` are variables of type `channel`, i.e.,

```
role serviceProvider(C,IdP,SP:agent,
             KIdP:public_key,
             SP2C_1,C2SP_1,SP2C_2,C2SP_2: channel,
             URI:  protocol_id) played_by SP def=
local State:nat, ID:text, Resource:text
const sent, rcvd:channel

init State:=1

transition
1. State=1 /\ rcvd(C2SP_1,C,C.SP.URI)
    =|>
    State':=3 /\ ID':=new() /\
    sent(SP2C_1,C,C.IdP.ID'.SP.URI)

2. State=3 /\
   rcvd(C2SP_2,C,SP.{C.IdP}_inv(KIdP).URI)
   =|>
   State':=5 /\ Resource':=new()
     /\ sent(SP2C_2,C,Resource'.URI)
     /\ request(SP,C,sp_c_uri,URI)
     /\ witness(SP,C,c_sp_resource,Resource')
     /\ secret(Resource',c_sp_response,{C,SP})
end role

role client( ... )
...
end role

role identityProvider ( ... )
...
end role

role session(C,IdP,SP:agent,
             KIdP:public_key,
             C2SP_1,SP2C_1,C2IdP,IdP2C,
              C2SP_2,SP2C_2:channel,
             URI:protocol_id)
def=
  init  confidential(IdP,C2IdP)
     /\ weakly_authentic(C,C2IdP)
     /\ authentic(IdP,IdP2C)
     /\ confidential(C,IdP2C)
     /\ link(C2IdP,IdP2C)
     /\ unilateral_conf_auth(C,SP,C2SP_2,SP2C_2)
     /\ dy(C2SP_1) /\ dy(SP2C_1)

  composition
     client(C,IdP,SP,KIdP,C2SP_1,SP2C_1,C2IdP,IdP2C,C2SP_2
          ,SP2C_2,URI)
  /\ serviceProvider(C,IdP,SP,KIdP,SP2C_1,C2SP_1,SP2C_2,
       C2SP_2,URI)
  /\ identityProvider(C,IdP,SP,KIdP,IdP2C,C2IdP)
end role

role enviroment()
def=
  const sp_c_uri,c_sp_resource:protocol_id,
        c,idp,sp:agent, ...

  intruder_knowledge={c,sp,idp,kidp,ki,uri_sp,inv(ki),...}

  composition
       session(c,idp,i,kidp,c2i_1,i2c_1,c2idp,idp2c,c2i_2,
            i2c_2,uri_i)
    /\ session(c,idp,sp,kidp,c2sp_1,sp2c_1,c2idp,idp2c,
         c2sp_2,sp2c_2,uri_sp)
end role

goal
 secrecy_of c_sp_resource
 authentication_on c_sp_resource
end goal
```

Figure 4. HLPSL Specification of the SAML-based SSO for Google Apps

they represent the communication channels used to exchange messages.

The **transition** section of a HLPSL specification contains a set of transitions. A transition consists of a set of preconditions (left hand side) and a set of effects that occur upon execution (right hand side). For example, the first transition in Figure 4 specifies that if the value of `State` is equal to `1` and a message is received on channel `C2SP_1` from agent `C` which contains values `C`, `SP`, and `URI`, the transition may fire and—when this happens—the value of `State` is set to `3`, a new value of `ID` is generated, and a message `C.IdP.ID'.SP.URI` is sent to `C` on channel `SP2C_1`. Here, `ID'` denotes the new value of variable `ID`.

A session is a parallel composition of several basic roles. A session has no transition section, but rather a composition section in which the basic roles are instantiated. At the same time, one usually declares all the channels used by the basic roles. Security properties of channels, such as confidentiality and authenticity, are specified. For example, in our case study, there are two channels `C2SP_2` and `SP2C_2` between instances of basic roles `client` and `serviceProvider`. The security properties of these two channels are declared to be `unilateral_conf_auth(C, SP, C2SP_2, SP2C_2)` which means that `C2SP_2` is confidential to `SP` and weakly authentic, `SP2C_2` is weakly confidential and authentic for `SP`, and that the principal sending messages on `C2SP_2` is the same principal that receives messages from `SP2C_2`. Please refer to [1] for the definitions of channel properties. A channel of type `dy`, is a channel which is under complete control of the intruder.

Finally, a top-level role `environment` is defined. This role contains global constants and a composition of one or more sessions, where the intruder may play some roles as a legitimate user. There is also a statement which describes the knowledge possessed by the intruder. Typically, this includes the names of all agents, all public keys, his own private key, any keys he shares with others, and all publicly known functions. The constant `i` is used to refer to the intruder.

In the SSO case study, three agents `c`, `idp` and `sp`, representing a client, an identity provider, and Google, respectively, are defined.

In order to perform model checking, we still need to specify the expected security properties. In the SSO case study, the following two security properties are considered: *(i)* SP authenticates C, i.e., at the end of its protocol run SP believes it has been talking with C; and *(ii)* $Resource$ must be kept secret between C and SP. In HLPSL, with the help of predefined macros, these two properties are specified as "`authentication_on c_sp_resource`" and "`secrecy_of c_sp_resource`", respectively.

B. Message Mappings

Messages are specified abstractly in the HLPSL specification. For example, in our case study, the authentication request message sent by the SP is represented as `C.IdP.ID'.SP.URI`. This level of abstraction is essential for analyzing the protocol using a model checker.

When a violation of an expected security property is identified by the model checker, a counterexample is returned as a sequence of abstract messages exchanged by the agents involved in the protocol. In order to generate the corresponding test sequence, abstract messages must be mapped to concrete ones. Thus, in the modeling step, in addition to specifying the behavior of the protocol, we need to describe how abstract messages correspond to concrete ones. In our approach, the message mapping information is organized in three tables: a message format table, an abstraction mapping table, and a concretisation mapping table.

An example of message format table is given in Table I. In this table, the message format column is a structured description of the concrete message format, in which we use `<value id="`*id*`" type="`*type*`"/>` to denote a field whose value needs to be filled when generating a concrete message, where id is the identifier of the field and *type* its type. Possible types include `nonce`, to denote freshly generated strings, and `timestamp`, to denote timestamps. Elements of the form `<encode type="`*type*`">...</encode>` and `<encrypt type="`*type*`">...</encrypt>` denote those fragments of the message which are encoded and encrypted respectively, where "*type*" denotes the corresponding encoding or encryption type. A complete specification of the language used to specify the concrete format is out of the scope of this paper. In the table, the column labeled by "#" denotes the relative position of the message in the HLPSL specification.

During the modeling step, concrete messages are investigated, relevant information for the analysis is abstracted into abstract fields in HLPSL messages. This mapping information is used while *(i)* generating concrete messages from abstract messages in order to probe the SUT and *(ii)* extracting abstract messages from concrete messages in order to check the feedback of the SUT. We record the mapping information in the concretisation mapping table and the abstraction mapping table for these two usages respectively.

The structure of the concretisation mapping table is depicted in Table II. As before, the column labeled by "#" refers to the position of the message in the HLPSL specification. The column "Concrete Field" refers to exactly one message element defined by the `<value .../>` tag in the message format table, while the column "Abstract Fields" refers to one or more elements in the corresponding HLPSL message. The way to obtain the concrete field from these abstract fields is specified in the column "Mapping Function". For example, the message at position 5 is the Authentication Response message sent to SP. The corresponding abstract message is `SP.{C.IdP}_inv(KIdP).URI`.

Table I
MESSAGE FORMAT TABLE

<table>
<tr><th>#</th><th>Format</th></tr>
<tr><td>1</td><td>...</td></tr>
<tr><td>2</td><td>HTTP1.1 302 Object Moved
Location: <value id="IdP" type="string"/>?
SAMLRequest=
<encode type="base64">
<samlp:AuthnRequest
xmlns:samlp="urn:oasis:names:tc:SAML:2.0:
protocol"
ID=<value id="ID" type="nonce"/>
Version="2.0"
IssueInstant=<value id="II" type="timestamp"/>
ProtocolBinding="urn:oasis:names.tc:SAML:2.0:
bindings:HTTP-Redirect"
ProviderName=<value id="PN" type="string"/>
AssertionConsumerServiceURL=<value id="SP"
type="string"/>
</encode> &
RelayState=<encode type="base64"><value id="URI"
type="string"/></encode></td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>5</td><td>POST <value id="IdP" type="string"/> HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: <value id="Length" type="length"/>
RelayState = <value id="URI" type="string"/> &
SAMLResponse =
<encode type="base64">
<samlp:Response
xmlns:samlp="urn:oasis:names:tc:SAML:2.0:
protocol"
xmlns="urn:oasis:names:tc:SAML:2.0:assertion"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
ID=<value id="ID_R" type="nonce"/>
IssueInstant=<value id="II_R" type="timestamp"/>
Version="2.0">
<Signature xmlns="http://www.w3.org/2000/09/
xmldsig#">
<SignedInfo>...</SignedInfo>
<SignatureValue><value id="S" type="signature
"/></SignatureValue>
<KeyInfo>...</KeyInfo>
</Signature>
<samlp:Status>
<samlp:StatusCode Value="urn:oasis:names:tc:
SAML:2.0:status:Success"/>
</samlp:Status>
<Assertion ID=<value id="ID_AA" type="nonce"/>
IssueInstant=<value id="II_AA" type="timestamp
"/>
Version="2.0">
<Issuer><value id="IdP" type="string"/></Issuer
>
<Subject>
<NameID Format="urn:oasis:names:tc:SAML:1.1:
nameid-format:emailAddress">
<value id="C" type="string"/> </NameID>
<SubjectConfirmation Method="urn:oasis:names:
tc:SAML:2.0:cm:bearer"/>
</Subject>
<Conditions NotBefore=<value id="NB" type="
timestamp"/> NotOnOrAfter=<value id="NA"
type="timestamp"/> />
<AuthnStatement AuthnInstant=<value id="AI"
type="timestamp"/>>
<AuthnContext>
<AuthnContextClassRef>urn:oasis:names:tc:SAML
:2.0:ac:classes:Password
</AuthnContextClassRef>
</AuthnContext>
</AuthnStatement>
</Assertion>
</samlp:Response>
</encode></td></tr>
<tr><td>...</td><td>...</td></tr>
</table>

Table II
CONCRETISATION MAPPING TABLE

#	Concrete Field	Abstract Fields	Mapping Function
...	...	...	...
5	SP	SP	$\lambda x.x$
5	URI	URI	$\lambda x.x$
5	IdP	IdP	$\lambda x.x$
5	C	C	$\lambda x.x$
...	...	...	...

Table III
ABSTRACTION MAPPING TABLE

#	Abstract Field	Concrete Fields	Mapping Function
...	...	...	...
2	IdP	IdP	$\lambda x.x$
2	ID′	ID	$\lambda x.x$
2	SP	SP	$\lambda x.x$
2	URI	URI	$\lambda x.x$
...	...	...	...

In this case, all the values of four concrete message fields can be obtained from the corresponding abstract message fields using simple functions. Although in the example described all the mapping functions are identity functions, please note that in general more complex functions could be used.

The structure of abstraction mapping table is depicted in Table III. In this table, the column "Abstract Field" refers to exactly one message element in HLPSL message, while the column "Concrete Fields" refers to one or more fields defined by the `<value .../>` tag in the message format table. The way to obtain the abstract field from these concrete fields is specified in the column "Mapping Function". The message at position 2 is the Authentication Request message sent by SP. The corresponding abstract message is `C.IdP.ID′.SP.URI`. In this case, all the values of four abstract message fields can be obtained from the corresponding concrete message fields using simple functions.

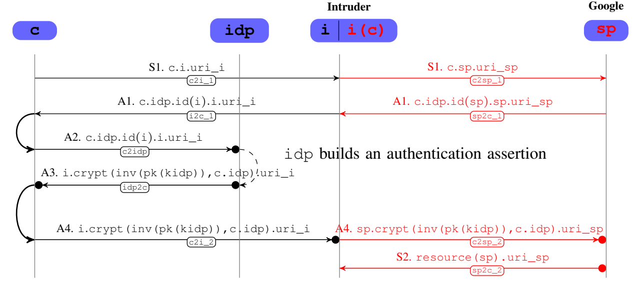
C. Testing Environment

An important issue in specifying the testing environment is to decide what the System Under Test (SUT) is. In other words, we need to decide which agents in the system belong to the SUT and which are simulated by the tester. We call the first set of agents *SUT agents* and the others *tester agents*. This decision depends on the specific security protocol analysis and testing problem. In the SSO case study, the SUT is the service provider provided by Google, which is represented by the agent `sp` in the system model. The agents `c`, `idp`, and the intruder `i` are all simulated by the tester.

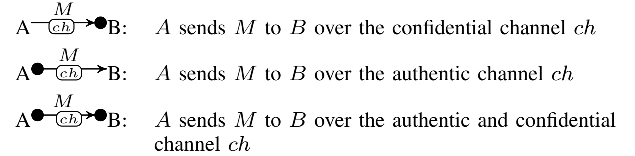
In addition to the SUT, we must define the *Control Point*, i.e., where input can be fed to the SUT, and the *Observation Point*, i.e., where the output of the SUT can be observed. In the SSO case study, the Internet connection with the Google

```
Step 0: sent(c, c, i, c.i.uri_i, c2i_1)
Step 1: sent(i, i, c, c.idp.id(i).i.uri_i, i2c_1)
Step 2: rcvd(c, i, i, c.idp.id(i).i.uri_i, i2c_1)
Step 3: sent(c, c, idp, c.idp.id(i).i.uri_i, c2idp)
Step 4: rcvd(idp, c, c, c.idp.id(i).i.uri_i, c2idp)
Step 5: sent(idp, idp, c, i.crypt(inv(pk(kidp)), c.idp).
     uri_i, idp2c)
        sent(i, c, sp, c.sp.uri_sp, c2sp_1)
Step 6: rcvd(sp, i, c, c.sp.uri_sp, c2sp_1)
        rcvd(c, idp, idp, i.crypt(inv(pk(kidp)), c.idp).
            uri_i, idp2c)
Step 7: sent(sp, sp, c, c.idp.id(sp).sp.uri_sp, sp2c_1)
        sent(c, c, i, i.(crypt(inv(pk(kidp)), c.idp).uri_i
            , c2i_2)
Step 8: rcvd(i, c, c, i.(crypt(inv(pk(kidp)), c.idp).uri_i
     , c2i_2)
Step 9: rcvd(c, sp, sp, c.idp.id(sp).sp.uri_sp, sp2c_1)
Step 10: sent(i, c, sp, sp.(crypt(inv(pk(kidp)), c.idp).
     uri_sp, c2sp_2)
Step 11: rcvd(sp, i, c, sp.(crypt(inv(pk(kidp)), c.idp).
     uri_sp, c2sp_2)
Step 12: sent(sp, sp, c, resource(sp).uri_sp, sp2c_2)
```

Figure 5. Counterexample in the SSO Case Study



Legend:



A sends M to B over the confidential channel ch

A sends M to B over the authentic channel ch

A sends M to B over the authentic and confidential channel ch

Figure 6. Attack on the SAML-based SSO for Google Applications

service provider is both the control point and the observation point.

IV. Model Checking

When the model does not satisfy the expected security property, a counterexample is generated by the model checker.[2] The counterexample consists of a sequence of steps. Each step includes one or more message sending or receiving actions. There is no temporal order among actions within a step. Message sending is represented as `sent`(*sender, as-is, target, m, ch*), meaning that an agent *sender* sent message *m* on channel *ch* to agent *target* pretending to be agent *as-is*. Message reception is denoted by `rcvd`(*receiver, sender, as-is, m, ch*), meaning that message *m*, sent by agent *sender*, has been received on channel *ch* by agent *receiver* and *receiver* considers the message as if it were sent by agent *as-is*.

By running SATMC against the HLPSL specification in Figure 4 the counterexample in Figure 5 is returned. A pictorial description of the attack is depicted in Figure 6.

In the attack, `c` initiates a session of the protocol to access a resource located at the address `uri_i`, provided by the (malicious) service provider `i` that in parallel starts a new session of the protocol with Google (`sp`) pretending to be `c` and that mischievously reuses the authentication assertion received by `c` to trick Google into believing he is `c`. The attack completes with the delivery of the `resource(sp)` (whose access should be reserved to `c`) to `i`. This attack represents a violation of the two security properties (authentication and secrecy) that the protocol is expected to enjoy.

[2]For our experiments we used the SATMC model checker, one of the back-ends of the AVISPA Tool.

V. Abstract Test Sequence Generation

When a counterexample is found, we need to check whether the corresponding security vulnerability exists also in the actual implementation. As the first step toward that purpose, an abstract test sequence is derived from the counterexample generated by the model checker. This is done by checking the actions in the counterexample one by one. In each message sending action from a tester agent to a SUT agent, the sender and target fields are replaced by tester and the SUT, respectively. In each message sending action from a SUT agent to a tester agent, the sender and target fields are replaced by the SUT and the tester, respectively. All other actions are removed. In the SSO case study, the abstract test case obtained is as follows:

`sent`(tester, `c`, SUT, `c.sp.uri_sp`, `c2sp_1`) (1)

`sent`(SUT, `sp`, tester, `c.idp.id(sp).sp.uri_sp`, `sp2c_1`) (2)

`sent`(tester, `c`, SUT, `sp.crypt(...).uri_sp`, `c2sp_2`) (3)

`sent`(SUT, `sp`, tester, `resource(sp).uri_sp`, `sp2c_2`) (4)

where (1) is obtained from the second `sent` action of Step 5 in Figure 5, (2) is obtained from the first action of Step 7, (3) is obtained from the action in Step 10, and (4) is obtained from the action in Step 12.

In the abstract test case, the message sending actions from the tester to the SUT are used to probe the SUT. The message sending actions from SUT to tester are expected outputs of SUT in the following sense:

- During the testing procedure, if the message received from SUT is not consistent with the expectation, this means the system implementation does not conform to the system model, this fault is reported, and the testing procedure is terminated.
- Otherwise, if it is not the last message receiving step, the testing procedure is continued.

- If it is the last message receiving step, this means the vulnerability identified in the system model does exist in the system implementation. This observation is reported, and the testing procedure is terminated.

VI. Test Case Concretization and Execution

After obtaining the abstract test case, the next activity is to concretize and execute the test case. This could be performed automatically by a test execution engine. To this end, besides the abstract test case, information about the testing configuration is necessary. This information includes the names and addresses of the involved agents, the keys used, etc.

In the test execution engine, two are the main components: *message generator* and *message checker*.

A. Message Generation

The message generator takes an HLPSL message and a group of corresponding values as input, performs the following operations, and generates concrete HTTP messages as output. As an example, we suppose the HLPSL message is `SP.{C.IdP}_inv(KIdP).URI`, and the corresponding Authentication Response message needs to be generated.

- The message generator locates the HLPSL message in the protocol specification to obtain the message number. In the example, the message number is 5.
- The message generator looks up the Message Format Table to obtain the concrete message format using the message number. In the example, the message format can be seen in Table I.
- The message generator generates the concrete HTTP message as follows:
  - For the fixed structure and field values of the message, generate as it is. In the example, "`<samlp:Response...`" is such a fixed part.
  - For fields being referred to in the Concretisation Mapping Table, the message generator looks up in the Concretisation Mapping Table, identifies the corresponding abstract elements, calculates the concrete value according to the mapping function specified, and puts the value into the concrete message. In the example, "`<value id="IdP" type="string">`" is such a field.
  - For fields that must be calculated at runtime but not depending directly on abstract elements, such as timestamps, the values are generated according to the time of test execution. In the example, "`<value id="II_R" type="timestamp">`" is such a field.
  - For parts of the concrete message that need to be calculated according to directives, such as `<encode>...</encode>` and `< encrypt>...</encrypt>`, the parts are calculated according to encoding type or encryption type specified. In the example, the part from "`<samlp:Response...`" to "`</samlp:Response>`" needs to be encoded.

Since the message format and the way to generate all the parts of the concrete message are specified in the Message Format Table and Concretisation Mapping Table, the message generator is protocol independent and can be automated.

B. Message Checker

The message checker takes a concrete message and an expected HLPSL message as inputs, and extracts values of abstract elements in the HLPSL message as outputs. The outline of its operation is described as follows. As an example, we suppose the expected HLPSL message is `C.IdP.ID'.SP.URI`, which is the Authentication Request message.

- The message checker locates the HLPSL message in the protocol specification to obtain message number. In the example, the message number is 2.
- The message checker looks up the Message Format Table to obtain the concrete message format using the message number. In the example, the message format can be seen in Table I.
- The message checker parses the incoming concrete message according to the expected concrete message format. During parsing, the message checker verifies the message format and extracts the values of message fields.
  - If the message format does not match the expectation, or certain values are wrong according to the way of calculation specified in the message format table, an error is reported.
  - Otherwise, the values of the abstract elements in the HLPSL message are calculated according to the mapping functions specified in Abstraction Mapping Table. In the example, the abstract element "`ID'`" is obtained from the string after "`ID=`" and before "`Version="2.0"`".

Similar to the message generator, with the help of the Message Format Table and Abstraction Mapping Table, the message checker is protocol independent and can be automated.

C. Test Execution Engine

With all this information and software modules, the test execution engine processes the abstract test case action by action. For each message sending action to the SUT, the test execution engine passes HLPSL message and corresponding concrete values obtained from testing configuration and stored values to the message generator, obtains concrete message as feedback, and sends the message to SUT. For a message sending action from SUT, the test execution

engine waits for receiving a message from the SUT, passes the received message and expected HLPSL message to the message checker, and stores the received values for future usage. If an error is reported by the message checker, it is treated as described in Section V.

The test execution engine is specific to model checker but protocol independent. This means it could be applied generally to the analysis of different security protocols.

D. Before Test Execution

From the abstract test case, the channels connecting the tester and SUT are identified. From the model, we can see the security properties of the channels. In order to execute the test case, some configuration and message exchange need to be performed to establish the communication channels. For example, in order to make the channel `C2SP_2` confidential and authentic, an SSL/TLS channel must be preliminarily established between the tester and SUT. In the SSO case study, in addition to channel configuration, we also need to register an IdP to Google to make Google trusts it and sends Authentication Request redirecting to it.

In [1], after identifying the vulnerability by model checking, we reproduced manually the identified attack in an actual deployment of the SAML-based SSO for Google Applications. By using the approach presented in this paper the testing has been carried out automatically by the test execution engine. Since Google updated its implementation of SAML-based SSO after the discovery reported in [1], the test execution engine reports that the system implementation does not conform to the system model. However, the HTTP messages generated by the test execution engine are equivalent to the ones we used when we manually probed the flawed version of the SAML-based SSO for Google Apps.

VII. Related Work

In the past fifteen years, automated analysis of security protocols has been widely studied and several analysis tools with different degrees of automation have been developed (see e.g., [5], [6], [7], [4]). There have also been many applications of model checking to the formal analysis of security aspects of Web Services (e.g., [8], [9], [10], [11], [12]). These approaches however mostly focus on design time verification, and fall short in validating whether the actual implementations satisfy the desired properties at deployment and consumption time.

There has been recent interest in combining ideas from formal verification and testing in the program analysis community [13]. The respective algorithms target static analysis of program source, and mostly focus on reachability properties defined as undesirable states in programs. These results are however not applicable to the problems we are facing: source code is not always available in validating security protocols, and, due to concurrency, security properties often cannot be expressed simply as local states.

Model-based testing has been applied to security-relevant systems in the past, e.g., [14], [15], [16], [17], [18]. The fundamental problem of translating specified (universal) security properties to test case specifications has, however, not been solved yet. The problem of relating the abstract and concrete levels continues to be solved in an ad-hoc manner only [19]. Moreover, these approaches do not propose a coherent generic methodology for security testing.

VIII. Conclusions

In this paper, we have presented an approach to security testing of web-based applications in which test cases are automatically derived from counterexamples found through model checking. With message format and message mapping information properly specified and the test execution engine, test case generation and execution could be automated to check whether security flaws identified by model checking exist in the actual implementation. This approach has been applied successfully in the SSO case study.

Acknowledgment

This work was partially supported by the FP7-ICT-2007-1 Project no. 216471, “AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures” (www.avantssar.eu).

References

[1] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. T. Abad, “Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps,” in *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*, V. Shmatikov, Ed. ACM Press, 2008, pp. 1–10.

[2] OASIS Consortium, “Security Assertion Markup Language V2.0 Technical Overview,” http://wiki.oasis-open.org/security/Saml2TechOverview, Mar. 2008.

[3] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron, “A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols,” in *Proc. SAPS’04*. Austrian Computer Society, 2004.

[4] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Heám, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, “The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications,” in *Proceedings of the 17th International Conference on Computer Aided Verification (CAV’05)*. Springer-Verlag, 2005, available at www.avispa-project.org.

[5] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe, *Modelling and Analysis of Security Protocols*. Addison Wesley, 2000.

[6] B. Blanchet, “Automatic verification of cryptographic protocols: A logic programming approach (invited talk),” in *Proceedings of PPDP’03*. ACM Press, 2003, pp. 1–3.

[7] J. K. Millen and G. Denker, “Capsl and mucapsl,” *Journal of Telecommunications and Information Technology*, vol. 4, pp. 16–27, 2002.

[8] M. Backes and T. Gross, “Tailoring the dolev-yao abstraction to web service realities,” in *ACM Secure Web Services Workshop (SWS)*, 2005, pp. 65–74.

[9] M. Backes, S. Mödersheim, B. Pfitzmann, and L. Viganò, “Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario,” in *Proceedings of FOSSACS’06*, ser. LNCS 3921. Springer, 2006, pp. 428–445.

[10] X. Fu, T. Bultan, and J. Su, “Analysis of interacting bpel web services,” in *WWW ’04: Proceedings of the 13th international conference on World Wide Web*. ACM Press, 2004, pp. 621–630.

[11] M. Hondo, N. Nagaratnam, and A. Nadalin, “Securing web services,” *IBM Systems Journal*, vol. 41, no. 2, pp. 228–241, 2002.

[12] G. Salaün, L. Bordeaux, and M. Schaerf, “Describing and reasoning on web services using process algebra,” in *Proceedings of the IEEE International Conference on Web Services (ICWS’04)*. IEEE Computer Society, 2004.

[13] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, “Synergy: a new algorithm for property checking,” in *SIGSOFT ’06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2006, pp. 117–127.

[14] P. A. P. Salas, P. Krishnan, and K. J. Ross, “Model-based security vulnerability testing,” *Australian Software Engineering Conference*, vol. 0, pp. 284–296, 2007.

[15] E. Martin and T. Xie, “A fault model and mutation testing of access control policies,” in *WWW ’07: Proceedings of the 16th international conference on World Wide Web*. New York, NY, USA: ACM, 2007, pp. 667–676.

[16] J. Jürjens, “Model-based security testing using umlsec: A case study,” *Electr. Notes Theor. Comput. Sci.*, vol. 220, no. 1, pp. 93–104, 2008.

[17] P. P. Salas and P. Krishnan, “Testing privacy policies using models,” in *SEFM ’08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 117–126.

[18] M. Zulkernine, M. F. Raihan, and M. G. Uddin, “Towards model-based automatic testing of attack scenarios,” in *SAFECOMP*, ser. Lecture Notes in Computer Science, B. Buth, G. Rabe, and T. Seyfarth, Eds., vol. 5775. Springer, 2009, pp. 229–242. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04468-7

[19] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing,” University of Waikato, New Zealand, Tech. Rep. 04/2006, April 2006.