RZ 3732 (# 99742) 05/06/2009
Computer Science 14 pages

Research Report

Test Cases for a WS-Federation Passive Requestor Profile

T. Gross, B. Pfitzmann

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

{tgr,bpf}@zurich.ibm.com

Test Cases for a WS-Federation Passive Requestor Profile

Thomas Groß and Birgit Pfitzmann
IBM Zurich Research Lab
{tgr,bpf}@zurich.ibm.com

December 27, 2004

Abstract

Recent advances in proving the security of browser-based identity federation protocols do not necessarily carry over to real-world deployments of the protocols proven secure. The protocols are not executed by a single well-defined protocol machine, but by the cooperation of multiple complex products. Thus, formal verification faces a vast complexity rendering it useless. To gain confidence in real world deployments of federated identity protocols in spite of this obstacle, we systematically derive black-box test cases from the precise protocol specification of a federated identity protocol proven secure.

1 Introduction

In [1], Gross and Pfitzmann proved the security of the WS-Federation Passive Requestor Interop profile [5] (WSFPI). This was the first positive security statement for a browser-based protocol as well as for WS-Federation [4]. Although the security proof guarantees the authenticity of the WSFPI protocol, this result does not necessarily carry over to real implementations. This stems from the fact that such implementations are deployed to real-world products and are not developed according to the formal protocol definition of WSFPI. Therefore, on the one hand real deployments do not necessarily fulfill the assumptions introduced by [1]. On the other hand, it is still to be shown that real protocol implementations are correct in the sense of the formal definition of [1]. In this paper, we concentrate on the latter issue, the correctness of WSFPI implementations.

In order to proof correctness of a concrete protocol implementation against a formal specification such as WSFPI, the preferred method would be a formal verification of the implementation's source code. However, such a verification is not realistic for a real-world deployment of WSFPI. In such a deployment the protocol is carried out by cooperation of multiple complex products, but not by a single well-defined protocol machine. In addition, there is no suitable tool that supports the analysis of large amounts of native source code. Thus, we face a vast complexity hampering a formal verification.

Therefore, we use testing as less formal, but widely-used technique in software engineering as an alternative. Of course, the test cases only check whether certain vulnerabilities occur in a concrete implementation, which implies that this method cannot provide a complete correctness proof. We can still gain confidence in the correctness statement by deriving test cases systematically from the formal protocol specification. This method is weaker than a formal verification of the implementation's source code, but still has some advantages compared to the verification. The test cases do not need to know details of the source code and are therefore more generic than a verification of a specific implementation; they may be used for various implementations of the WS-Federation Passive Requestor Interop profile. In addition, the testing method is widely accepted in industry and has a low entry-barrier for product development.

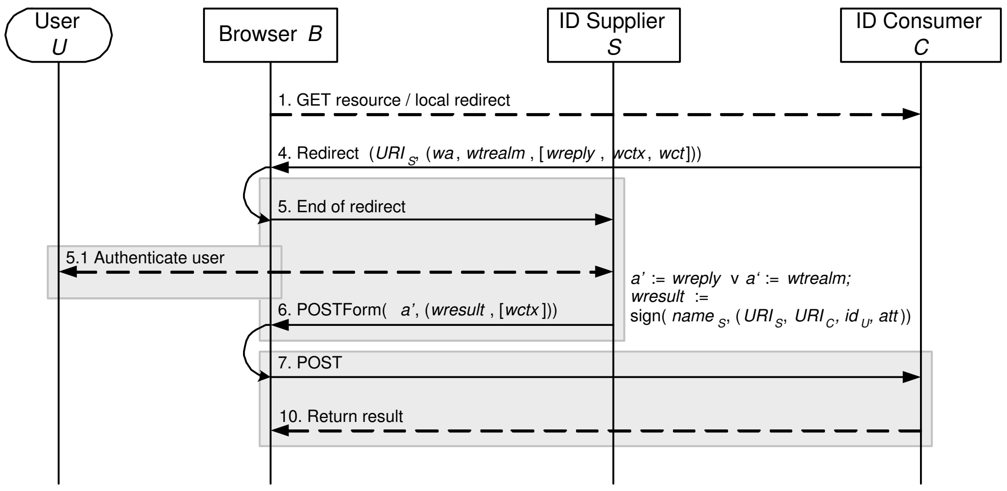
User U
Browser B
ID Supplier S
ID Consumer C
1. GET resource / local redirect
4. Redirect (URI_S, (wa, wtrealm, [wreply, wctx, wct]))
5. End of redirect
5.1 Authenticate user
a' := wreply ∨ a' := wtrealm;
wresult :=
sign(name_S, (URI_S, URI_C, id_U, att))
6. POSTForm(a', (wresult, [wctx]))
7. POST
10. Return result

Figure 1: WSFPI protocol with abstract parameters. Steps with uninterrupted lines are actually specified in the protocol. The boxes denote secure channels.

1.1 Organization

We organize the remainder of this paper as follows: in Section 2, we give a short overview over the WSFPI protocol. Section 3 explains our general testing methodology and how we derive test cases. In Section 4, we classify the test cases according to three dimensions. We describe in Section 5 how we define a test case and which notation we use. Section 6 contains the test cases for the identity supplier, whereas Section 7 covers the test cases for the identity consumer. In Section 8, we conclude this paper.

2 Overview of the WSFPI Protocol

In [1], Gross and Pfitzmann provided a formal protocol definition for an instantiation of the WS-Federation Passive Requestor Interop profile (WSFPI) [3]. They have proven WSFPI to be secure under certain assumptions; more precisely the authors have proven WSFPI to provide entity authentication of a user and to bind this authentication to a certain secure channel (authenticity). In this section, we introduce WSFPI as defined in [1]. Figure 1 depicts the message flow of the WSFPI protocol when no error occurs. Browser B communicates on behalf of user U with two other principals C and S. User U wants to sign-in at an identity consumer C ("resource" or "destination site" in other terminologies) using WSFPI. An identity supplier S authenticates U and confirms its identity to identity consumer C by means of a signed SAML assertion.

Steps 1 and 10 show that user U is assumed to browse at identity consumer C before the protocol (Step 1) and to get some application-level response after the protocol (Step 10). In [3] these arrows are drawn to a "WS resource" different from the "resource IP/STS" that executes the protocol and thus corresponds to our identity consumer. This reflects that the browser will typically be intercepted or redirected when accessing the resource. However, as [3] says nothing about Steps 1 and 10 except that they must be supported, this does not matter: we will in fact only analyze Steps 4 to 7. Steps 4-5 redirect the browser to identity supplier S, the unspecified Step 5.1 authenticates the user to the identity supplier, and Steps 6-7 essentially redirect the browser back to identity consumer C with a signed SAML assertion as response. The assertion contains an authentication statement and an attribute statement about user U.

The figure contains all the exchanged top-level parameters with their original names. Additionally, the most important elements of $wresult$ are shown. In both abstract messages, Redirect and POSTForm,

$\downarrow$Who/about$\rightarrow$	U	S	C
U	-	$MetaS$: ch_id, $login$	-
S	$MetaU$: id, $login$	-	$MetaC$: URI, att_n
C	-	$certS$ ($\rightarrow$ $nameS$), $URIS$	ch_id

Table 1: Summary of metadata in database notation

the first parameter in the figure is the address and the second parameter the payload, here a list of the protocol parameters. Square brackets mean that parameters are optional. The end of a redirect message gets its parameters from the redirect message, and the POST message gets them from the message denoted POSTForm. In the latter case a form, typically including a script, is used to make the browser or user POST the described message.

In Table 1, we summarize the metadata each party acquires during setup.

3 Methodology

We use the precise definition of the WSFPI protocol in [1] as basis for deriving test cases. The WSFPI definition basically specifies the behavior of correct protocol machines, i.e. which actions the machines take and how they verify the messages received. If we want to test a concrete protocol implementation against this specification, it is not sufficient to simulate a successful protocol run, in which all messages are well formed and all participants behave honestly. Such a simulation would test whether the implementation fulfills a minimum of the specification's functionality, but not its robustness (correct behavior in spite of incorrect inputs and malicious behavior of other parties). To judge the robustness of a protocol implementation, we generate test cases according the following method: we test whether a principal performs all security checks prescribed by the protocol upon receiving a message and whether the principal is robust against erroneous metadata the principal relies on. To do this systematically and transparently, we name the security checks in the protocol definition such as in Figure 3 for the identity supplier. We label the checks with the name of the principal principal performing the check (S or C) and a sequence number, where we denote the label by a box, e.g. $\boxed{S.1}$. We use the same notation for consistency assumptions about a principal's metadata. Our test cases are supposed to cover all situations in which something goes wrong. This set of situations is the complement of the one single good situation, in which everything is alright. Thus, we iterate over all security checks and over all possible variants of test messages that are supposed to trigger security checks. Note that there is a grey area between the security checks specified and the correct message format, which we do not address in the test cases.

As this set is too large to be handled efficiently, we cut it down into a manageable size by means of classification and exemplary black-box testing. The exemplary black-box testing simplifies the testing in two ways. Firstly, we do not take into account the source code of the protocol machine to be tested. We consider the protocol machine as a black box and test its correct output behavior given inputs that we chose as test cases. Therefore, our test cases only provide low coverage of program statements/states/flows/... and should be complemented with white-box tests[1]. Secondly, we do not iterate over all bad inputs possible, but only consider some specific examples of inputs chosen for the test cases. These examples are chosen such that the classes specified are covered. We then specify the examples by describing the difference to a correct input. We describe a classification by multiple dimensions for the test cases in Section 4. Section 5 introduces a notation for the technique of exemplary black-box testing.

[1]A white-box test takes into account the a program's source code for the choice of the test cases.

4 Test Case Classification

We classify the test cases derived from the WSFPI specification according to three dimensions:

Area of the protocol We consider certain areas of the protocol to which one can turn one's attention without lacking valuable information:

- Setup phase. Covers the test of the meta data.
- Protocol Flow. Considers channel types used and message order.
- User interaction. Handling of the user's inputs.
- Message. Format of a HTTP message and the corresponding parameters.
- Token. Format of the security token and the token wrapper.

Test case type The type of the test case classifies how an object of a test case (e.g. a message) may differ from the original specification.

- Additional element (A): an object may contain additional elements that the specification prescribed not to be used.
- Missing element (M): a mandatory element is missing.
- Format (F): an element is ill-formatted. This may be based on purely syntactic properties (e.g. not UTC format), semantic properties (date specifies Feb 29th in a non-leap-year), parsing ambiguities, type ambiguities, etc.).
- Version (V): an element has not the version prescribed.
- Consistency (C): an element is not consistent with other elements (e.g. $URI_C \neq wtrealm$) or the overall context of the protocol run (e.g. metadata, state).

Principal We distinguish between the view of the identity consumer C and the identity supplier S.

5 Defining Test Cases

To specify all test cases individually would blow up this paper with a lot of details not of specific interest. Thus, we propose a more powerful specification method. We take an abstract message specified in the formal protocol definition as template for test case messages. We only specify differences to such a correct message. We use a pseudocode language to manipulate and describe such test case messages. We design this language similar to common object-oriented languages to keep a low entry barrier for adoption in real products. We describe the language in the following and depict its structure in Figure 2

- **class** Message - is an abstract message used to describe template as well as test case messages. As attribute it contains a set of Element instances. It has the following methods for manipulation.
  - Message ($template$: Message) - constructs an exact copy of the $template$ but with independent state (a *clone* in object-oriented terms) i.e. manipulating the elements of the result message will not influence the elements of the template.
  - add (e : Element) - adds another Element to the message; the method does not overwrite existing elements. We assume that the order of the elements in the message is not important.
  - remove (e : Element) - removes a Element from the message.

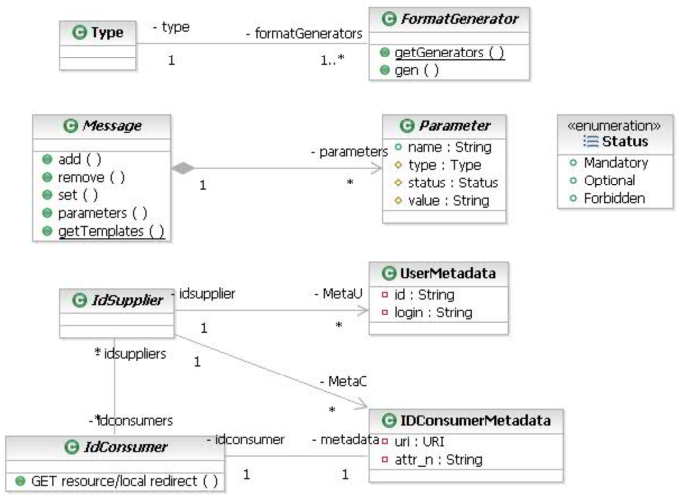

Figure 2: Structure of the test case definition classes.

  - set (e : Element) - replaces the first element p' of the Message with $name = e$.getName() with Element e.
  - set (n : String, v : String) - changes the first element with $name = n$ to $value := v$.
  - elements () - iterates all Element instances stored by the Message.
  - **static** getTemplates ($step$: String) - constructs a set of correct Message instances for a certain protocol $step$. The Message instances use different combinations of optional elements and are supposed to be used as templates for test case messages.

- **class** Element - is an abstract message part with a certain Type. It may be manipulated on its own. A Element instance has the attributes
  - $name$: String
  - $type$: Type, where Type denotes a well-defined set of types (e.g. String, UTC, URI, etc.).
  - $status$: {M, O, F }, where M means mandatory in WSFPI, O optional, F forbidden.
  - $value$: String

- **class** FormatGenerator - We assume that for each Type, there exists a set of generators that generate well-formed as well as malformed strings for that Type. We have the following methods:
  - **static** getGenerators ($type$: Type) - returns a set of all FormatGenerator instances known for a certain Type.
  - **static** getGenerators ($type$: Type, $correct$: Boolean, $exceptions$: Set) - returns a subset of the FormatGenerator instances known for a certain Type, where $correct$ specifies whether the generators are supposed to produce a correct or incorrect format for the Type. The Set $exceptions$ names values that are excluded from the generation.
  - gen (e : Element) - executes the FormatGenerator's generation algorithm. The method generates a Element instance that has a specific format. The method takes the argument e as template and changes it according to the generator's specification.

```
            Browser        Identity supplier $S$
5a-c                       Establishment of secure HTTPS channel $ch$ $\boxed{S.1}$
5d                         if $(a, path) \neq URI_S$ then abort; $\boxed{S.2}$
                           $(wa, wtrealm, wreply, wctx, wct) \leftarrow query$;
                           if this fails $\boxed{S.3}$ or $wa \neq$ wsignin1.0 $\boxed{S.4}$
                             or $eC := MetaC[URI = wtrealm] = \perp$ $\boxed{S.5}$
                             or $wreply \neq \perp \wedge wreply \not\subseteq wtrealm$ $\boxed{S.6}$
                             or $wct \neq \perp$
                           then abort

5.1, 6a                    ... User authentication ...
6b                         if $wreply = \perp$ then $a' := wtrealm$ else $a' := wreply$;
                           if $a'$ is not https then abort; $\boxed{S.7}$
                           Send POSTForm over established channel $ch$
```

Figure 3: Security checks of the identity supplier (Steps 5 and 6). We omit the details of channel establishment and user authentication.

We introduce a statement

$$\textbf{Forall } x \in Set \textbf{ do } statement$$

in order to iterate over arbitrary sets Set and have the $statement$ executed for each x in Set. The statement

$$\textbf{test}(m : \mathsf{Message}, expected\_result : \mathsf{Result})$$

executes a certain test case. The statement sends Message m to the machine to be tested and compares the machine's reaction with the $expected\_result$. Analogous to WSFPI, we specify abort as only Result for faulty protocol runs.

6 Testing the Identity Supplier

We define the test suite for an identity supplier in this section. We introduce the WSFPI definition [1] for the behavior of an identity supplier in Figure 3. Section 6.1 contains a description of the test setting for the identity supplier tests. The following subsections provide the test cases in plain text with a pair referencing the security check affected and the classification, e.g. (S.2, C) for a missing consistency check on URI_S.

We use Section 6.5 to elaborate the test case specification in full detail as an example. We there illustrate the principle of breaking-down the high-level test case description into pseudocode.

6.1 Test Setting

In this section, we consider the setting for the test of an identity supplier. We fix the meta data and message elements for a correct protocol run to generate a basis for the test cases.

According to Table 1 on page 3, identity supplier S knows the following metadata:

- The identity supplier S has a user database $MetaU$. For each user that identity supplier S knows this database contains an entry id, $login$
- The identity supplier has a consumer database $MetaC$. For each identity consumer C that has a relationship to S, the identity supplier has an entry in database $MetaC$: URI, att_n. The URI is supposed to equal the identity consumer's service address URI_C, whereas att_n specifies the attributes the identity consumer requires.

To test the identity supplier, the Step 5 message of WSFPI is the most important input. We consider a certain set of message elements:
$wa := \mathsf{Element}(name := \mathsf{wa}, type := \mathsf{String}, status := \mathsf{M}, value := \mathsf{wsignin1.0});$
$wtrealm := \mathsf{Element}(name := \mathsf{wtrealm}, type := \mathsf{URI}, status := \mathsf{M}, value := MetaC.URI);$
$wreply := \mathsf{Element}(name := \mathsf{wreply}, type := \mathsf{URI}, status := \mathsf{O});$
$wctx := \mathsf{Element}(name := \mathsf{wctx}, type := \mathsf{String}, status := \mathsf{O});$
$wct := \mathsf{Element}(name := \mathsf{wct}, type := \mathsf{UTC}, status := \mathsf{O});$
$wp := \mathsf{Element}(name := \mathsf{wp}, type := \mathsf{URLHostPath}, status := \mathsf{F});$
$wres := \mathsf{Element}(name := \mathsf{wres}, type := \mathsf{URLHostPath}, status := \mathsf{F});$
$wreq := \mathsf{Element}(name := \mathsf{wreq}, type := \mathsf{SecurityTokenRequest}, status := \mathsf{F});$
$wreqptr := \mathsf{Element}(name := \mathsf{wreqptr}, type := \mathsf{URLHostPath}, status := \mathsf{F});$

A correct Step 5 message only contains the optional and mandatory elements:
$m5 := \mathsf{Message}(Step5);$
$m5.\mathsf{add}(wa);$
$m5.\mathsf{add}(wtrealm);$
$m5.\mathsf{add}(wreply);$
$m5.\mathsf{add}(wctx);$
$m5.\mathsf{add}(wct).$

6.2 Setup

This section shortly introduces the testing of the protocol setup which is often neglected. Basically, these test cases require a protocol implementation to do consistency checks in each initialization phase.

[1, p. 6]

- Unspecified metadata (M)
- Metadata is ill-formatted at setup time (F).
- Pieces of metadata are inconsistent to each other (C)
- Uniqueness of the relevant metadata is not given (A)

6.3 Protocol Flow

- Step 5 channel is not HTTPS (S.1, M)
- Step 5 channel is HTTPS but wrong version or weak ciphersuite (S.1, V)
- Step 5 message did arrive at another URI than URI_S (S.2, C)

6.4 User Interaction

We omit test cases probing the user interaction of the protocol run as they are optional for the protocol's security.

6.5 Message

We use the class of message-specific test cases to illustrate the principles of choosing test cases and breaking down the high-level test case descriptions to pseudocode in the language of Section 5.

Choice of test cases We chose the test cases such that they cover all security checks of a message receiver and the consistency with other message parts or the general protocol state. We start with testing security checks performed on messages and message elements. These test cases basically check how the protocol engine reacts if message elements are added (A), missing (M), ill-formatted (F) or in the wrong version (V). In general we iterate over all combinations of message elements and these classes. The last five cases test the consistency checks with other message elements and the protocol state (C).

Deriving pseudo-code Let's consider the first test case as an example (wa contains wrong action or version). We first iterate over all possible good messages as template for the test case generation. The set of actions possible contains the correct action wsignin as well as actions specified in WS-Federation, yet forbidden by WSFPI (e.g. wpseudo) and a completely undefined action (undefined). Thus, we cover all main classes of actions. We do the same with the protocol version, whereby 1.0 is correct and 1.1 forbidden. Then we iterate over all possible combinations of actions and versions and modify the correct message template with the combination of action and version to generate a test message.

- wa element contains the wrong action or protocol version (S.4, V):

```
$M$ := Message.getTemplates(Step5);
$Actions$ := {wsignin, wsignoutcleanup, wattr, wpseudo, undefined};
$Versions$ := {1.0, 1.1};
**Forall** $m \in M$ **do** {
  **Forall** $a \in Actions$ **do** {
    **Forall** $v \in Versions$ **do** {
      $m^*$ := **new** Message($m$);
      $m^*$.set(wa, $a + v$);
      **test**($m^*$, abort);
    }
  }
}
```

- The Step 5 message contains elements allowed by WS Federation Passive but not by WSFPI (S.3, A):

```
$M$ := Message.getTemplates(Step5);
$WSFedParams$ := {$wp, wres, wreq, wreqptr$};
**Forall** $m \in M$ **do** {
  **Forall** $e \in WSFedParams$ **do** {
    $e$.fill($MetaC$);
    $m^*$ := **new** Message($m$);
    $m^*$.add($e$);
    **test**($m^*$, abort);
  }
}
```

- The Step 5 message contains elements that are not specified in WS Federation Passive Requestor Profile (S.3, A):

```
$M$ := Message.getTemplates(Step5);
**Forall** $m \in M$ **do** {
  $G$ := FormatGenerator.getGenerators(String, true, $ws_fed_params$);
  **Forall** $gen \in G$ **do** {
    $e$ := gen($m$);
    $m^*$ := **new** Message($m$);
    $m^*$.add($e$);
    **test**($m^*$, abort);
  }
}
```

- The step 5 HTTP message contains elements allowed twice (S.3, A).

```
$M$ := Message.getTemplates(Step5);
**Forall** $m \in M$ **do** {
  $E$ := $m$.elements();
  **Forall** $e \in E$ **do** {
    $e$.fill($MetaC$);
    $m^*$ := **new** Message($m$);
    $m^*$.add($e$);
    **test**($m^*$, abort);
  }
}
```

- A mandatory element of the step 5 message is missing (S.3, M).

```
$M$ := Message.getTemplates(Step5);
**Forall** $m \in M$ **do** {
  $E$ := $m$.elements();
  **Forall** $e \in E$ **do** {
    $m^*$ := **new** Message($m$);
    $m^*$.remove($e$);
    **test**($m^*$, abort);
  }
}
```

- Elements of the step 5 HTTP message have the wrong format (S.3, F).

```
$M$ := Message.getTemplates(Step5);
**Forall** $m \in M$ **do** {
  $E$ := $m5$.elements();
  **Forall** $e \in E$ **do** {
    $type$ := $e$.getType();
    $G$ := FormatGenerator.getGenerators($type$);
    **Forall** $gen \in G$ **do** {
      $p^*$ := $gen(e)$;
      $m^*$ := **new** Message($m$);
      $m^*$.set($p^*$);
      **test**($m^*$, abort);
    }
  }
}
```

- Generate two protocol runs of WSFPI. Exchange the $wctx$ element from the Step 5 messages of both protocol runs. (C).

```
$m_1$ := WSFPI.simulate(Step5);
$m_2$ := WSFPI.simulate(Step5);
```

```
$wctx_2$ := $m_2$.get(wctx);
$m_1^*$ := $m_1$.set($wctx_2$);
test($m_1^*$, abort);
```

- *wtrealm* that is not known to S (S.5, C).

```
$M$ := Message.getTemplates(Step5);
Forall $m \in M$ do {
   $type$ := $wtrealm$.getType();
   $G$ := FormatGenerator.getGenerators($type$, true, $MetaC.URI$);
   Forall $gen \in G$ do {
     $wtrealm^*$ := $gen(wtrealm)$;
     $m^*$ := new Message($m$);
     $m^*$.set($wtrealm^*$);
     test($m^*$, abort);
   }
}
```

- *wtrealm* that is not HTTPS (S.7, C).

```
$M$ := Message.getTemplates(Step5);
Forall $m \in M$ do {
   $wtrealm^*$ := $wtrealm$;
   $wtrealm^*$.substituteValue("https://", "http://");
   $m^*$ := new Message($m$);
   $m^*$.set($wtrealm^*$);
   test($m^*$, abort);
}
```

- *wreply* is present and not under *wtrealm* (S.6, C).

```
$M$ := Message.getTemplates(Step5);
Forall $m \in M$ do {
   $type$ := $wreply$.getType();
   $G$ := FormatGenerator.getGenerators($type$, true, $MetaC.URI$);
   Forall $gen \in G$ do {
     $wreply^*$ := $gen(wreply)$;
     $m^*$ := new Message($m$);
     $m^*$.set($wreply^*$);
     test($m^*$, abort);
   }
}
```

- *wreply* that is present but not HTTPS (S.7, C).

```
$M$ := Message.getTemplates(Step5);
Forall $m \in M$ do {
   $wreply^*$ := $wreply$;
   $wreply^*$.substituteValue("https://", "http://");
   $m^*$ := new Message($m$);
   $m^*$.set($wreply^*$);
   test($m^*$, abort);
}
```

```
        Browser       Identity consumer $C$
7a-c                  Establishment of secure HTTPS channel $ch$ $\boxed{C.1}$
7d                    $(wa, wresult, wctx) \leftarrow response$;
                      **if** this fails $\boxed{C.2}$ **or** $wa \neq$ wsignin1.0 **then abort**; $\boxed{C.3}$
                      $(name, m) \leftarrow \mathsf{test}(wresult)$;
                      **if** $name \neq nameS$ **then abort**; $\boxed{C.4}$
                      $(issuer, audience, idu, att) \leftarrow m$;
                      **if** this fails **then abort**; $\boxed{C.5}$
                      **if** $issuer \neq URIS$ **then abort**; $\boxed{C.6}$
                      **if** $audience \neq URI_C$ **then abort**; $\boxed{C.7}$
7e                    **output** $(\mathsf{accepted}, cid_{bc}, idu, att)$.
```

Figure 4: Security checks of the identity consumer (Step 7). Again we omit the channel establishment.

7 Testing the Identity Consumer

In this section, we derive the test cases for an identity consumer. We consider the setup of the identity consumer as well as the security checks for Step 7, which is most important for the security of WSFPI. We define the behavior of a correct identity consumer in Figure 4 and name the security checks required by WSFPI.

7.1 Test Setting

We fix metadata and message elements to get a test setting for an identity consumer. We use Table 1 on page 3 to derive the metadata an identity consumer knows according to WSFPI:

- The identity consumer knows the certificate $certS$ of its identity supplier and can derive $nameS$, under which its identity supplier signs security tokens for WSFPI.
- The identity consumer holds the WSFPI protocol handler URL of its identity supplier $URIS$.
- Identity consumer C has a channel identifier ch_id, which it uses to establish secure channels.

We define a Step 7 message of WSFPI as most important template for the identity consumer test cases:
$wa := \mathsf{Element}(name := \mathsf{wa}, type := \mathsf{String}, status := \mathsf{M}, value := \mathsf{wsignin1.0})$;
$wresult := \mathsf{Element}(name := \mathsf{wresult}, type := \mathsf{RequestSecurityTokenResponse}, status := \mathsf{M})$;
$wctx := \mathsf{Element}(name := \mathsf{wctx}, type := \mathsf{String}, status := \mathsf{O})$;
$wresultptr := \mathsf{Element}(name := \mathsf{wresultptr}, type := \mathsf{URI}, status := \mathsf{F})$;
$wreply := \mathsf{Element}(name := \mathsf{wreply}, type := \mathsf{URI}, status := \mathsf{F})$;
$wct := \mathsf{Element}(name := \mathsf{wct}, type := \mathsf{UTC}, status := \mathsf{F})$;
$wp := \mathsf{Element}(name := \mathsf{wp}, type := \mathsf{URLHostPath}, status := \mathsf{F})$;
$wres := \mathsf{Element}(name := \mathsf{wres}, type := \mathsf{URLHostPath}, status := \mathsf{F})$;
$wreq := \mathsf{Element}(name := \mathsf{wreq}, type := \mathsf{SecurityTokenRequest}, status := \mathsf{F})$;
$wreqptr := \mathsf{Element}(name := \mathsf{wreqptr}, type := \mathsf{URLHostPath}, status := \mathsf{F})$;

A correct Step 7 message only contains the optional and mandatory elements:
$m7 := \mathsf{Message}(Step5)$;
$m7.\mathsf{add}(wa)$;
$m7.\mathsf{add}(wresult)$;

$m7.\mathsf{add}(wctx)$.

The $wresult$ element itself is a complex construct. It is a RequestSecurityTokenResponse wrapping a SAMLToken. This SAML token needs to have a well-defined structure, which is introduced in [3]. The protocol definition of [1] as well as Figure 4 treat the parsing and security analysis of the token only in the statement

$$(wa, wresult, wctx) \leftarrow response$$

. We abstract from the concrete analysis of the token, but stress that its parsing and comparison to the specifications of [3] is not trivial. Thus, the checks of assumption $\boxed{C.2}$ require careful elaboration.

7.2 Setup

[1, p. 6]

- Unspecified metadata (M)
- Metadata is ill-formatted at setup time (F).
- Pieces of metadata are inconsistent to each other (C)
- Uniqueness of the relevant metadata is not given (A)

7.3 Protocol Flow

- Step 7 channel is not HTTPS ($\boxed{C.1}$, M)
- Step 7 channel is HTTPS but wrong version or weak ciphersuite ($\boxed{C.1}$, V)
- Step 7 message with the same token arrives twice (A)
- Step 7 message arrives without a request being originally issued by the identity consumer (M)
- Step 7 message did not arrive at URI_C (C) Behavior of the service mapping, of error handling?
- Step 7 message did arrive outside the hierarchy under URI_C) (C)

7.4 Message

- Step 7 message contains manipulated $wctx$ element (F)
- Elements of Step 7 message are ill-formatted ($\boxed{C.2}$, F)
- Step 7 message contains other elements than prescribed ($\boxed{C.2}$, A)
- Step 7 message contains a $wresultptr$ ($\boxed{C.2}$, A)
- Mandated elements are missing in Step 7 message ($\boxed{C.2}$, M)
- Elements in the URL query string and the POST body of Step 7 collide ($\boxed{C.2}$, C)
- The $wtrealm$ element is not equal URI_C ($\boxed{C.2}$, C)
- Data in the message does not match the metadata ($\boxed{C.3}$,C)
- wa element of Step 7 has the wrong version or action ($\boxed{C.3}$, V)

7.5 Token

- The token is not signed under $name_S$ ($\boxed{C.4}$, C)
- the signature is missing ($\boxed{C.4}$, M)
- the signature is present but invalid ($\boxed{C.4}$, F)
- The trust chain for the signature is broken ($\boxed{C.4}$, C)
- The RequestTokenWrapper is not correctly formatted ($\boxed{C.5}$, F)
- The SAML token is not correctly formatted ($\boxed{C.5}$, F)
- The token is invalid considered its validity time ($\boxed{C.5}$, F)
- Parts of the token are not well formatted ($\boxed{C.5}$, F)
- The token contains additional parts than mandated (e.g. Statements) ($\boxed{C.5}$, A)
- Mandated parts are missing ($\boxed{C.5}$, M)
- The naming of different token parts does not match ($\boxed{C.5}$, C)
- The token has the wrong SAML version ($\boxed{C.5}$, V)
- The token issuer is unspecified ($\boxed{C.6}$, M)
- The token issuer is unequal to $name_S$ / URI_S ($\boxed{C.6}$, C)
- The token $audience$ does not match the $wtrealm$. ($\boxed{C.7}$, C)
- The token $audience$ does not match the $AppliesTo$. ($\boxed{C.7}$, C)
- The token $audience$ is missing ($\boxed{C.7}$, M)
- The token contains multiple $audience$ parts ($\boxed{C.7}$, A)
- The $audience$ specifies more than one token consumer ($\boxed{C.7}$, A)
- Send two tokens with colliding token identifiers. (C)

8 Conclusion

Firstly, we observe that building up different sets of generators reduces the complexity of realizing a test case suite for WSFPI and other identity federation protocols a lot. Such generators may address correct messages as templates for test cases or message elements with anomalies to modify the templates.

Breaking down the test cases into pseudocode showed that the general structure of the test cases is very similar. We first iterate over all correct message templates, then we iterate over the set of message elements affected by the current test case. Finally, we iterate over the set of generators that produce anomalies suitable for the current test case. Therefore, one might use a meta test case generator that generated test cases according to this pattern. One might cover nearly all generic test cases by means of such a meta generator.

The approach of generating test cases automatically may be taken one step further by means of model-based architecture. One may model a testing profile for browser-based identity federation protocols or even the protocols themselves and deduce test cases to be generated from the model. One candidate for such a model-based testing approach is the UML Profile for Testing [2] which has an advanced concept of test cases, test subjects, expected and actual results. This profile is supported by the Hyades extension for Eclipse [7] and has the potential to also capture the whole testing approach (classification). Less advanced yet still a fine testing framework for Java is jUnit [6]. It supports a modular design of test cases and test case suites.

Acknowledgements

We thank Christopher Giblin for his valuable comments and the interesting discussions about the software engineering aspects of the test case suite.

References

[1] Thomas Groß and Birgit Pfitzmann. Proving a WS-federation passive requestor profile, September 2004. Will be published in the Proceedings of the 2004 ACM Workshop on Secure Web Services.

[2] Object Management Group. The UML testing profile, April 2004. `http://www.fokus.gmd.de/research/cc/tip/projects/u2tp/`.

[3] Matt Hur, Ryan D. Johnson, Ari Medvinsky, Yordan Rouskov, Jeff Spellman, Shane Weeden, and Anthony Nadalin. Passive Requestor Federation Interop Scenario, Version 0.4, February 2004. `ftp://www6.software.ibm.com/software/developer/library/ws-fpscenario2.d%oc`.

[4] Chris Kaler and Anthony Nadalin (ed.). Web Services Federation Language (WS-Federation), Version 1.0, July 2003. BEA and IBM and Microsoft and RSA Security and VeriSign, `http://www-106.ibm.com/developerworks/webservices/library/ws-fed/`.

[5] Chris Kaler and Anthony Nadalin (ed.). WS-Federation: Passive Requestor Profile, Version 1.0, July 2003. BEA and IBM and Microsoft and RSA Security and VeriSign, `http://www-106.ibm.com/developerworks/library/ws-fedpass/`.

[6] Incorporated Object Mentor. JUnit, testing resources for extreme programming, August 2002. `http://www.junit.org/`.

[7] Eclipse Platform. Hyades automated software quality evaluation framework, December 2004. `http://www.eclipse.org/hyades/`.