An Authentication Flaw in Browser-based Single Sign-On Protocols: Impact and Remediations

Alessandro Armando[a,b], Roberto Carbone[b], Luca Compagna[c], Jorge Cuellar[d], Giancarlo Pellegrino[c,f], Alessandro Sorniotti[e]

[a]*AI-Lab, DIST U. di Genova, Viale Causa 13, 16145, Genova, Italy*
[b]*Security & Trust Unit, FBK-irst, Via Sommarive 18, 38050, Trento, Italy*
[c]*SAP Labs France SAS, 805 Av. du Dr M. Donat, 06254, Mougins, France*
[d]*Siemens AG, Corporate Technology, D-80200 Munich, Germany*
[e]*IBM Research Zurich, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland*
[f]*Institut Eurécom, 2229 Route des Cretes, BP 193, F-06560 Sophia-Antipolis Cedex, France*

Abstract

Browser-based Single Sign-On (SSO) protocols relieve the user from the burden of dealing with multiple credentials thereby improving the user experience and the security. In this paper we show that extreme care is required for specifying and implementing the prototypical browser-based SSO use case. We show that the main emerging SSO protocols, namely SAML SSO and OpenID, suffer from an authentication flaw that allows a malicious service provider to hijack a client authentication attempt or force the latter to access a resource without its consent or intention. This may have serious consequences, as evidenced by a Cross-Site Scripting attack that we have identified in the SAML-based SSO for Google Apps and in the SSO available in Novell Access Manager v.3.1. For instance, the attack allowed a malicious web server to impersonate a user on any Google application. We also describe solutions that can be used to mitigate and even solve the problem.

Keywords: Single Sign-On, Security Protocols, Model-checking, OpenID, SAML SSO, Vulnerability, Model-based Security Testing

1. Introduction

Browser-based Single Sign-On (SSO) is replacing conventional solutions based on multiple, domain-specific credentials by offering an improved user experience: clients perform a single log in operation to an identity provider, and are yet able to access resources offered by a variety of service providers. Moreover, by replacing multiple credentials (one per service provider) with a single one (associated

Email addresses: armando@{dist.unige.it,fbk.eu} (Alessandro Armando), carbone@fbk.eu (Roberto Carbone), luca.compagna@sap.com (Luca Compagna), jorge.cuellar@siemens.com (Jorge Cuellar), giancarlo.pellegrino@{sap.com, eurecom.fr} (Giancarlo Pellegrino), aso@zurich.ibm.com (Alessandro Sorniotti)

with the identity provider), SSO solutions are expected to improve the overall security as users tend to use weak passwords and/or to reuse the same password on different service providers.

At the core of a browser-based SSO solution lies a browser-based authentication protocol. Three roles take part in the protocol: a client (C), an identity provider (IdP) and a service provider (SP). The objective of C, typically a web browser guided by a user, is to get 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 decide on C's entitlement to the requested resource.

A number of solutions for browser-based SSO are available: the OASIS *Security Assertion Markup Language* (SAML) 2.0 (OASIS, 2007), Microsoft® Passport (Microsoft, 2011), the Liberty Alliance project (OASIS, 2004), the Shibboleth Initiative (Internet2, 2007), and OpenID (OpenID Foundation, 2007b) are the most popular. The Web Browser SSO Profile of SAML 2.0 (SAML SSO, for short) (OASIS, 2005b, Chapter 4) is the *de facto* standard in the business domain: it defines an XML-based format for encoding security assertions as well as a number of protocols and bindings that prescribe how assertions must be exchanged in a variety of applications and/or deployment scenarios. Prominent software companies, including IBM, Novell, Oracle, and SAP, base their SSO implementations on SAML SSO. Google has developed a SAML-based SSO service for its popular web applications (namely Gmail, Google Calendar, Talk, Docs and Sites), called the SAML-based SSO for Google Apps (Google, 2008).

OpenID is an SSO solution more suited to Web 2.0 applications (e.g. blogs, wikis) as it supports dynamic interactions between identity providers and service providers without requiring any configuration, prior metadata exchange, or administrative privileges for deployment.

The security of browser-based SSO protocols critically relies on a number of assumptions on the trustworthiness of the principals involved and on the security of the transport protocol used to exchange messages. As we will discuss in more detail later, IdP is trusted (and in particular it is not compromised). But we do not assume that all SPs which C may play the protocol with are trusted or uncompromised. In particular, we ask ourselves, if a malicious SP can compromise the security of sessions of the protocol played with other SPs. We also assume that the communications between C and SP, and also between C and IdP, are carried over unilateral SSL or TLS connections. It is worth noticing, and this fact will play a crucial role later, that the single messages between C and SP may be sent over different SSL/TLS connections, as far as the standard and the typical implementations are concerned. Trying to enforce that all such messages are always sent over the same SSL/TLS connection is quite difficult in practice, as we will discuss later on. The consequence will be that a malicious SP will indeed be able to affect the security of sessions of the SAML protocol of C with other SPs.

Thus, in this paper we show that the usual assumptions, and in particular the one concerning the nature of the communication channel between C and SP are not enough and leave the protocols vulnerable to an authentication flaw,

which is present in straight-forward implementations of the standard. We will also discuss further security measures to mitigate or fully avoid the risks.

We then show how this flaw can be practically exploited in a number of prominent SSO solutions based on SAML, including the SAML-based SSO for Google Apps, used by over one million business customers, and the SSO solution available in Novell Access Manager v.3.1. In particular, on the SAML-based SSO for Google Apps, a malicious web server was able to impersonate a target user on any Google application. A similar attack was possible on deployments of Novell Access Manager v.3.1. Both vendors have been informed of the vulnerabilities in their products and both have promptly patched their implementations. All our findings have also been discussed with members of the OASIS Security Services Technical Committee and a SAML V2.0 Errata has been redacted and approved (OASIS, 2012).

We also show that the OpenID protocol too is affected by the same problem, and – for both SAML SSO and OpenID – we provide solutions that allow the authentication flaw and its exploitations to be mitigated or even completely eliminated. We have just informed the OpenID working group and some of the implementors of this solution about our recent findings on OpenID.

The findings reported in this paper have been found with the help of state-of-the-art techniques for the formal modeling and automated analysis of security protocols. Although the main goal of the paper is to present and discuss the findings themselves, details about the formal modelling and analysis that we performed are also given.

Structure of the paper. In the next section we introduce the SAML SSO profile for web-based authentication. In Section 3 we present the authentication flaw on the SAML SSO, and in Section 4 describe how it can be exploited on actual implementations. In Section 5 we provide a number of solutions to the problem. In Section 6 we discuss the effects of the same vulnerability in OpenID and show how similar countermeasures as those presented in Section 5.2 for SAML SSO may be used to protect OpenID implementations. In Section 7 we briefly present the formal modelling and the automatic analysis of SAML SSO that allowed us to discover the flaw. In Section 8 we discuss the related work and in Section 9 we draw some final remarks.

2. The SAML 2.0 Web Browser SSO Profile

SAML SSO provides a standardized, open, interoperable SSO solution applicable in a multitude of environments and situations, and can therefore be instantiated according to the specific requirements posed by the application scenario. In this paper we focus on one of its most widely used instantiations, the SP-Initiated SSO with Redirect/POST Bindings, whose typical use case is described in (OASIS, 2007). In the remainder of this paper we will refer to this use case as *the SAML SSO use case* and to the associated protocol as *the SAML SSO Protocol*.

In Figure 6 we capture the most important steps of the *SAML SSO Protocol*, abstracting away the steps that are irrelevant for our analysis, such as—among others—the IdP discovery phase. In step S1, C asks SP to provide the resource located at URI, say *Resource*(URI), without having a valid, active logon session (i.e. security context) with SP. SP then initiates the *SAML Authentication Protocol* by sending to C an HTTP redirect response (status code 302) to IdP, containing an authentication request AuthReq(ID,SP), where ID is a (pseudo-) randomly generated string uniquely identifying the request (steps A1 and A2). A frequent implementation choice is to use the `RelayState` field to carry the original URI that C has requested (see (OASIS, 2007)).

If C does not have an existing security context with IdP, then IdP challenges C to provide valid credentials. If the authentication succeeds, IdP creates the local security context, builds an authentication assertion as the tuple $AA = \text{AuthAssert}(\text{ID}, \text{C}, \text{IdP}, \text{SP})$, and places it in a response message $Resp = \text{Response}(\text{ID}, \text{SP}, \text{IdP}, \{AA\}_{\text{K}_{\text{IdP}}^{-1}})$, where $\{AA\}_{\text{K}_{\text{IdP}}^{-1}}$ is the assertion signed with $\text{K}_{\text{IdP}}^{-1}$, IdP's private key. IdP then places *Resp* and the value of `RelayState` received from SP into an HTML form (indicated as Form(...) in Figure 6) and sends the result back to C in an HTTP response (step A3) together with some script that automatically posts the form to SP (step A4). This completes the SAML Authentication Protocol. SP can then deliver the requested resource, *Resource*(URI), to C (step S2), and the SAML SSO Protocol completes as well.

Note that the steps at message S1 and S2 admittedly fall outside of the scope of the standard, and their implementation is left free. In this paper we capture steps S1 and S2 as described in *the SAML SSO use case*; a number of commercial SAML SSO solutions indeed adopt similar approaches to implement those steps.

As pointed out in (Armando et al., 2008) the security of the protocol critically relies on (unstated) assumptions about the trustworthiness of the participants involved and about the transport protocols used to exchange the protocol messages; we shall review these in the next Sections.

2.1. Trust and Transport Protocol Assumptions

The above protocols work under the assumption that *(i)* IdP is not compromised, i.e. it is not under the control of an intruder and it abides by the rules of the protocol and *(ii)* IdP is trusted by SP to generate authentication assertions about C. Even if they are not explicitly stated in the SAML 2.0 specifications, these are very reasonable assumptions to make and, in fact, both protocols are useless if IdP is not trusted to generate authentication assertions about C or if there is the doubt that IdP is compromised. However, *we do not assume that all SPs which C may play the protocol with are uncompromised*. In other words, unlike (Hansen et al., 2005), we want to consider also those situations in which C runs the protocol with compromised SPs in order to determine whether they affect the security of sessions of the protocol played with other uncompromised SPs. This is very important as SPs are usually managed by different organizations that do not always share trust relationships.

The SAML 2.0 specifications repeatedly state the following assumptions of the transport protocols used to carry the protocol messages:

(TP1) Communication between C and SP is carried over a unilateral SSL 3.0 or TLS 1.0 connection (henceforth called SSL), established through the exchange of a valid certificate (from SP to C).

(TP2) Communication between C and IdP is carried over a unilateral SSL connection that becomes bilateral once C authenticates itself on IdP. This is established through the exchange of a valid certificate (from IdP to C) and of valid credentials (from C to IdP).

2.2. Security Requirements

The SAML specifications do not explicitly state the security properties that the SAML SSO Protocol and the SAML Authentication Protocol are expected to achieve. By comparison with classic web authentication schemes, it is however natural to expect that at the end of the *SAML SSO Protocol*, the following security property is fulfilled:

(P1) SP and C mutually authenticate and agree on the value URI

As pointed out in (Lowe, 1997), different definitions of authentication are possible. The notion of authentication we consider in this paper includes *recentness*, i.e. the fact that the principal being authenticated recently took part in the protocol run so as to exclude replay attacks.

We note that the SAML Authentication Protocol, the building block of the SAML SSO Protocol, is only able to guarantee the property

(P2) SP authenticates C

The converse is not true, i.e., the SAML Authentication Protocol does not provide to C any guarantee on SP's identity; indeed in message A1, SP may instruct IdP to force C to redirect message A4 to an arbitrary location. Even the use of SSL certificates only guarantees that there is no man-in-the-middle in the communications between C and the recipient of message A4.

In the remainder of this paper, we will investigate whether the SAML SSO Protocol, constructed with a building block that only guarantees *(P2)*, is able to fulfill the original property *(P1)*, and we will show that the fulfillment of this property is not automatically guaranteed; in particular depending on the implementation choices, a malicious SP may be able to hijack C's authentication attempt and force the latter to access a resource without its consent or intention.

3. An Authentication Flaw in the SAML SSO Protocol

An analysis of the SAML specifications reveals that the standard does not specify whether the messages exchanged at steps S1 and A4 must be transported

over the same SSL connection or whether two different SSL connections can be used for this purpose. In other words, there is a certain degree of ambiguity on how assumption *(TP1)* of Section 2 can be interpreted.

The reuse of the SSL connection established at step S1 to also transport the message at step A4 is at first sight the most natural option. However this is difficult to achieve in practice for a number of reasons:

Resuming SSL connections. The use of a single SSL connections for the exchange of different messages cannot be guaranteed as, e.g., the underlying TCP connection might be terminated (e.g. timeout, explicitly by one of the end points), an SSL server could not resume a previously established session, or a client might be using a browser that very frequently renegotiates its SSL connection.[1]

Software modularity. Nowadays, software is designed to be increasingly modularized, capitalizing on layering and separation of concerns. This may result in the fact that—within SP implementations—the software module that handles SAML messages has no access to the internal information of the transport module that handles SSL. Thus, the information on whether the client has used a single SSL connection or two different ones may not be available.

Distributed SPs. The SAML SP may be distributed over multiple machines, for instance, for work-balancing reasons. This results in physically different SSL endpoints, with the inherent impossibility of enforcing a single session for all communications between SP and C.

We have extended the formal model discussed in (Armando et al., 2008) to faithfully capture the SAML SSO use case in which the messages of steps S1 and A4 can be transported over different SSL connections and fed it to SATMC (Armando et al., 2009), a state-of-the-art model-checker for security protocols. (See Sections 7 and 8 for more details.) The model checker detected the attack depicted in Figure 7, thereby witnessing a violation of property *(P1)* in the *SAML SSO Protocol*.

The attack involves four principals: a client (c), an honest IdP (idp), an honest SP (sp) and a malicious SP (i). The attack is carried out as follows: c initiates the protocol by requesting a resource $\mathrm{uri_i}$ at SP i. Now i, pretending to be c, requests a different resource uri at sp and sp reacts according to the standard by generating an Authentication Request, which is then returned to i. Now i maliciously replies to c by sending an HTTP redirect response to idp containing $\mathrm{AuthReq(id, sp)}$ and uri (instead of $\mathrm{AuthReq(id_i, i)}$, and $\mathrm{uri_i}$ as the standard would mandate). The remaining steps proceed according to the standard. The attack makes c consume a resource from sp, while c originally asked for a resource from i.

Note that the attack is possible essentially because the client—usually a normal browser with no knowledge of the SAML protocol—has no means of

[1] See, for instance, `http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp?topic=/com.ibm.itame2.doc_5.1/am51_webseal_guide54.htm`

verifying whether the authentication request and the authentication assertion are related to the initial request.

Interestingly enough, standard username/password authentication mechanisms do not suffer from this authentication flaw. To see this, let us assume that c has no active sessions with SPs i and sp; let us also assume that c's usernames and passwords are different for each SP.[2] Then under no circumstance can i hijack c's authentication attempt and unawarely and automatically force it to consume a protected resource at sp. From this point of view, the advantage of domain-specific credentials in the control of the user is that the user knows exactly for whom the credentials are intended upon providing them. With SSO, "binding" the views of the user and of the service provider is not so easy.

Note that the attack would be prevented if sp could enforce that the initial request and authentication response are carried over the same secure channel, but we have previously explained why this requirement is very difficult to achieve in practice. Note also that requiring digitally signed authentication requests would not fix the vulnerability; indeed the authentication request is actually generated by sp, and only blindly forwarded to c by the attacker; the signature is therefore valid and will be accepted.

Even more interestingly, the attack does not strictly require a malicious SP in order to be successful. Any malicious web server i would be able, upon a request from c, to mount the attack provided that *(i)* c is a client of sp and *(ii)* c has an active authentication context with idp.

The attack in Figure 7 can be exploited in a number of ways:

Delivery of an unrequested resource. The most trivial exploitation of the flaw consists in the attacker forcing the client to receive a different protected resource from the initially requested one. The same exploitation may also be mounted if a malicious web server redirects the browser to a legitimate SP before the SAML SSO Protocol starts. However this attack can be prevented by using well-known browser-side plugins that restrict HTTP redirections (e.g., the NoRedirect addon for Firefox). By allowing only IdP-to-SP and SP-to-IdP redirections, the delivery of an unrequested resource upon redirection outside of the SAML SSO Protocol is prevented, but a malicious SP can still mount the one depicted in the Figure 7.

Launching pad for cross-Site Request Forgery (CSRF) attacks. This attack assumes that the URI that was initially requested did not point to a resource, but rather contained a URL-encoded command, such as a request for the change of some settings or user's preferences, for the deletion of some resource or for the annulment of/committing to an action, such as the purchase of a paid good. Depending on the output provided by the execution of the command, the client may or may not be able to detect the attack. This type of attack is even more pernicious than classic CSRF, because CSRF requires C

[2] If this assumption does not hold, c is vulnerable to a number of other trivial attacks anyway.

to have an active session with SP, whereas in this case, the session is created automatically hijacking C's authentication attempt.

Launching pad for cross-Site Scripting (XSS) attacks. It is straightforward to see that this attack also constitutes a launching pad to reflected XSS attacks, i.e. XSS attacks that can be triggered by visiting a maliciously-crafted URL. In addition, a vanilla implementation of the SAML SSO protocol exposes the `RelayState` field to a possible injection of malicious code that may be executed at the honest SP side. Although the SAML standard recommends to protect the integrity of this field, our experience shows that this often is not the case (see Section 4).

In addition, unlike normal XSS attacks, where the attacker has to rely on social engineering (phishing, spam and so forth) to lure a victim into clicking on a malicious link, an exploitation of the vulnerability paves the way for systematically luring victims into visiting URIs that may be vulnerable to cross-site scripting attacks. Note also that in this case, unlike in the previous exploitations, the client is not suspicious about receiving a different resource than the one requested. On the contrary, with reference to Figure 7, because arbitrary code can be embedded in uri, a redirection to uri_i, the page that c initially requested, can be eventually forced at the end of the attack. As an example, if uri is forged as `javascript:window.open('`uri_i`'+document.cookie)` the client would be victim of the theft of its cookies for the domain sp through a visit to the requested uri_i.

Although in this paper we focus on the SP-initiated SSO protocol, it is worth mentioning that IdP-Initiated flows may suffer from *login CSRF* attacks (Barth et al., 2008), whereby the attacker forges a cross-site request to the login form and logs the victim into an honest web site *as the attacker*.

4. Exploitations in Actual SSO Implementations

An interesting question that we also address in this paper is whether exploitations of the abstract weakness of the standard are possible in actual implementations of the SAML SSO Protocol. To this end, we have analysed a number of SAML-based SSO solutions available on the market, including the SAML-based SSO for Google Apps, a deployment of Novell Access Manager v.3.1, and SimpleSAMLphp as deployed for Foodle (`https://foodl.org`). More details are reported in Sections 4.1, 4.2, and 4.3, respectively. All these deployments support the SAML SSO use case. As expected, by inspecting the messages exchanged between the parties we verified that SPs accept and process SAML responses carried over SSL connection different from that used to deliver the SAML request.

4.1. The SAML-based SSO for Google Apps

The protocol implemented in the SAML-based SSO for Google Apps in operation until 2009 is depicted in Figure 8. The model has been obtained by

carefully inspecting the reference implementation of SAML-based Single Sign-On for Google Apps[3] and by experimenting with the online service.

In the implementation offered by Google, when SP receives a request for a resource URI from C, if the request is accompanied by a valid session cookie, then the resource is returned right away (step S10 in Figure 8). The name of the session cookie depends on the specific service considered, e.g. it is named `CALH` in case of Google Calendar, and `GXAS` in case of Gmail. If the request is accompanied by valid values for the parameters `auth` and `husr` in the URI, then SP creates a fresh session cookie and sends it back to C; C then is asked to resubmit the request by means of an HTTP redirect (steps S8 and S9). If neither of the above conditions hold, C is redirected to the Service Login (SL) and the requested URI is passed as the value of the `continue` URL-encoded parameter. Upon receipt of this request, SL initiates the SAML Authentication Protocol (step A1) using SL`?continue=`URI as the value of the aforementioned `RelayState` field. If the SAML Authentication Protocol completes successfully, then SL sets the cookies `HID`, `HUSR`, and `ASIDAS` and returns an HTML page of the form (concisely indicated as Script(...) in step S4 of Figure 8):

```
<html>
 ...
 <body>
  <script>
   var url=URI
   ...
   window.setTimeout(
         function() {
            window.location = url;
         },
         0);
  </script>
  ...
 </body>
</html>
```

This simulates a redirection by setting the value of the browser variable `window.location` to URI and forcing the browser to reload the page. Notice that since the value of URI is embedded into the HTML page, it will be evaluated by the JavaScript interpreter. As already mentioned in Section 3, this means that if URI contains some malicious code, e.g.

```
javascript:window.open(
    'http://i/collect.php?cookies=' + document.cookie
)
```

then the cookies of C for the SP domain are sent to a web server under control of the intruder.

[3] `http://code.google.com/googleapps/domain/sso/saml_reference_implementation_web.html`

Our analysis of the SAML-based SSO for Google Apps shows that by exploiting the weakness of the standard, a compromized SP can force C to consume a resource from Google, e.g., by visiting any page of the gmail service. This trivial attack is however easily detected by the user using C, and does not bring any real advantage to the attacker. Definitely more serious was the XSS attack we were able to execute and that allowed the compromised SP to steal the cookies of C for the Google domain and thus to impersonate C on any Google application. The abstract flaw of Figure 7 served indeed as launching pad for this XSS. The attack is depicted in Figure 9. As we can see in the figure, c requires a resource from a compromised SP i; i, acting in turn as a client, receives from sp an Authentication Request, and passes it back to c, with the malicious code injected into the `RelayState`. The client's browser eventually executes the redirection to the maliciously-crafted URI, as if coming from the Google domain (thus circumventing the same origin policy). This redirection leads to the theft of the `HID`, `HUSR`, and `ASIDAS` cookies by sp. The malicious SP i can then use these cookies to obtain a valid session cookie from SL and unrestricted access to the Google Apps under c's name. In other words, the combination of the abstract flaw and the missing sanitization was key to this XSS attack. In response to our vulnerability report Google patched the issue by properly sanitizing the `RelayState` value. An acknowledgement of our contribution can be found in the Google corporate web pages (Google, 2009).

4.2. Novell Access Manager

We have also analysed the SAML SSO solution of the Novell Access Manager v.3.1 as deployed in a real industrial environment and even in this case we detected the authentication flaw. We have been able to mount a XSS attack similar to the one found in the Google SSO solution. In this deployment `RelayState` is not used to store the URI; instead, a URL-encoded parameter is used to this end and also in this case, the parameter was not sanitized. In response to our findings Novell promptly patched their implementation and issued a vulnerability report (Novell, 2011).

4.3. SimpleSAMLphp

The SimpleSAMLphp, as deployed in Foodle, stores the initially requested URI into the URL parameter `ReturnTo`. Although that field is not sanitized, we have not been able to mount any XSS. The reason is that SPs running SimpleSAMLphp additionally use cookies that block the abstract flaw we discovered. More details on this solution will be given in the next section.

Also in this case we promptly informed the developer and maintainer of the SSO solution, namely UNINETT. UNINETT credited us in the release notes of a new version of SimpleSAMLphp (UNINETT, 2010).

5. Fixing the vulnerability

The root of the authentication flaw presented in Section 3 lies in the following two main factors:

1. Clients are not able to link the Authentication Request they receive from SP in step A1 with their initial requests for a resource issued in step S1;
2. SP is not able to ensure that the messages exchanged with C (cf. steps S1, S2, A1, and A4) are carried over the same channel.

We have verified that – could one of the two causes be removed – the vulnerability would no longer be exploitable. In what follows, we shall outline a number of mitigations and subsequently of possible solutions, highlighting their strengths and shortcomings.

5.1. Mitigations

This section describes a number of controls – specified by the standard – that mitigate the impact of possible exploitations of the flaw, but that do not tackle its root causes.

The OASIS SAML Guidance for Implementations. We have thoroughly discussed our findings and possible solutions with members of the working group of the OASIS Security Services Technical Committee. The erratum "E90: RelayState sanitization", which has been included in the newly released errata to the SAML SSO v2.0 (OASIS, 2012), specifically addresses the security concerns we raised. The erratum summarizes the changes to (OASIS, 2005a) and (OASIS, 2005b) that have been eventually decided by OASIS. In a nutshell, the main points are:

- When using RelayState, implementations must carefully sanitize the URL schemes they permit, disallowing unencoded characters; and
- SPs should have a means of disabling the acceptance of unsolicited responses.

The first measure limits the damage an attacker can do upon a successful exploitation of the flaw; the second attempts to restrict the attack surface by refusing to accept unsolicited responses. It is worth pointing out that neither measures address the root cause of the vulnerability.

Signing the Authentication Request. Signing Authentication Requests limits the ability of an attacker to alter the content of sensitive fields in the message, such as the RelayState, whose modification opens up to XSS attacks as we have seen in the Section 4. Chapter *E1: RelayState for HTTP Redirect* of the SAML V2.0 Approved Errata (OASIS, 2012) clarifies how Authentication Requests (and sensitive fields within, such as RelayState) are to be signed. However, this security control alone is not sufficient to block all XSS attacks. Indeed an attacker may well discover a URL whose parameters are susceptible to injection of malicious code, request said URL and feed the signed Authentication Request to its victim.

In addition, signatures may even be removed from an Authentication Request: as an example, the HTTP redirection binding mandates the signature to

be stored as a URL encoded parameter and appended to the URL containing the request. By removing said parameter, the attacker is still able to modify the content of the RelayState and may thus trigger a successful XSS attack. Such attack vector is blocked only if IdP refuses to accept requests that are not signed.

5.2. Solutions

In this section we shall highlight measures that tackle the root cause of the presented flaw, in increasing order of effectiveness. The introduced measures may be deployed separately or may be combined to achieve an improved effectiveness. We emphasize however that the *SAML SSO Protocol* alone does not mandate the implementation of any of these solutions, thus leaving a vanilla implementation in principle flawed. The challenge is to fix the vulnerability with minimal changes so that existing solutions can be secured without radical modifications to the software components (e.g. SAML ECP profile) or to the standard.

Enforcing a single session. As pointed in Section 3, there is no guarantee that messages associated with steps S1, S2, A1, A4 are carried out over a single SSL connection. A different way to achieve this objective could be for SP to make sure that it is interacting with the same C throughout all these messages. SP may use IP addresses to identify clients and bind together the protocol messages. However, IP addresses can be spoofed by malicious principals, and it is a good and well-established security practice to not base any security decision on fields whose integrity cannot be satisfactorily verified.

Feedback from the user. As seen in the previous sections, the user may initiate the SAML SSO profile, authenticating to an SP without actually having explicitly requested anything from it. This could be avoided if IdP informed the user about the attempt to access URI on SP during the authentication and asked for an explicit consent before issuing the authentication assertion to SP. In this way, the user may realise that the authentication was going to be sent to a different SP than expected and may be given the possibility to stop the protocol. This solution however has a number of drawbacks: first of all, it forces a security decision upon a (possibly technically unaware) user, who is asked to tell apart legitimate SP-to-SP redirections from malicious ones. In addition, it breaks the seamlessness of SSO, in which the authentication process is supposed to be carried out with minimal interactions with the user.

Cookies. A standard way of enforcing bindings on connections is implemented using session cookies. With reference to Figure 6, by setting a session cookie in step A1 and expecting to receive it back on message A4, SP could check that the communication has occurred with the same C. This solution is adopted by a number of implementations of the standard, for instance by SimpleSAMLphp. An alternative approach that is suggested as a best practice, is to set the target URI of step S1 as the value of a cookie that the SP sets in step A1. The name

of said cookie is the hash of its value (i.e. of URI), and the cookie name is in turn set as the content of the RelayState. Originally this method was suggested to avoid revealing confidential information about the requested URI to IdP. A side-effect of the adoption of such measure is that responses that come in HTTP messages that do not contain a cookie *(i)* whose name is the same as specified in the RelayState field, *(ii)* whose content is not a valid URI and *(iii)* whose content does not hash to the cookie name are automatically rejected.

Unfortunately, cookies do not represent a complete solution: indeed cookies are designed to be difficult to steal and it is not as hard to set them. For instance, cookies with the "Secure" flag set (which instructs the browser not to transmit them over unencrypted channels) can be set over unprotected connections: modern browsers (e.g. Firefox 9.0) allow this. In practice an attacker could circumvent the protection offered by cookies by *(i)* setting a cookie for the victim SP through injected JavaScript or HTML META tags; *(ii)* corrupting the proxy discovery phase setting up a rogue WPAD or DHCP server, thus becoming the user's proxy; *(iii)* performing ARP poisoning thus becoming the victim's default gateway. Admittedly such attacks require a relatively powerful attacker.

In order not to block the SAML SSO IdP-initiated profile (see Section 4.1.5 in (OASIS, 2005b)) – an SP ought to accept unsolicited Authentication Responses (i.e. responses that do not carry an InReturnTo field) even if no valid session cookie is presented. However, as we have seen above, unsolicited responses may not be desirable in security sensitive deployment scenarios.

Self-signed Client Certificates. A simple, yet effective way for the SP to ensure that it is interacting with the same client is to require the latter to present a self-signed certificate. This solution is particularly attractive since it requires no changes to the protocol and can be deployed over existing components, since modern browser can perform certificate generation quite easily. The solution goes as follows: during the first SSL connection (cf. steps A1 and A2 in Figure 6), C is asked to present the certificate. SP will then generate an Authentication Request whose ID field is set to be equal to $n \,||\, \texttt{HMAC}_{K\,||\,n}(\texttt{RSA modulus})$, where n is a nonce, K is a secret known only to SP, `RSA modulus` is the RSA modulus of the public key contained in the client's certificate for the SSL connection in steps S1-A1. HMAC is the well-known HMAC keyed hash function (Bellare et al., 1996) and $||$ denotes the concatenation. After this, SP deletes all state information and sends the Authentication Request to C. During the second SSL connection (cf. steps A4 and S2), C is again asked for the certificate and the same certificate will be delivered to SP. The standard requires the *InResponseTo* field of the Response message to contain the same value of the ID field of the Authentication Request message: therefore SP can parse such field as n' and H' and then check whether $H' = \texttt{HMAC}_{K\,||\,n'}(\texttt{RSA modulus'})$; `RSA modulus'` is the RSA modulus of the public key contained in the client's certificate for the SSL connection in steps A4-S2.

If the above equality is verified, SP has the assurance that the assertion contained in step A4 is delivered by a principal whose RSA modulus used in

steps A4-S2 is the same as the one used in steps S1-A1. Identity of RSA moduli across both SSL connections implies identity of the principal triggering both connections: indeed, if we exclude the possibility of stolen certificates/private keys,[4] having two certificates with the same modulus is widely considered to happen with negligible probability: RSA moduli are currently 1024 or 2048 bits integers so the chance of randomly picking the same modulus twice is negligible.[5] Forging a second certificate with the same modulus as the first one is considered hard as it would break RSA's security: the SSL handshake indeed forces the user to prove possession of the private key associated to the advertised modulus, and generating such private key requires solving the RSA problem on input the initial modulus.

Notice that generating multiple certificates with the same modulus by exploiting the knowledge of its factorization is commonly discouraged as it may open up to the well-known common modulus attack (Schneier, 1996). Note also that the possibility of circumventing this security measure by performing replay attacks is not feasible because replayed assertions are not accepted by compliant implementations of the standard.

This security control effectively thwarts the attack presented in Figure 7: indeed c's self-generated certificate could not be the same as i's, and as a consequence, sp would reject message A4. On the client side, no difference with respect to the standard operations of SAML-based SSO are perceived, except the very first time in which – in case no client certificate is available – the client is instructed to create one. When the client gets the Authentication Assertion from IdP, an SSL connection with SP is opened. This connection may be the same or may be different. The client is again asked for a certificate and the browser will seamlessly present the same certificate as before. As pointed out in the previous section, extra care should be taken when this security control is used in conjunction with the SAML SSO IdP-initiated profile.

6. OpenID

OpenID Authentication 2.0 (hereafter OpenID) is an open and user-centric Web browser-based Single Sign-On protocol. It provides a way to authenticate a user C asking her to prove that she controls an identifier (OpenID Foundation, 2007a). OpenID is decentralized in the sense that it does not require SPs [6]

[4] Exporting a private key is considered a sensitive operation in most operating systems and browsers; consequently it presents warning to the users and requires explicit approval.

[5] If weak randomness generation techniques are used, two independent users may draw the same two random primes and therefore end up with two certificates with the same RSA modulus. This may happen in practice, as shown by (Lenstra et al., 2012): we therefore require users to be equipped with software libraries that implement secure random number generation routines.

[6] The OpenID Authentication 2.0 specification uses the term *Relying Parties* for the web applications that want a proof that the end user controls an identifier. However, in order to avoid confusion to the reader by introducing new terms, we will adhere to the SAML SSO terminology.

and OpenID IdPs to have pre-established relationships. This is actually one of the main differences with respect to SAML SSO where SPs and IdPs have to perform a prior metadata exchange and configuration to establish trust among each others.

Figure 10 shows the most relevant steps of OpenID, abstracting away the steps that are irrelevant for our analysis, such as the IdP discovery phase, the association session protocol and the user authentication (grey arrows). In the remainder, we call *the OpenID SSO Protocol* the steps from S1 till S2 and *the OpenID authentication protocol* the steps from A1 till A4.

The OpenID SSO protocol is initiated by C who accesses a resource URI at SP in step S1. Between the steps S1 and A1, SP executes the association session protocol with IdP in order to generate a shared secret key K, used to sign and verify assertions, and a value H used as a pointer to refer to the key K. Once SP possesses K and H, it issues an authentication request $\mathrm{AuthReq}(\mathrm{C}, \mathrm{IdP}, \mathrm{H}, \mathrm{SP})$ and redirects C to IdP (steps A1 and A2). C authenticates now with IdP (abstracted away). If the authentication succeeds, IdP issues an authentication assertion $\{AA\}_K$ where $AA = \mathrm{AuthAssert}(\mathrm{IdP}, \mathrm{C}, \mathrm{SP}, \mathrm{H})$ and K is the key associated with H. The assertion is placed into an HTML form that C automatically posts to SP (steps A3 and A4). Upon receiving the assertion, SP verifies the signature and delivers the resource to C (step S2). It is worth pointing out that the OpenID standard does not describe a mechanism to let SP recover in step S2 the resource originally asked in S1. This feature is offered by many actual implementations using customized techniques.

Trust and Transport Protocol Assumptions. As for SAML SSO, also OpenID works under the assumptions that IdP is not compromised and that IdP is trusted by SPs to generate authentication assertions. The latter requires a certain care from SPs as in principle any entity can claim itself to be an IdP. SPs are assumed to be capable to select those IdPs that can be considered trustworthy. Of course, if the IdP is not trusted in generating authentication assertions, then the OpenID protocols become useless.

The OpenID specifications strongly recommend the use of SSL connections for all parts of the interaction, including communication with the user. Not following this recommendation would make the OpenID protocols vulnerable in many trivial aspects that may not fit relevant business scenarios. In our analysis we follow this recommendation and we assume then that the protocol is working under the assumptions *(TP1)* and *(TP2)* as discussed in Section 2.1.

Security Requirements. The OpenID specifications do not state the security properties the OpenID SSO protocol and the OpenID authentication protocol are expected to achieve. Similarly as seen for SAML, we expect that the OpenID SSO protocol and the OpenID Authentication Protocol meet, respectively, the property *(P1)* and *(P2)* in Section 2.2.

6.1. Authentication Flaw

Like SAML SSO, we modeled the OpenID protocols of Figure 10 under the assumption that steps S1 and A4 can be transported over different SSL

connections. We fed the SATMC model checker with it and SATMC returned the attack trace in Figure 11, showing that the flaw found in SAML SSO is also present in OpenID.

The attack involves four principals: a client (c), an honest IdP (idp), an honest SP (sp) and a malicious service provider (i). The client c requests uri_i at SP i. Here, the attacker i, impersonating c, requests a different resource to sp and sp reacts starting the OpenID authentication protocol by crafting a proper authentication request for c. The malicious SP i uses this authentication request in its protocol session with c. The protocol simply proceeds according to the OpenID standard resulting in c accessing to a resource of sp, while c originally asked for a resource from i.

The attack in Figure 11 is witnessing that the OpenID SSO protocol does not offer the security property *(P1)*. As said in Section 3, under no circumstance a compromised SP should hijack C's authentication attempt to force C to consume a protected resource at an honest SP. We verified that if C can link the message A1 with S1 or SP can enforce that S1, A1, A4, and S2 are exchanged over the same SSL connection then the attack is solved. However, the client, a web browser guided by a user, have no means of verifying whether the authentication request and the assertion are related to the initial request. Also, the reuse of the same SSL connection for exchanging the messages S1, A1, A4 and S2, though theoretically possible, it is not feasible in practice.

6.2. Exploitations

As seen in Section 3, the authentication flaw on SAML SSO can be exploited in several ways. However, the differences between SAML SSO and OpenID make the exploitations on SAML SSO not directly applicable to OpenID. Indeed, OpenID does not prescribe any parameters to let SP recover its previous state (e.g. `RelayState` in SAML SSO). Therefore, the Cross-Site Request Forgery and the Cross-Site Scripting attacks described in Section 3 are no directly exploitable. However, the OpenID specifications enable SP to append customized query parameters to the authentication requests whose names and values are out of the scope of the OpenID specifications. For instance, in this way SP can recover the URI of resource the client originally asked for. Depending on the usage by the SP, these parameters can be exploited to mount XSS and CSRF attacks.

We have verified some OpenID suffers from this kind of vulnerabities. Also in this case, we have promptly informed the vendors and details will be made available to the public as soon as the vulnerabilities are fixed.

6.3. Solutions

This section shows how to apply the countermeasures discussed in Section 5.2 to OpenID. *Enforcing a single session* between SP and C is straight-forward for OpenID and it does not require any adaptation. Particular care must be taken in adapting *self-signed client certificate*. When issuing an Authentication Request, SP calculates the value $n \mid\mid \texttt{HMAC}_{K \mid\mid n}(\texttt{RSA modulus})$, where n is a nonce, K is

a secret known only to SP, `RSA modulus` is the RSA modulus of the public key contained in the client's certificate for the SSL connection in steps S1-A1. Then SP stores it in an URL parameter `ID` and appends it to the Authentication Request as a customized parameter. Afterwards, in step A3, IdP sends the assertion to SP together with `ID`. Upon receiving the assertion, SP checks the value in `ID` as explained in Section 5.

Interesting enough, *cookies* and the *feedback from the user* solutions are already used by actual OpenID implementations. For example, Flickr[7] and Sourceforge[8] use cookies that block the flaw as explained in Section 5. The feedback from the user is specified in the OpenID User Interface Extension specifications (OpenID Foundation, 2009a) and the OpenID User Interface Extension Best-Practices for Identity Providers (OpenID Foundation, 2009b). They permit SP to request IdP to display the IdP's login window on top of the SP's page together with a consent page containing both the IdP and SP's names. Although the focus is merely improving the user experience, this mechanism fixes the flaw.

7. Formal Modeling and Analysis of the SAML 2.0 Web Browser SSO Profile

We carried out the formal analysis of SAML SSO using the AVANTSSAR (Armando et al., 2012) Platform, an integrated toolset for the formal specification and automated validation of distributed, security-sensitive applications. The platform supports a variety of specification languages and a number of model checking tool specifically tailored for the specification and automatic analysis of security-sensitive applications. For the analysis of SAML SSO we used HLPSL++ as specification language and SATMC (Armando and Compagna, 2004) as model checker.

HLPSL++ is a role-based specification language for security protocols. Unlike its predecessor HLPSL (High-Level Protocol Specification Language) (Chevalier et al., 2004), HLPSL++ supports the specification of communication channels that go beyond the Dolev-Yao (DY) model. The DY model assumes that communication between honest principals is controlled by a very powerful intruder (called *Dolev-Yao intruder* (Dolev and Yao, 1983)) capable to overhear, divert, and fake messages. Since SAML SSO, as most browser-based security protocols, runs over SSL, the adoption of a DY intruder for the analysis is problematic. HLPSL++ allows the specification of confidential and authentic channels and therefore is suited to the task of specifying browser-based security protocols.

SATMC is a SAT-based bounded model checker for security protocols. Properties and assumptions on the security of the channels used to transport the protocol messages can be expressed in LTL (Armando et al., 2007, 2009). SATMC

[7] `http://www.flickr.com`
[8] `http://www.sourceforge.net`

performs a bounded analysis of the problem by considering scenarios with a finite number of sessions. At the core of SATMC lies a procedure that, given a security problem, automatically generates a propositional formula whose satisfying assignments (if any) correspond to counterexamples on the security problem of length bounded by some integer k. Intuitively, the formula represents all the possible evolutions, up to depth k, of the transition system described by the security problem. Finding attacks (of length k) on the protocol therefore reduces to solving propositional satisfiability problems. For this task, SATMC relies on state-of-the-art SAT solvers, which can handle propositional satisfiability problems with hundreds of thousands of variables and clauses or more. SATMC can also be instructed to perform an iterative deepening on the number k of steps. As soon as a satisfiable formula is found, the corresponding model is translated back into a *partial-order plan* (i.e., a partially ordered set of rules whose applications lead the system from the initial state to a state witnessing the violation of the expected security property).

In Section 7.1 we provide a brief description of the HLPSL++ specification of SAML SSO and in Section 7.2 we discuss the security analysis of the protocol that we carried out by using SATMC.

7.1. Formal Specification of SAML SSO

An HLPSL++ specification consists of the definition of

1. the basic roles, i.e. the roles played by the agents participating in the protocol;
2. the interplay among the basic roles (role `session`); this includes the declaration of the channels used to exchange the protocol messages;
3. the set of sessions to be analysed (role `environment`).

7.1.1. Roles

Actions carried out by a protocol participant are grouped and specified in a *basic role*. Basic roles are then instantiated and glued together into *composed roles*.

The three basic roles participating in the SAML SSO, namely `client`, `serviceProvider`, and `identityProvider`, are defined in Figures 1, 2, and 3 respectively. The definition of a role includes a list of parameters and the specification of agent playing that role in the protocol. A basic role with parameters thus corresponds to many possible instances of same role obtained by instantiating the parameters with terms of the appropriate type. For instance, the role `serviceProvider` has the following parameters (cf. lines 2-5 of Figure 2): `C`, `IdP`, and `SP` of type `agent`, `KIdP` of type `public_key`, `SP2C_1`, `C2SP_1`, `SP2C_2`, and `C2SP_2` of type `channel` (used by SP to communicate with the client), and `URI` of type `text`. In addition, this role will be played by agent `SP` (cf. line 6). Roles (both basic and composed) can also contain the definition of local variables and constants within the section `local`. For instance, the role `serviceProvider` uses the local variable `State` of type `nat`, `AnyC` of type `agent`, and the local variables `ID` and `Resource` of type `text` (cf. lines 9-12).

```
1  role client(
2      C, IdP, SP      : agent,
3      KIdP            : public_key,
4      C2SP_1, SP2C_1, C2IdP, IdP2C  : channel,
5      Set_SP2C_2      : (agent.channel.channel) set,
6      URI             : text
7  ) played_by C
8  def=
9
10   local
11     State           : nat,
12     ID, Resource    : text,
13     AnySP           : agent,
14     AnyURI          : text,
15     C2SP_2, SP2C_2 : channel
16
17   init
18     State:=0
19
20   transition
21       %% C asks for a resource to SP
22    S1. State=0  /\ SP2C_1(start)
23         =|>
24       State':=2 /\ snd(C2SP_1, SP, URI)
25                 /\ witness(C, SP, sp_c_uri, URI)
26
27       %% C receives an AuthnReq(ID, SP) to be forwarded to IdP
28 A1_A2. State=2  /\ rcv(SP2C_1, SP, C.IdP.(ID'.AnySP').AnyURI')
29         =|>
30       State':=4 /\ snd(C2IdP, IdP, C.IdP.(ID'.AnySP').AnyURI')
31
32       %% C receives a Response(SP, C, IdP, ID) for SP
33 A3_A4.State=4  /\ rcv(IdP2C, IdP, AnySP'.{AnySP'.C.IdP.ID'}_inv(KIdP).AnyURI')
34                /\ in(AnySP'.C2SP_2'.SP2C_2', Set_SP2C_2)
35       =|>
36      State':=6 /\ snd(C2SP_2',AnySP',AnySP'.{AnySP'.C.IdP.ID'}_inv(KIdP).AnyURI')
37
38       %% C receives a resource from SP
39 A4_S2. State=6  /\ rcv(SP2C_2, AnySP, Resource')
40         =|>
41       State':=8
42 end role
```

Figure 1: HLPSL++ specification of the SAML SSO protocol: role C

Local variables can be initialized in the section `init`. For instance, `State` is initially assigned to the value `1` in the `serviceProvider` role (cf. line 14).

The behavior of roles is specified in the `transition` section. This comprises a collection of transitions of the form *Pre* `=|>` *Post*, where *Pre* is the conjunction (denoted by the `/\` symbol) of preconditions for the applicability of the transition and *Post* is the conjunction of effects that result by the execution of the transition. For example, the first transition in Figure 2 (cf. lines 19-22) occurs only if `State=1` and a message `URI` is received on channel `C2SP_1`

```
1  role serviceProvider (
2      C, IdP, SP     : agent,
3      KIdP           : public_key,
4      SP2C_1,C2SP_1, SP2C_2,C2SP_2 : channel,
5      URI            : text
6  ) played_by SP
7  def=
8
9    local
10     State        :  nat,
11     AnyC         :  agent,
12     ID, Resource : text
13
14   init State:=1
15
16   transition
17       %% SP receives a request for a resource and issues an
18       %% AuthReq(ID, SP)
19 S1_A1. State=1 /\ rcv(C2SP_1, AnyC', URI)
20        =|>
21       State':=3 /\ ID' := new()
22                 /\ snd(SP2C_1, AnyC', AnyC'.IdP.(ID'.SP).URI)
23
24       %% SP receives a Response(SP, C, IdP, ID) and serves the
25       %% resource to C
26       %% NOTE: SP check that ID is equal to what it generated before
27 A4_S2. State=3 /\ rcv(C2SP_2, C, SP.{SP.C.IdP.ID}_inv(KIdP).URI)
28        =|>
29       State':=5 /\ Resource' := new()
30                 /\ snd(SP2C_2, C, Resource')
31                 /\ request(SP, C, sp_c_uri, URI)
32 end role
```

Figure 2: HLPSL++ specification of the SAML SSO protocol: role SP

from agent `AnyC'` (represented by `rcv(C2SP_1,AnyC',URI)`) and its execution sets the value of `State` to `3`, generates a fresh value for `ID` (cf. second conjunct in line 21), and makes `SP` (the player of the role) sending the message `AnyC'.IdP.(ID'.SP).URI` to `AnyC'` over channel `SP2C_1`. The primed variable `ID'` denotes the new value of `ID`. When a primed variable occurs in a received message, it means that the variable will be assigned to a new value, the one specified in the corresponding part of the message received. If the primed variable occurs in an outgoing message, then the value just assigned to that variable will be included in the message. In summary, unprimed variables denote message elements that the role is checking while receiving that message (as they are already stored somewhere in its state), while primed variables model those elements that are unknown to the receiver. For instance, the transition of `identityProvider` (see Figure 3, lines 19-21) states that `IdP` checks whether the first two elements of the message received correspond to the (known) values of `C` and `IdP` respectively, while the last three elements, namely `ID'`, `SP'`, and `URI'`, are not checked.

```
1  role identityProvider (
2     C, IdP        : agent,
3     KIdP          : public_key,
4     IdP2C, C2IdP : channel
5  ) played_by IdP
6  def=
7
8    local
9     SP: agent,
10    URI   :  text,
11    ID    :  text,
12    State : nat
13
14   init State:=7
15
16   transition
17   %% IdP receives an AuthReq(ID, SP) from C and issues a
18   %% Response(SP, C, IdP, ID)
19 A2_A3. State=7 /\ rcv(C2IdP, C, C.IdP.(ID'.SP').URI')
20        =|>
21       State':=9 /\ snd(IdP2C, C, SP'.{SP'.C.IdP.ID'}_inv(KIdP).URI')
22 end role
```

Figure 3: HLPSL++ specification of the SAML SSO protocol: role IdP

As mentioned in the paper, we have adapted the formal model in (Armando et al., 2008) so to let the messages of steps S1 and A4 be transported over different SSL connections. In Figure 2, lines 27 and 30, we model a new SSL connection with a pair of channels (namely, `C2SP_2` and `SP2C_2`) different from that used in the first transition (namely, `C2SP_1` and `SP2C_1`). Moreover, in the corresponding transition of the client we let the client to choose the appropriate channel for the specific recipient among the set of available channels (cf. line 34 in Figure 1). Finally, at line 39 in Figure 1, `C` receives the resource on channel `SP2C_2`. Details on the properties of the channels used in our specification are given in Section 7.1.2.

Once the basic roles are defined, their parallel composition is defined by means of the composed role `session` as shown in Figure 4. The parameters of the composed role can be related to those of the component roles so that when the composed role is instantiated its component roles are properly instantiated too. Parameters include the channels used by component basic roles to exchange messages.

7.1.2. Channels

`C2SP_1` and `SP2C_1` model one other two SSL connections between `C` and `SP`: `C2SP_1` carries messages from `C` to `SP`, `SP2C_1` carries messages flowing in the opposite direction. They are assumed to enjoy the following properties (cf. lines 23-33 in Figure 4):

- `confidential(SP, C2SP_1)`, i.e. the output of `C2SP_1` is accessible to `SP`

```
role session (
    C, IdP, SP      : agent,
    KIdP            : public_key,
    C2SP_1, SP2C_1,
    C2SP_2, SP2C_2,
    C2IdP, IdP2C    : channel,
    URI             : text,
    Set_SP2C_2      : (agent.channel.channel) set
)
def=

  init

    %% Channel assumptions
        %% C <-> IdP
       confidential(IdP, C2IdP)
    /\ weakly_authentic(C2IdP)
    /\ authentic(IdP, IdP2C)
    /\ confidential(C, IdP2C)
    /\ link(C2IdP, IdP2C)

    %% C <-> Service Providers
    /\ confidential(SP, C2SP_1)
    /\ weakly_authentic(C2SP_1)
    /\ weakly_confidential(SP2C_1)
    /\ authentic(SP, SP2C_1)
    /\ link(C2SP_1, SP2C_1)

    /\ confidential(SP, C2SP_2)
    /\ weakly_authentic(C2SP_2)
    /\ weakly_confidential(SP2C_2)
    /\ authentic(SP, SP2C_2)
    /\ link(C2SP_2, SP2C_2)

  composition

       client(C, IdP, SP, KIdP, C2SP_1, SP2C_1, C2IdP, IdP2C, Set_SP2C_2, URI)
    /\ serviceProvider(C, IdP, SP, KIdP, SP2C_1, C2SP_1, SP2C_2, C2SP_2, URI)
    /\ identityProvider(C, IdP, KIdP, IdP2C, C2IdP)
end role
```

Figure 4: HLPSL++ specification of the SAML SSO protocol: session

only, and `weakly_authentic(C2SP_1)`, i.e. the input of `C2SP_1` is exclusively accessible to a single, yet unknown, sender;

- `weakly_confidential(SP2C_1)`, i.e. the output of `SP2C_1` is exclusively accessible to a single, yet unknown, receiver and `authentic(SP, SP2C_1)`, i.e. the input of `SP2C_1` is accessible to `SP` only; and
- `link(C2SP_1, SP2C_1)`, i.e. the principal sending messages on `C2SP_1` is the same principal that receives messages from `SP2C_1`.

Same considerations hold for C2SP_2 and SP2C_2, which model the second SSL connection between C and SP.

Channels C2IdP and IdP2C model the SSL connection between C and IdP. The former carries messages from C to IdP, the latter carries the messages flowing in the opposite direction. The properties enjoyed by C2IdP (IdP2C) are similar to those of C2SP_1 (resp. SP2C_1), the only difference being that IdP2C is confidential to C and not simply weakly confidential thanks to the authentication of C on IdP (cf. lines 16-20).

A precise definition of the properties of channels supported by HLPSL++ can be found in (Armando et al., 2008).

7.1.3. Environment

The HLPSL++ specification is concluded by the top-level role environment (see Figure 5). This role includes a const section (cf. lines 9-17) containing the declaration of global constants, the definition of the knowledge initially possessed by the intruder (cf. lines 22-25), and a composition section (cf. lines 28-31) that defines the parallel composition of the sessions. Two sessions are combined in our specification:

1. one session in which c and idp play the protocol with i (here acting as SP); the pair of channels c2i_1 and i2c_1 models the first SSL connection, whereas the pair of channels c2i_2 and i2c_2 models the second SSL connection; the pair of channels c2idp_s1 and idp2c_s1 models the SSL connection between C and IdP;
2. and a session in which c and idp play the protocol with sp; the pair of channels c2sp_1 and sp2c_1 models the first SSL connection, whereas the pair of channels c2sp_2 and sp2c_2 models the second SSL connection; the pair of channels c2idp_s2 and idp2c_s2 models the SSL connection between C and IdP;

A constant i of type agent is implicitly declared and associated with the intruder.

7.1.4. Security Properties

The security properties that the protocol is expected to meet are specified in the goal section (cf. lines 34-36). HLPSL++ allows for the specification of an authentication property that corresponds to the notion of agreement as defined in (Lowe, 1997). Thus, authentication_on sp_c_uri (in words *sp authenticates c on uri*) means that whenever sp completes a run of the protocol apparently with c, then *(i)* c has previously been running the protocol apparently with sp, and *(ii)* the two agents agree on the value of uri (Lowe, 1997). This corresponds to one of the two authentication properties implied by *(P1)*.

To specify this property in HLPSL++ it suffices for sp to assert the fact request(sp, c, sp_c_uri, uri) in its last transition (cf. line 31 in Figure 2) and for c to assert the fact witness(c, sp, sp_c_uri, uri) as soon as it sends uri (cf. line 25 in Figure 1). The statement authentication_on sp_c_uri in the goal section (cf. line 35 in Figure 5) states that if request(sp, c, sp_c_uri,

```
role environment()
def=
  local
    %% Maps client c to the channels it is using in sending
    %% authentication assertions to a given SP
    Set_c_SP_2: (agent.channel.channel) set

  const
    sp_c_uri                        : protocol_id,
    c, idp, sp                      : agent,
    c2idp_s1, idp2c_s1,
    c2idp_s2, idp2c_s2,
    c2i_1, i2c_1, c2i_2, i2c_2,
    c2sp_1, sp2c_1, c2sp_2, sp2c_2 : channel,
    kidp, ki                        : public_key,
    uri_i, uri_sp                   : text,
    n                               : text

  init
    Set_c_SP_2 := {i.c2i_2.i2c_2, sp.c2sp_2.sp2c_2}

  intruder_knowledge={c, sp, idp, kidp, ki, inv(ki), n, uri_i, uri_sp,
                      c2i_1, i2c_1, c2i_2, i2c_2,
                      c2sp_1, sp2c_1, c2sp_2, sp2c_2,
                      c2idp_s1, idp2c_s1, c2idp_s2, idp2c_s2}

  composition
       session(c, idp, i, kidp, c2i_1, i2c_1, c2i_2, i2c_2,
               c2idp_s1, idp2c_s1, uri_i, Set_c_SP_2)
    /\ session(c, idp, sp, kidp, c2sp_1, sp2c_1, c2sp_2, sp2c_2,
               c2idp_s2, idp2c_s2, uri_sp, Set_c_SP_2)
end role

goal
 authentication_on sp_c_uri
end goal

environment()
```

Figure 5: HLPSL++ specification of the SAML SSO protocol: environment and goals

`uri)` is asserted, then sometime in the past a corresponding `witness(c, sp, sp_c_uri, uri)` must have been asserted for the authentication property to hold. Indeed, if a request is not matched by a corresponding witness, then the intruder must have been playing the protocol run pretending to be the one that otherwise would have asserted the witness expression.

7.2. Automatic Security Analysis of SAML SSO

By running SATMC against the HLPSL++ specification presented in Section 7.1 the tool returned the attack trace in Figure 7. We then tested our conjecture that the authentication flaw is solved if C can link the Authentication Request received in step A1 with the request in step S1 as discussed in

Section 5. To this end we modified the model in such a way to impose that the second transition of `C` is applicable only if the `SP` is exactly the one expected by `C`. This is done by replacing `AnySP'` with `SP` in lines 28 and 30 of Figure 1. When asked to analyse the resulting protocol SATMC did not report any attack, thereby confirming our conjecture.

8. Related Work

(Groß et al., 2005; Pfitzmann and Waidner, 2003a,b) lay the theoretical basis for a rigorous analysis of web-based federated identity-management protocols (e.g. the SSO protocol proposed by Liberty Alliance in 2002). They discuss some security vulnerabilities and possible preventive measures. Some of these results have been fed into the Liberty Alliance project and indirectly into the SAML 2.0 standard.

Security analyses of the SAML SSO v1.0 are presented in (Groß, 2003) and in (Hansen et al., 2005). The security analysis presented in our paper refers to SAML SSO v2.0, the latest version of the standard. Moreover, in our work we focus on scenarios that are most likely to occur in actual deployments. For instance, unlike (Hansen et al., 2005) we do not assume that SPs are trustworthy and unlike (Groß, 2003) we assume that messages are exchanged over secure channels as recommended by the standard.

In (Wang et al., 2012) the authors present a traffic-guided black-box security study of web SSO systems. The web traffic is processed for inferring syntactics and semantics of message fields showing what fields an attacker could play with. Afterwards, the security analyst interprets and uses them for verifying potential vulnerabilities on implementations. Eight serious implementation-specific vulnerabilities and their exploitation are reported proving the effectiveness of their approach. This work shares commonalities with ours as well as differences. First, our approach classifies as model-based testing in which models and real implementations are sitting on different abstraction layers. This may result in overlooking low-level vulnerabilities therefore missing a wide-range of implementation-specific security problems. On the contrary, the traffic-guided approach does not use any model but enriches web traffic with new information for guiding the security analyst. Security flaws may lay in deeper states of the application and not using models may leave them undetected. Second, in our work we use automatic reasoning techniques while the traffic-guided approach relies on the security expert for verifying hypothesis. However, formal methods require a formal specification that might not be available in practice and it has to be provided by a modeler. In addition, the counterexamples returned by the model checker are not directly verifiable against real implementations and model checkers do not offer any support on that. Therefore, the security analyst has also the burden of testing implementations.

In (Armando et al., 2008) we provide a formal model of the SAML SSO protocol as well as of a variant implemented in the SAML-based SSO for Google Apps. By using SATMC, we discovered a subtle man-in-the-middle attack on the SAML-based SSO for Google Apps. In reaction to this discovery Google

modified the implementation of the protocol. The version of the protocol described in Section 4 is the one currently in use by Google and therefore does not suffer from the attack reported in (Armando et al., 2008). Interestingly, in (Armando et al., 2008) we did not find any attack on the Web Browser SAML 2.0 SSO profile as in our analysis we assumed that communication between C and SP is carried over a single unilateral SSL connection.

In (Armando et al., 2011) we have adapted the formal model of (Armando et al., 2008) so to let the messages of steps S1 and A4 be transported over different SSL connections. By using SATMC we discovered the previously unknown attack described in Section 3. This paper consolidates the work reported in (Armando et al., 2011) and shows that the vulnerability also applies to OpenID (OpenID Foundation, 2007b), another prominent browser-based SSO protocol as we discussed in Section 6.

9. Conclusions

Authentication protocols are notoriously difficult to get right, even more so for browser-based authentication protocols because "browsers, unlike normal protocol principals, cannot be assumed to do nothing but execute the given security protocol" (Groß et al., 2005). In this paper we have showed that browser-based SSO protocols are no exception. We have presented an authentication flaw that applies to both SAML SSO and OpenID. This flaw allows a compromised service provider to hijack a client authentication attempt or force the latter to access a resource without its consent or intention. We have showed how this flaw can be generally exploited, and reported related security issues that we have detected in actual SAML-based SSO solutions developed by prominent software companies, including a severe attack on the SAML-based SSO for Google Apps. We have presented a number of possible solutions that mitigate or even solve the problem.

All our findings have also been discussed with various companies developing browser-based SSO protocols as part of their software portfolio. At the same time, we have discussed the oucomes of our work with members of OASIS and a SAML V2.0 Errata has been redacted and approved (OASIS, 2012). Last, but not least, we have recently informed the OpenID working group and some of the implementations of this solution about our recent findings on OpenID.

As for future work we aim at automatically generate test-cases to verify implementations of browser-based SSO protocols with minimal human intervention. First steps in this direction are reported in (Armando et al., 2012).

Acknowledgments

This work has partially been supported by the FP7-ICT Projects AVANTSSAR (no. 216471) and SPACIOS (no. 257876), and by the project SIAM funded in the context of the FP7 EU "Team 2009 - Incoming" COFUND action. We are also grateful to Scott Cantor, Brian Eaton, Matteo Grasso, and the SAP NetWeaver SIM team for the valuable discussions and feedback they provided.

References

Armando, A., Arsac, W., Avanesov, T., Barletta, M., Calvi, A., Cappai, A., Carbone, R., Chevalier, Y., Compagna, L., Cuéllar, J., Erzse, G., Frau, S., Minea, M., Mödersheim, S., von Oheimb, D., Pellegrino, G., Ponta, S. E., Rocchetto, M., Rusinowitch, M., Dashti, M. T., Turuani, M., Viganò, L., 2012. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In: Flanagan, C., König, B. (Eds.), TACAS. Vol. 7214 of LNCS. Springer, pp. 267–282.

Armando, A., Carbone, R., Compagna, L., July 2007. LTL model checking for security protocols. In: 20th IEEE Computer Security Foundations Symposium (CSF20). Venice (Italy).

Armando, A., Carbone, R., Compagna, L., 2009. LTL Model Checking for Security Protocols. In: Journal of Applied Non-Classical Logics, special issue on Logic and Information Security. Hermes Lavoisier, pp. 403–429.

Armando, A., Carbone, R., Compagna, L., Cuéllar, J., Pellegrino, G., Sorniotti, A., 2011. From multiple credentials to browser-based single sign-on: Are we more secure? In: Camenisch, J., Fischer-Hübner, S., Murayama, Y., Portmann, A., Rieder, C. (Eds.), SEC. Vol. 354 of IFIP Advances in Information and Communication Technology. Springer, pp. 68–79.

Armando, A., Carbone, R., Compagna, L., Cuéllar, J., Tobarra, M. L., 2008. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In: FMSE. ACM.
URL http://doi.acm.org/10.1145/1456396.1456397

Armando, A., Compagna, L., 2004. SATMC: a SAT-based model checker for security protocols. In: Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04). Vol. 3229 of LNAI. Springer-Verlag, Lisbon, Portugal.

Armando, A., Pellegrino, G., Carbone, R., Merlo, A., Balzarotti, D., 05 2012. From model-checking to automated testing of security protocols: Bridging the gap. In: TAP 2012, 6th International Conference on Tests and Proofs, May 31-June 1, 2012, Prague, Czech Republic / To be published also in LNCS, 2012, Springer. Prague, CZECH REPUBLIC.
URL http://www.eurecom.fr/publication/3659

Barth, A., Jackson, C., Mitchell, J. C., 2008. Robust defenses for cross-site request forgery. In: 15th ACM Conference on Computer and Communications Security (CCS 2008).
URL http://seclab.stanford.edu/websec/csrf/csrf.pdf

Bellare, M., Canetti, R., Krawczyk, H., 1996. Keying hash functions for message authentication. In: Koblitz, N. (Ed.), Advances in Cryptology CRYPTO 96. Vol. 1109 of LNCS. pp. 1–15.
URL http://dx.doi.org/10.1007/3-540-68697-5_1

Chevalier, Y., Compagna, L., Cuellar, J., Hankes Drielsma, P., Mantovani, J., Mödersheim, S., Vigneron, L., 2004. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In: Proc. SAPS'04. Austrian Computer Society.

Dolev, D., Yao, A., 1983. On the Security of Public-Key Protocols. IEEE Transactions on Information Theory 2 (29).

Google, 2008. Web-based SAML-based SSO for Google Apps. `http://code.google.com/apis/apps/sso/saml_reference_implementation_web.html`.

Google, 2009. Google security and product safety. [Online; accessed 16-July-2012].
URL `http://www.google.com/about/company/security.html`

Groß, T., Dec. 2003. Security analysis of the SAML Single Sign-on Browser/Artifact profile. In: Proc. 19th Annual Computer Security Applications Conference. IEEE.

Groß, T., Pfitzmann, B., Sadeghi, A.-R., 2005. Browser model for security analysis of browser-based protocols. In: ESORICS.

Hansen, S. M., Skriver, J., Nielson, H. R., 2005. Using static analysis to validate the SAML single sign-on protocol. In: WITS '05. ACM Press, New York, NY, USA.

Internet2, 2007. Shibboleth Project. `http://shibboleth.internet2.edu/`.

Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., Wachter, C., 2012. Ron was wrong, whit is right. IACR Cryptology ePrint Archive 2012, 64.

Lowe, G., 1997. A hierarchy of authentication specifications. In: Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW'97). IEEE Computer Society Press.

Microsoft, 2011. Windows Live ID. `http://www.passport.net/`.

Novell, 2011. Access Gateway Appliance security concerns poisoning or tampering cookies. [Online; accessed 16-July-2012].
URL `http://www.novell.com/support/kb/doc.php?id=7008342`

OASIS, 2004. Identity Federation. Liberty Alliance Project. `http://www.projectliberty.org/resources/specifications.php`.

OASIS, March 2005a. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/samlbindings-2.0-os.pdf`.

OASIS, March 2005b. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. http://docs.oasis-open.org/security/saml/v2.0/samlprofiles-2.0-os.pdf.

OASIS, March 2007. SAML V2.0 – Technical Overview. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security.

OASIS, May 2012. SAML Version 2.0 Errata 05.
URL http://docs.oasis-open.org/security/saml/v2.0/sstc-saml-approved-errata-2.0.pdf

OpenID Foundation, December 2007a. OpenID Authentication 2.0. Available at http://openid.net/specs/openid-authentication-2_0.html.

OpenID Foundation, 2007b. OpenID Specifications. http://openid.net/developers/specs/.

OpenID Foundation, May 2009a. The OpenID User Interface Extension 1.0, draft 0.5. Available at http://svn.openid.net/repos/specifications/user_interface/1.0/trunk/openid-user-interface-extension-1_0.html.

OpenID Foundation, August 2009b. the OpenID User Interface Extension Best-Practices for Identity Providers. Available at http://wiki.openid.net/w/page/12995153/Details-of-UX-Best-Practices-for-OPs.

Pfitzmann, B., Waidner, M., 2003a. Analysis of Liberty Single-Sign-on with Enabled Clients. IEEE Internet Computing 7 (6).

Pfitzmann, B., Waidner, M., 2003b. Federated identity-management protocols. In: Security Protocols Workshop.

Schneier, B., 1996. Applied cryptography - protocols, algorithms, and source code in C (2. ed.). Wiley.

UNINETT, 2010. simplesamlphp-1.6.3 is available, with a security fix. [Online; accessed 16-July-2012].
URL https://rnd.feide.no/2010/12/20/simplesamlphp-1-6-3-is-available-with-a-security-

Wang, R., Chen, S., Wang, X., 2012. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. Security and Privacy, IEEE Symposium on 0, 365–379.

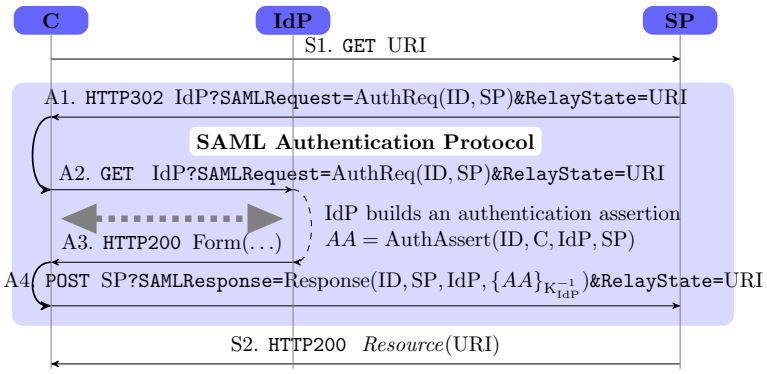


Figure 6: SAML SSO Protocol: SP-Initiated SSO with Redirect/POST Bindings

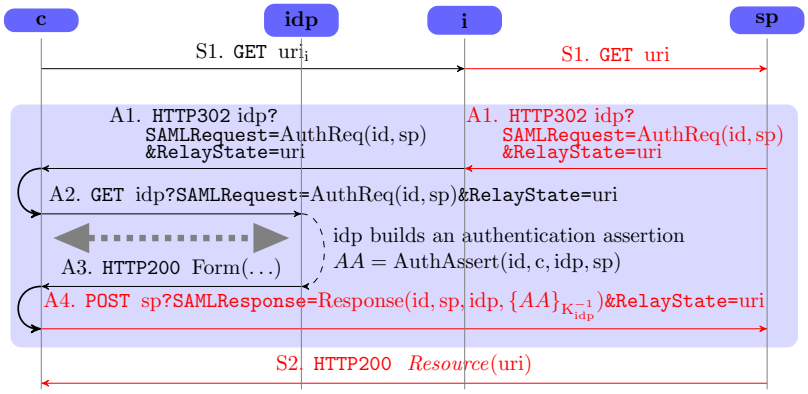
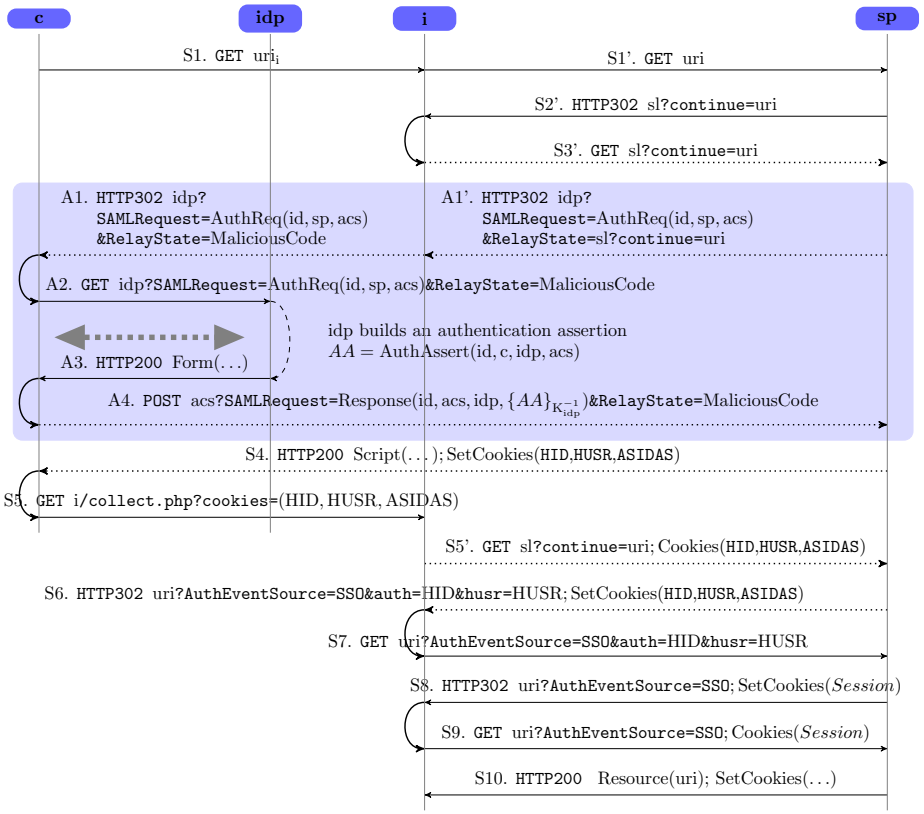


Figure 7: Authentication Flaw of the SAML 2.0 Web Browser SSO Profile

Legenda: ·····➤ : `https`

Figure 8: SAML-based Single Sign-On for Google Apps



Legenda: ·····➤ : `https`

Figure 9: XSS Attack on the SAML-based SSO for Google Apps

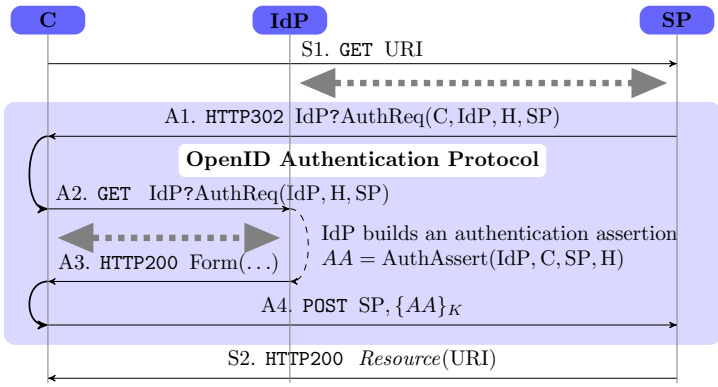


Figure 10: OpenID

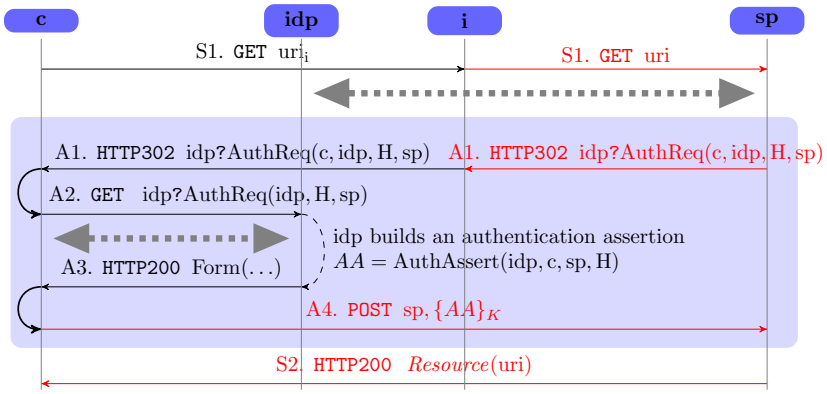


Figure 11: Authentication Flaw of the OpenID SSO Protocol

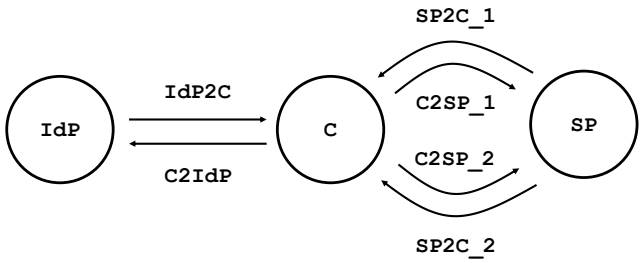


Figure 12: Communication channels among basic roles