Security Analysis of OpenID

Pavol Sovis, Florian Kohlar, Joerg Schwenk
{*vorname.nachname*}@rub.de
Ruhr-University Bochum
Bochum, Germany

Abstract: OpenID is a user-centric and decentralized Single Sign-On system. It enables users to sign into Relying Partiesby providing an authentication assertion from an OpenID Provider. It is supported by many leading internet companies and there are over a billion accounts capable of using OpenID. We present a security analysis of OpenID and the corresponding extensions and reveal several vulnerabilities. This paper demonstrates how identity information sent within the OpenID protocol can be manipulated, due to an improper verification of OpenID assertions and no integrity protection of the authentication request.

1 Introduction

The web applications' de facto standard is the "username/password" authentication over an TLS/SSL [DA99, FKK96] secured connection. However, this mechanism is an unacceptable solution for the internet today, because it leads to several security problems, the number of usernames and passwords to remember being the worst. It leads to forgotten passwords, resulting in the need of a password renewal over e-mail and/or low-entropy passwords, which are easy to remember (and easy to guess as well). In order to solve these problems, Single Sign-on systems have been introduced and their goal is to authenticate a user only once and obviating the need of re-authentication. A prominent player on this field is OpenID [FRHH07]. It authenticates a user against a web application using an authentication assertion gained by a trusted third party. The biggest supporters of OpenID count Google, Microsoft, Yahoo, MySpace, Verisign, GMX/Web.DE or France Telecom and this provides for a solid user-base of more than a billion OpenID-capable users worldwide. Besides, many companies already support OpenID authentication, including Facebook, Sears, KMart or LiveJournal.

2 Related Work

Microsoft Passport has been analyzed by Kormann and Rubin [KR00] and several weaknesses have been found. They are based on redirecting the user to a bogus Passport server,

either by deploying a fake merchant site and luring unsuspicious users to this site (e.g. through phishing attacks) or actively by attacking the DNS or modifiying a response from a legitimate service provider. The bogus server may act as a proxy and thus obtain the user's credentials. Microsoft Cardspace - the successor of MS Passport - was analyzed by Gajek, Schwenk, Steiner and Chen [GSSX09], who were able to steal a user's security token and subsequently impersonate him. SAML, an XML-based standard also used widely to encorporate Single Sign-On, was analyzed by Groß [Gro03], who intercepted the authentication token from a referer tag by redirecting the user to a server under the adversary's control. His analysis led to a SAML revision, which was later proven by Groß and Pfitzman [GP06] to be also vulnerable. Pfitzman cooperated with Waidner [PW03] on an analysis of another SAML based Single Sign-On system - the Liberty Single Sign-On protocol - and found similar flaws.

OpenID has not yet been examined with respect to security thoroughly. Eugene and Vlad Tsyrklevich [TT07] presented several OpenID Authentication 1.0 related attacks at the Black Hat USA 2007 Conference and pointed out phishing and unauthenticated Die-Hellman Key Exchange as the biggest shortcomings of OpenID. Shakir James [Jam] analyzed Web Single Sign-On Systems in his report and again identified phishing as the major security issue regarding OpenID and called attention to the lack of security related material in the documentation of the OpenID Suite. Newman and Lingamneni [NL08] have conducted an attack which results in the victim being logged in at the Relying Party as an adversary (Session Swapping). This is possible due to the lack of any bond between a positive authentication assertion from the OpenID Provider and the victim's User agent. Barth, Jackson and Mitchell [BJM08] proposed a mechanism to mitigate this attacks by a secret token validation technique, where the Relying Party generates a fresh nonce at the start of each protocol flow, stores it in the user's browser-cookies and simultaneously appends it to the authentication request. The OpenID Provider returns the nonce in the authentication response. The user's cookie must match the nonce in this response in order for the User to become authenticated at the Relying Party.

3 OpenID Authentication 2.0

3.1 OpenID Roles

- The **User** is an entity wanting to authenticate against a Relying Party with his digital identity.
- The **Identifier** is generally a url, representing the User. It points to a resource, which holds information such as the User's OpenID Provider url, version of OpenID which the OpenID Provider is compatible with etc.
- The **Relying Party** is an entity accepting an assertion from an OpenID Provider, representing a digital identity of a specific User.
- The **OpenID Provider** or **Identity Provider** (interchangeable terms) is responsible

for authenticating the User against a Relying Party, therefore it is the trusted third party on which the User as well as the Relying Party rely. In order to do so, the User must authenticate against the OpenID Provider first and so prove his digital identity. This identity is then used to sign-in the User at the Relying Partyby accepting a security assertion from the OpenID Provider.

- The **Identifier host** is the host, where the Identifier-describing resource resides.

3.2 How does OpenID work?

OpenID is a suite of protocols, which enables users to authenticate against multiple web applications (Relying Parties) using only a single identity. In order to do this, the user must create such an identity at an OpenID Provider of his choice, link this identity to any Relying Party and use it afterwards as a key, proving his identity at the Relying Party. The concept of identity linking (shown in Figure 1) is a mechanism to create a trust relationship between the Relying Party and an OpenID Provider. Afterwards, the Relying Party recognizes the user by his OpenID Provider-identity. The OpenID Provider-identity is transported to the Relying Party in form of an assertion from the OpenID Provider.

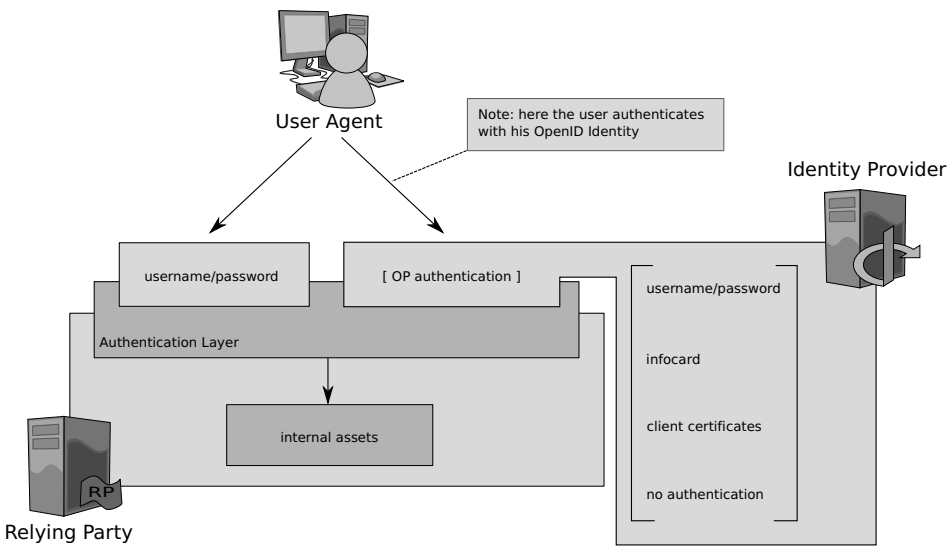

Figure 1: The OpenID Authentication Concept

A typical OpenID Authentication 2.0 Protocol Protocol flow corresponds to Figure 2 and runs as follows:

1. OpenID Authentication 2.0 Protocol Protocol is initiated by the User by requesting

the Relying Party's site.

2. The Relying Party responds with its login page presenting an input field for an Identifier.
3. The User enters his Identifier and submits the login page form, i.e. requests OpenID Authentication 2.0 Protocol.
4. The Relying Party performs discovery upon the received Identifier i.e. retrieves the data resource held at an Identifier's url and
5. subsequently receives metadata representing the User and his OpenID Provider.
6. Based upon metadata from the previous step, the Relying Party requests an association from the OpenID Provider, i.e. requests to exchange a shared secret.
7. The OpenID Provider responds with a shared key, which is encrypted (either using HTTPS as transport protocol or using Die-Hellman Key Exchange).
8. The Relying Party then redirects the User to the OpenID Provider by sending a HTTP(S) response with the redirect header pointing to the OpenID Provider's endpoint.
9. The User is presented with a login form at the OpenID Provider.
10. The User fills out the login form and submits it, hence authenticating against the OpenID Provider.
11. The OpenID Provider verifies the User's credentials and, if these are valid, redirects the User to the Relying Party along with the authentication result (MAC-protected by the previously established shared key). Again, this is done using a HTTP(S) redirect with the 'Location:' header pointing to the Relying Party's endpoint. The assertion in this request to the Relying Party indicates the login success from the OpenID Provider and the MAC ensures the integrity of the response.
12. According to the OpenID Provider's response, the User is either authenticated against the Relying Party or presented with an adequate error message.

The transport protocol used in OpenID Authentication 2.0 Protocol flow is either HTTP or HTTPS (HTTP used within a TLS/SSL secured channel). Independent of the method used (POST, GET), OpenID facilitates a key-value representation of its payload, e.g. "$openid.claimed\_id = http://sovo.myopenid.com$" or "$mode : error$". We refer to such pairs as OpenID parameters. OpenID uses `HTTP 302` status codes to redirect the user from the Relying Party to the OpenID Provider and vice versa. An example (based on Figure 2) of such redirection is given in the following listing:

1. The user initiates a HTTP request in step 3.
2. The user obtains a response to this request in step 8, comprising the HTTP 302 status code as well as the "Location:" header set to the desired destination, while the OpenID parameters are a part of the URL in this header.

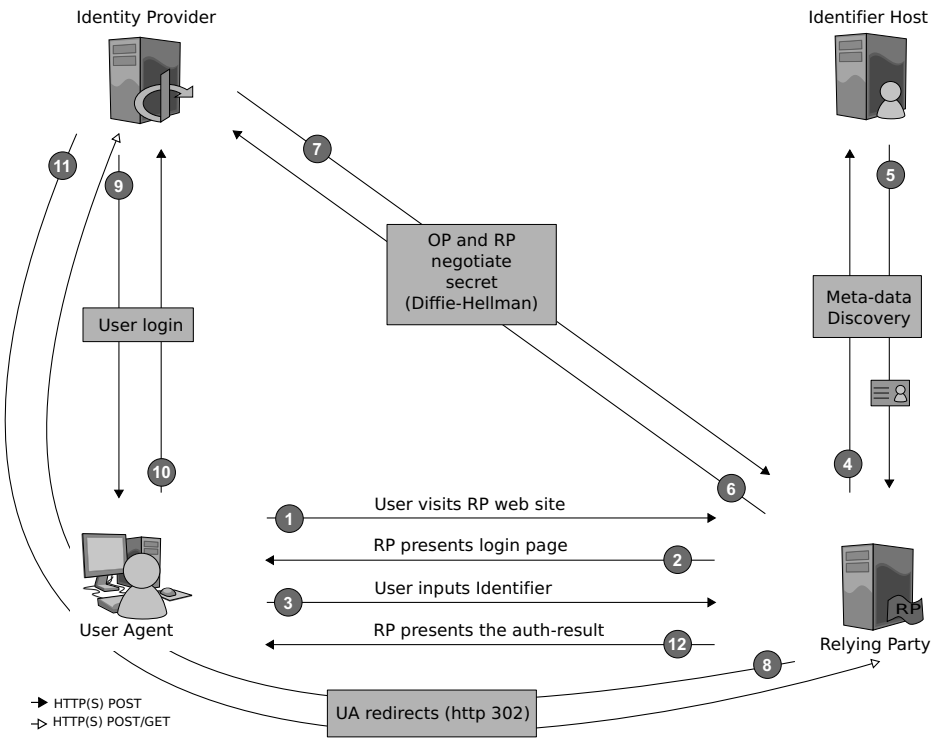

Figure 2: Typical OpenID Protocol Flow

3. The user requests the URL contained in the ”Location:” header from step 8.

An analog procedure is used to redirect the user back to the Relying Party. The security of OpenID messages can be divided into transport layer security (i.e. either using HTTP or HTTPS) and message level security (e.g. message authentication codes or MACs). A MAC is a hash-value generated over a specified list of parameter values xor-ed with the shared secret (pre-established in steps 6 and 7 in Figure 2), thus providing integrity of the OpenID message. Due to compatibility with specific OpenID flows which are not discussed in this paper, the only message in the whole OpenID flow, which is secured by the MAC, is the message from step 11 (see Figure 2).

3.3 Extensions

The basic OpenID parameters, which are compulsory and neccessary in any valid assertion, form the minimum (later also refered to as ”void”) assertion representing a positive or negative result of authentication at the OpenID Provider. However, OpenID al-

lows for extra identity information, such as email, name, date of birth and even self-defined parameters. These are facilitated through so called extensions. These can be appended to the compulsory parameters via a mechanism very similar to XML namespaces. Whereas standard parameters are key-value pairs in the form "[openid-prefix].[parameter-name]=[parameter-value]", e.g. "openid.identity=fooname.fooprovider.com", extensions must be defined first by a namespace of the form "[openid-prefix].[openid-extension-alias]=[extension-url]". For instance, a namespace may be set by "$openid.ns.sreg = http : //openid.net/extensions/sreg/1.1$" as is the case with OpenID Simple Registration Extension 1.0 [HDR06], and any parameters within this extension can be further addressed with help of the previously defined alias, e.g. "$openid.sreg.email = my@example.email$". Whereas OpenID Simple Registration Extension 1.0 can be used to send only a small set of predefined attributes, OpenID Attribute Exchange 1.0 [HBH07] allows to send custom attributes, which makes OpenID very flexible.

4 OpenID Security Analysis

In this section, we discuss several shortcomings that exist, if HTTP endpoints are used at the Relying Party and the OpenID Provider, though the User experiences HTTPS indicators at both of these parties.

4.1 Wrong Approaches on Transport Security

The endpoints of many OpenID Providers or Relying Parties are strictly HTTPS based. The problem is, if they are addressed via HTTP, they simply redirect the request to the HTTPS equivalent and proceed with the protocol flow (see Figure 3). This section comprises the dangers of such a workaround. The User's Identifier is responsible for the OpenID Provider's endpoint. In general, the User is given his identifier by the OpenID Provider, hence the OpenID Provider is overall responsible for the HTTP/HTTPS nature of its endpoint. Furthermore, the Relying Party sending an authentication request to the OpenID Provider is responsible for the $return\_to$ parameter representing the Relying Party's endpoint, where the User will later be redirected to. If both of these endpoints are HTTP URLs, then both of the User's redirects (steps 8 and 11 in Figure 2 or 9 and 15 in 3) are subject to forgery. The fact, that both of these parties may only allow communication over a TLS/SSL secured channel yields a false impression of security from the user's point of view.

The individual steps (of which 8,9 and 14,15 represent the redirects) represent the following workflow:

1 The User visits *http://www.plaxo.com* and klicks on the **Sign in** link.

2 The User receives a response redirecting him to *https://www.plaxo.com/signin?r=/events*.

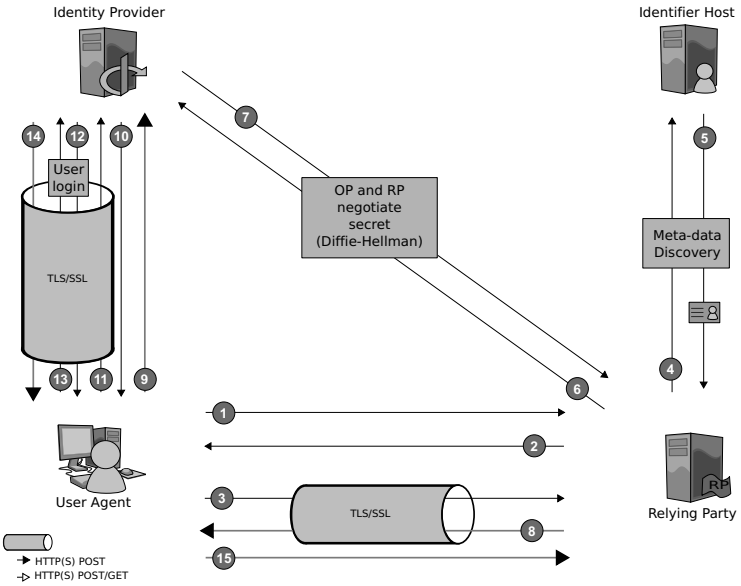

Figure 3: Reducing complexity to HTTP

3 The User chooses to sign in with OpenID, inserts his OpenID Identifier, and subsequently submits the form.

4,5 The Relying Party receives the User's meta-data and searches for the corresponding OpenID Provider.

6,7 An association must be made prior to exchanging a secret. This is done only once and left out in later iterations of the protocol flow.

8 The User receives a response within a secured channel, i.e. using the HTTPS protocol, with the location header pointing to $http://www.myopenid.com/server?openid.assoc$ ..., i.e. the unsecure HTTP protocol.

9,10 The initial request at the OpenID Provider is a HTTP request, hence resulting in another redirect advising the User to move to HTTPS.

11-14 The User inserts his credentials in a form and, in case he successfully authenticated against the OpenID Provider, he gets redirected back to the Relying Party with the authentication result (assertion) attached as GET parameter in the "Location" header.

15 The User evaluates the "Location" header from the previous response, containing the HTTP Endpoint of the Relying Party, and gets redirected there by the User agent.

In Figure 3, the authentication request (represented by the steps 8 and 9) can be modified after the User follows the redirect command (step 9). The authentication response (steps 14 and 15) can be modified as well (step 15). In Figure 4, we have stripped the communication up to these redirects. The two disctinct paths still represent a problem for a potential attacker, as he would need to attack these messages at two distinct network nodes. However, if we think of the User as a network node, which uses a gateway (e.g. an internet provider), both of these redirects may share several nodes on their way. If any of these shared nodes is attacked, then both of these redirects are susceptible to forgery as the information within is transported in plaintext. From the User's point of view, this attack is hard to detect, because it represents the standard OpenID flow and the User actually observes all neccessary HTTPS indicators.

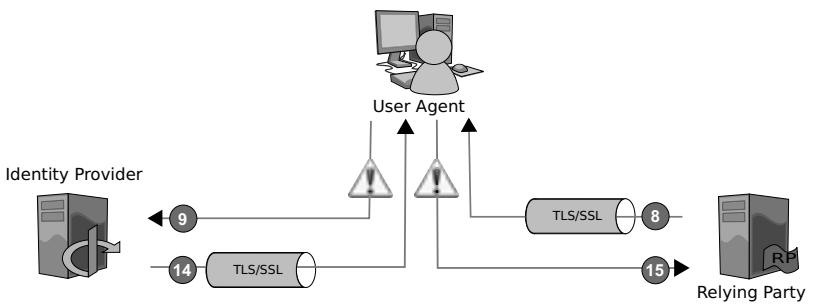

Figure 4: Reducing complexity to HTTP with emphasis on redirects

The topic discussed in this section does not present a threat on its own, it rather provides a perfect fundament for actual attacks discussed in the following sections of this document (i.e. Parameter Injection and Parameter Forgery).

4.2 Parameter Injection

In this section, we exploit the message level security mechanism of OpenID - MAC. With respect to MACs, the two most important OpenID parameters are `openid.sig`, representing the authentication code itself, and `openid.signed`, containing the hashvalue computed over all parameters and xor-ed with the pre-established shared key. The OpenID Authentication 2.0 Protocol Specification states, that if a positive assertion (meaning the User authenticated successfully) is received by the Relying Party, it must not be accepted until it is verified first. Any successful verification must satisfy, among others, the condition that 'the MAC of the assertion is valid and all required fields are MAC-protected'. Hence if a parameter is not defined as required (speaking of which, **none of the identity-related extension parameters are required**) and is not listed in `openid.signed`, it is automatically subject to forgery. In other words, appending arbitrary unused parameters to a MAC-protected message does not invalidate the assertion's MAC and the message stays

intact and valid in the eyes of the Relying Party.

The MAC-protected parameters are all part of the value of the ”openid.signed” parameter. Based on this parameter, we have the following options (examples make use of the OpenID Simple Registration Extension 1.0):

- parameter: ”openid.signed=...sreg.nickname,sreg.email,sreg.fullname...”
  - changing the ”openid.sreg.email” value in this setting would lead to a MAC verification missmatch, thus leading to an invalid assertion
  - however, appending the date of birth by appending the parameter ”openid.sreg.dob” would keep the MAC intact, leading to a valid assertion

Parameters returned to the Relying Party are affected by the Relying Party's request. The request contains a list of parameters, which should be returned by the OpenID Provider (e.g. ”openid.sreg.required=...”).
The OpenID Simple Registration Extension 1.0 states, that the ”openid.signed” list contained in the following response must include the returned ”sreg” parameter names and that the Relying Party bears responsibility of how to behave in case of missing required or additional unrequested parameters. As a consequence, the Relying Party may accept unsolicited parameters either as part of a normal behaviour, or as an implementation error. In fact, the attacker does not care which of both behaviours is the case, as long as such parameters are accepted. In the next section, we show how we can use such ”optimistic” behaviour to manipulate parameters, which have explicitely been requested and are MAC-protected.

4.3 Parameter Forgery

In the 'Parameter Injection' Section, we have shown how we can append our own parameters to the OpenID Authentication 2.0 Protocol response in the authentication phase. The problem, however, was that we were not able to append parameters which already were a component of the response, because they were part of the MAC and hence any modification would lead to a MAC-verification mismatch. Therefore we were only able to inject unused parameters. In the parameter forgery attack, we go a step further by removing parameters from the list of requested parameters ergo leaving it ”void”. As a result, in combination with parameter injection, we can modify any parameters we want, of course with the exception of obligatory OpenID Authentication 2.0 Protocol parameters, which must always be part of the MAC (marked as required in the specification).

Parameter Forgery is based on the fact, that although the OpenID Authentication 2.0 Protocol responses in the authentication phase are MAC-protected by the OpenID Provider, the request does not include any MAC and is therefore prone to forgery. The integrity of a request is in general secured either by the transport layer (using HTTPS) or not secured at all. In many scenarios, however, the integrity is naively achieved through the usage of

”semi-effective” HTTPS redirects, which do not take care of the integrity thoroughly, e.g. if the redirect is HTTPS, but the destination ’Location:’ header url inside is HTTP. Such a redirect is only secure on the way from the Relying Party to the User, but not further. Under such circumstances, there is no integrity on the transport layer and since there is no integrity at the application layer, any adversary acting as ’man-in-the-middle’ may modify the requests.

The reason why the whole request modification effort is performed is actually modifying the response by injecting parameters. According to a property of a modern identity metasystem - ’minimal disclosure’ - the OpenID Provider should protect its User’s privacy by returning only those parameters which it has explicitely been asked for, hence the OpenID Provider must check the request for requested parameters (this is not postulated by the OpenID Authentication 2.0 Protocol specification explicitely, but it is generally the case). These parameters are usually requested in two ways: as part of the `openid.sreg.required` field, meaning that these parameters are needed to successfully sign in the User, or they are listed in the `openid.sreg.optional` field, meaning they are desired, but the Relying Party does not rely on them to be returned from the OpenID Provider (on that matter, one can similarly attack any other OpenID extension, e.g. OpenID Attribute Exchange 1.0). It then depends on the OpenID Provider how it copes with such requested parameters, but if a parameter is not part of the ’required’ or ’optional’ field, it should not be sent (of course with the exception of the assertion-relevant required parameters, which are sent always, but generally do not contain any of the User’s private data). There is a direct relationship (the response-parameters dependend on the request-parameters) between the ’required’ and ’optional’ fields in the request and the returned fields in the response.

That being said, demanding no privacy-relevant parameters in the request inevitably leads to sending no privacy-relevant data in the response, ergo ”no parameters asked” means ”no parameters returned”. In such cases, only the basic assertion, specifying that the User has either successfully signed into the OpenID Provider or not, is sent back to the Relying Party.

The reason why such ’void’ assertions may be very interesting for an adversary lies in the ’Parameter Injection’ attack. If we strip the request of any demands (no parameters marked as required or optional), then there is no reason why an OpenID Provider would send any extra data back to the Relying Party (see Figure 5). Consequently, the OpenID Provider ends up sending a ’void’ assertion leaving all OpenID User relevant data vulnerable to the parameter injection. The adversary is then feasible to change almost anything.

We can use this along with modifying the extension parameters. Besides some extensions, which are solely informative and provide only data retrieval methods, OpenID also allows special extensions, which enable the Relying Parties to store data at the OpenID Provider. This way, the severity of this attack grows, because changing such parameters may affect

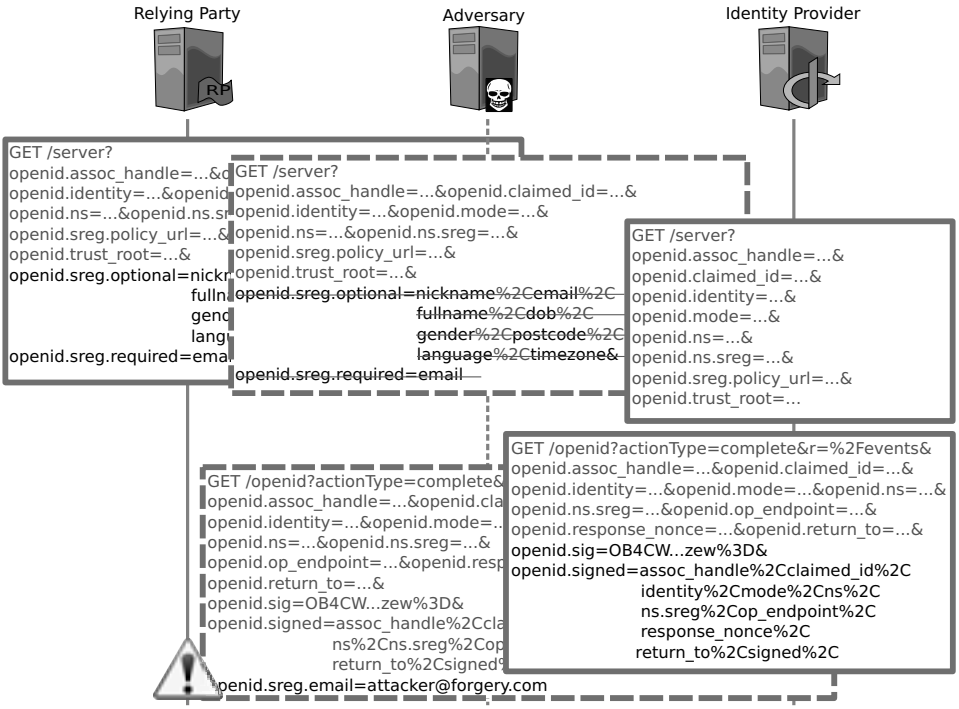

Figure 5: Parameter Forgery combined with Parameter Injection

the OpenID Provider and all Relying Parties therefrom.

5 Conclusion & Future Work

We have provided a description and analysis of the OpenID Single Sign-On protocol and its extensions. The model of OpenID seems to be a suitable Single Sign-On solution for the Internet of today. It has remarkable usability properties and the concept of extensions makes it very flexible. Besides that, giving the control in the user's hands with such a high grade of decentralization rises its popularity significantly. Unfortunately, there are a lot of drawbacks and OpenID has not yet learned from the mistakes of the past.

We have shown that an adversary is able to change arbitrary OpenID extensions' parameters. We recommend that Relying Parties accept only MAC-protected parameters and more importantly - protect the authentication requests with a MAC too. This becomes even more critical when OpenID Attribute Exchange 1.0 is used, due to its ability to change identity information at the OpenID Provider.

Although OpenID has a great potential, but yet again, a working protection against identity theft as one of the biggest challenges of browser-based Single Sign-On systems remains still unsolved.

References

[BJM08] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery, 2008.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0, January 1999.

[FKK96] A. Frier, P. Karlton, and P. Kocher. The SSL Protocol Version 3.0, November 1996.

[FRHH07] Brad Fitzpatrick, David Recordon, Dick Hardt, and Josh Hoyt. OpenID Authentication 2.0, December 2007.

[GP06] Thomas Groß and Birgit Pfitzmann. SAML Artifact Information Flow Revisited. In *In IEEE Workshop on Web Services Security (WSSS)*, pages 84–100, Berkeley, May 2006. IEEE.

[Gro03] Thomas Groß. Security Analysis of the SAML Single Sign-on Browser/Artifact Profile. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC'03)*. IEEE Computer Society Press, December 2003.

[GSSX09] Sebastian Gajek, Jörg Schwenk, Michael Steiner, and Chen Xuan. Risks of the CardSpace Protocol. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC*, volume 5735 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2009.

[HBH07] Dick Hardt, Johnny Bufu, and Josh Hoyt. OpenID Attribute Exchange 1.0, December 2007.

[HDR06] Josh Hoyt, Jonathan Daugherty, and David Recordon. OpenID Simple Registration Extension 1.0, June 2006.

[Jam] Shakir James. Web Single Sign-On Systems.

[KR00] David P. Kormann and Aviel D. Rubin. Risks of the Passport Single Signon Protocol, 2000.

[NL08] Ben Newman and Shivaram Lingamneni. CS259 Final Project: OpenID (Session Swapping Attack), 2008.

[PW03] B. Pfitzmann and M. Waidner. Analysis of liberty single-sign-on with enabled clients. *Internet Computing, IEEE*, 7(6):38–44, 2003.

[TT07] Eugene Tsyrklevich and Vlad Tsyrklevich. Single Sign-On for the Internet: A Security Story, July and August 2007.