Security Evaluation of the OpenID Protocol

ALEXANDER LINDHOLM

Master of Science Thesis
Stockholm, Sweden 2009
Security Evaluation of the OpenID Protocol

ALEXANDER LINDHOLM

Master’s Thesis in Computer Science (30 ECTS credits) at the School of Computer Science and Engineering
Royal Institute of Technology year 2009
Supervisor at CSC was Mads Dam
Examiner was Johan Håstad

TRITA-CSC-E 2009:076
ISRN-KTH/CSC/E--09/076--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se
Security evaluation of the OpenID protocol

Abstract
To remedy the problem of the increasingly amount of authentication credentials the OpenID protocol is developed. The OpenID protocol implements a single sign-on solution for the Internet to help reducing the number of authentication credentials. Single sign-on means that users only need to use one set of authentication credentials to authenticate to several service providers. The OpenID protocol consists of two parts, the Relying Party and the OpenID provider. The Relying Party is an OpenID enabled service that uses OpenID to authenticate the users and the OpenID provider is performing the actual authentication task. This thesis evaluates the OpenID protocol with the AVISPA analysis tool. A model of the OpenID protocol must be created in the High-Level Protocol Specification Language to evaluate the protocol with the AVISPA tool. During the analysis of the OpenID protocol a security weakness was discovered. This weakness makes it possible for an attacker to impersonate a user. An implementation of the discovered attack has been created to show that the attack can be performed in a real environment. The OpenID protocol messages need integrity protection to avoid this attack. The messages can be integrity protected by applying the Secure Socket Layer protocol. The DotNetOpenID, which is an OpenID implementation, is analysed with the help of two static code analysis tools. These analysis tools are Gendarme and FxCop, which analyses code written in Microsoft’s .Net framework. These two tools do not indicate any serious security issues in the code of the implementation but the tools find other less serious problems.

Säkerhetsevaluering av OpenID-protokollet

Sammanfattning
Table of contents

1 Introduction ..................................................................................................................... 1
  1.1 Authentication............................................................................................................. 1
    1.1.1 Computer authentication ...................................................................................... 1
  1.2 Single sign-on ............................................................................................................. 2
    1.2.1 Security aspects of single sign-on ........................................................................ 2
    1.2.2 Pseudo-SSO versus Real-SSO ............................................................................ 2
    1.2.3 Kerberos .............................................................................................................. 3
    1.2.4 Microsoft Windows Live ID ................................................................................. 4
    1.2.5 Shibboleth ........................................................................................................... 5
  1.3 Evaluation of security protocols .................................................................................. 5
  1.4 Evaluation of security critical software ....................................................................... 6
    1.4.1 Software reviewing ............................................................................................... 6
    1.4.2 Software testing .................................................................................................... 7
  1.5 Problem definition ..................................................................................................... 7
  1.6 Project limitations .................................................................................................... 7
  1.7 Contribution ................................................................................................................ 7

2 Theory and Background .................................................................................................. 8
  2.1 Security protocol analysis with AVISPA .................................................................... 8
    2.1.1 High-Level Protocol Specification Language ....................................................... 8
    2.1.2 AVISPA back-ends .............................................................................................. 12
  2.2 Code analysis ............................................................................................................. 13
    2.2.1 FxCop ................................................................................................................... 13
    2.2.2 Gendarme ........................................................................................................... 16

3 The OpenID protocol ....................................................................................................... 21
  3.1 Description of the protocol ....................................................................................... 21
  3.2 Discovery process ...................................................................................................... 21
    3.2.1 Yadis protocol ..................................................................................................... 22
    3.2.2 XRI ..................................................................................................................... 22
  3.3 Abstract description of OpenID ................................................................................. 24
  3.4 OpenID authentication example .................................................................................. 24
    3.4.1 Association request message .............................................................................. 25
    3.4.2 Association response message ............................................................................ 26
    3.4.3 Authentication request message ......................................................................... 27
    3.4.4 Positive assertion message ................................................................................ 28
    3.4.5 Verifying the assertion message ......................................................................... 29
  3.5 Security concerns ...................................................................................................... 30
    3.5.1 Eavesdropping attack ......................................................................................... 30
    3.5.2 DNS attacks on OpenID ..................................................................................... 30
    3.5.3 Other types of attacks ....................................................................................... 30
    3.5.4 Counter methods against the attacks .................................................................. 30

4 Security analysis of the OpenID protocol ....................................................................... 31
  4.1 Experiment setup ....................................................................................................... 31
  4.2 Formal analysis of OpenID protocol ......................................................................... 31
# Table of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An overview of the Kerberos protocol</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Overview of the different components of the AVISPA tool [31]</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>An overview of the Cecil's framework [7]</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>Message overview of the OpenID protocol</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Example of an association request message</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Example of an association response message</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>Example of an authentication request message</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>Example of a positive assertion message</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>The message chart for the discovered attack</td>
<td>34</td>
</tr>
<tr>
<td>10</td>
<td>An overview of the implemented attack</td>
<td>35</td>
</tr>
<tr>
<td>11</td>
<td>A class diagram of the most important classes in the Relying Party module</td>
<td>37</td>
</tr>
<tr>
<td>12</td>
<td>A class diagram of the most important classes in the OpenID provider module</td>
<td>38</td>
</tr>
</tbody>
</table>
Introduction

The need for authentication has increased over the years in our digitalised world. Everyday people use some form of authentication to access resources. The resources could be a computer, cell phone, credit card terminal or other systems that uses various form of authentication. Numerous systems on the Internet uses authentication, typically password based. One problem that arises is the amount of different passwords a person needs to handle. For instance a company could have multiple applications that the user must authenticate to. Each of these applications has their own authentication method and authentication credentials that the user needs to remember. This increases the likelihood for users to forget their passwords or the other types of authentication credentials. To solve these types of problems an authentication system with single sign-on capability can be used to perform the authentication. This allows the users to have one set of authentication credentials to multiple systems. Authentication credentials could for instance consist of passwords, tokens or the credentials could consist of finger prints or voice recognition. However some problems occur when applying single sign-on on the Internet regarding administration and security. One administrative problem can be the decision of who should decide which user should be able to authenticate to the application. It is imperative that an authentication protocol is secured to ensure that the right user is authenticated and that a user cannot be impersonated by a malicious user. The user needs to trust that the authentication protocol can securely store their identities and the application also needs to trust the authentication system. If the authentication protocol could be attacked, the applications cannot trust the users’ identities. OpenID is a new authentication and single sign-on protocol for systems on the Internet. The OpenID protocol’s security aspect is analysed in this thesis to evaluate if the OpenID protocol have any security defects that are affecting the authentication. An implementation of the OpenID protocol is also analysed statically in respect to code conventions and security aspects. This means that the source code of the implementation is statically analysed for errors by static analysis tools. The static code analysis is done to evaluate if any security issues are detected in the source code of the OpenID implementation.

1.1 Authentication

The definition of Authentication is that you can in a trustworthy manner identify persons or objects identity [12]. In everyday life there are multiple examples of authentication between people. For instance by hearing a person’s voice you can tell if that person is known to you. In this example the authentication credential is the person’s voice. There are four different categories of authentication credentials: [4]

- Something you know, for example a password.
- Something you have, which can be a smart card
- Something you are, for instance fingerprint or iris pattern.
- Where you are, for example at a bank office or at a certain computer.

When using a computer system these categories of authentication credentials are also used.

1.1.1 Computer authentication

Computer authentication systems use one or more authentication credentials based on the categories above. For example passwords are common authentication credentials in computer authentication. The main problem is if the authentication credentials are compromised. For instance if a user’s password is exposed someone could impersonate that user. The authentication system needs to store authentication information in order to validate identifications. It is vital that the authentication system stores the authentication information securely. If the authentication credentials are sent over a network it is crucial to protect them from eavesdropping. According to Bishop [4] an authentication system consists of five parts:
• The set of the “authentication information” which prove the users identity
• The set of the “corresponding information” which the system must store to be able to validate the users
• The set of functions that uses the authentication information to generate the corresponding information.
• The set of functions that verifies the identity of the users.
• The set of functions used to edit the authentication and corresponding information.

1.2 Single sign-on
De Clercq defines single sign-on property as “the ability for a user to authenticate once to a single authentication authority and then access other protected resources without re-authentication” [9]. The single sign-on property is desirable because of the increasing amount of applications a user has to operate. The company SafeNet performed a study during 2004 to determine how people handled their passwords [26]. The number of respondents was above 2000. Half of the respondents answered that they wrote down their passwords to remember them. Thirty five percent of the respondents had once in their lifetime shared a password and the same percentage answered that they needed to use three to four passwords at work. Around half of the respondents answered that they had to reset a password for a business application at least once a year. The result of the study indicates that there exist problems with the amount of passwords that the users are exposed to. The single sign-on system only have one set of authentication credentials per user, which reduces these types of problems. The administration of a single sign-on system decreases because the system could be centralised managed and it could use a centrally storage of credentials [9]. The storage of credentials is not restricted by a physical location. By using a single sign-on system the cost of developing new software can be reduced since the application does not need to have its own authentication mechanism [27]. As another point of view the software application is locked to one single sign-on system.

1.2.1 Security aspects of single sign-on
There are some security aspects regarding single sign-on systems that need to be taken into account. If a security defect is found in a single sign-on system, all the implementations that use this system may be affected. Also when sharing the same authentication solution the applications also share the same level of security. The problem with sharing the same security level is that the different entities that use the single sign-on system could have different needs of security level. The central credential storage must also be protected to avoid exposing the credentials. If one user’s credentials are exposed this user could be impersonated to all the entities that uses the single sign-on system.

1.2.2 Pseudo-SSO versus Real-SSO
Single sign-on has two different branches, Pseudo-single sign-on and Real-single sign-on [20]. Pseudo- single sign-on does not implement a complete authentication solution. Pseudo- single sign-on has methods to store authentication credentials from multiple authentication systems and when the user access the authentication system the Pseudo-single sign-on system presents the right authentication credentials. The credentials can be associated to the right service when the user accesses the service for the first time and these authentication credentials need to be stored securely. The Pseudo-single sign-on system authenticate the user by performing the same steps automatically that the users originally needed to perform to be authenticated to the system. To access the stored credentials the Pseudo-single sign-on system usually uses one set of authentication credentials. The user only needs to remember the Pseudo-single sign-on system’s credentials. When using a Pseudo- single sign-on solution the different systems still need to have their own authentication solution and separate administration of the services.
Services and application that uses a Real- single sign system need to have the same authentication solution in order to be able to interact with the single sign-on system. The services and the applications do not need to have their own authentication solution. The impact of this is that systems need to be adapted to the single sign-on system. Real- single sign-on only has one set of credentials for the entities and the administrators only need to handle these. The Real- single sign-on takes care of the authentication between the entities and services. If a user wants to access a service, the user needs to authenticate to the single sign-on system. Then the single sign-on system sends authentication information to convince the service that the user is authenticated. The authentication information does not contain other authentication credentials such as passwords because there is only one set of the authentication credentials. There are numerous single sign-on solutions that use the Real- single sign-on approach. Kerberos, Windows Live ID and Shibboleth are systems that use Real- single sign-on and they also have centralised storage of the authentication credentials. The OpenID protocol is using the Real- single sign-on approach but it has a decentralised architecture.

1.2.3 Kerberos

Kerberos is an authentication protocol with single sign-on capability. The protocol was designed and built at Massachusetts Institute of Technology (MIT) for their Athena project [29]. Kerberos is designed for use in a single administrative domain and with a centralised authentication authority. The latest version of Kerberos, Kerberos V5, has the ability to be used over multiple administrative domains by establishing a chain of trust.

![Figure 1 An overview of the Kerberos protocol](image)

1.2.3.1 Description of Kerberos authentication

In the Kerberos protocol the Key Distribution Centre (KDC) is the name of the central authentication authority [12]. The Key Distribution Centre is responsible to generate the different tickets that are used in the Kerberos system. A ticket contains authentication information, which can be keys and other cryptographic data. In Figure 1 the authentication process is displayed.

The first two steps in Figure 1 describe the authentication to the Kerberos Key Distribution Centre. To start the authentication procedure the user enters a password into the system at the user’s workstation. A cryptographic key is then generated from the password by applying a hash function. This generated key is the user’s master key.
The first message in Figure 1 is then sent to the Key Distribution Centre, which include the user name of the user and step 2 shows the return message which include a new session key and a ticket granting ticket. The entire return message in step 2 is encrypted with the user’s master key to ensure that the user’s workstation is the only one that can read the message. When the ticket granting ticket and the session key has been decrypted, the user’s workstation does not need the master key anymore because every message is encrypted with the session key. The ticket granting ticket is used to establish new authentications with different entities and the ticket contains the session key, a timestamp when the ticket expires and the user’s name. The ticket granting ticket is encrypted with the Key Distribution Centre’s master key, which is only known to the centre.

Step 3 to 6 in Figure 1 describes the different steps performed when the user wants to obtain access to a new resource. In step 3 the user’s workstation sends a new message to the Key Distribution Centre containing the name of the resource the user wants to access, a timestamp and the ticket granting ticket. The timestamp is used to prevent replay attacks. When the Key Distribution Centre receives the message the centre needs to validate the message by validating the timestamp in the message and verify that the ticket granting ticket is valid.

If the ticket granting ticket is valid the Key Distribution Centre proceeds with step 4 in Figure 1. The Key Distribution Centre generates a new session key, which is used between the user’s workstation and the new resource. The Key Distribution Centre sends a return message to the user containing the new session key and a ticket encrypted with the session key between the Key Distribution Centre and the user’s workstation. The ticket contains the user’s name and the new session key encrypted with the new resource’s master key. The user’s workstation receives this message and decrypts it to obtain the new session key and the ticket.

Step 5 in Figure 1 the ticket and a timestamp encrypted with the new session key are sent directly to the resource. The resource decrypts the ticket and extracts the session key and the user’s name. The resource also decrypts the timestamp with the new session key and check if the timestamp is valid.

If the timestamp is valid step 6 in Figure 1 is executed. The resource sends the timestamp increased by one back to the user. The return message is encrypted with the new session key. The user’s workstation checks that the received timestamp is the one that was sent to the resource increased by one. The user is then authenticated to the new resource. The communication between the user and the new resource could be encrypted and integrity protected but it is up to the application or the configuration to decide the type of security [12].

1.2.4 Microsoft Windows Live ID

Windows Live ID is an authentication and single sign-on system for the web. Windows Live ID is developed by Microsoft and was first released 1999 with the name .Net Passport. In June year 2000 this system had 40 million users and over 400 authentication requests each second [13]. Windows Live ID uses a central storage for the authentication credentials. To use the system the users need to create an account at Microsoft. To verify the user’s identity a valid email address is required. The email-address is also the user’s login name. If a web service wants to use Windows Live ID the service need to acquire a cryptographic key from Microsoft. This key is used to decrypt authentication information sent by the user. The authentication information is encrypted with the triple DES algorithm.

1.2.4.1 Authentication of a user on a website

To initialise authentication of a user, the user needs to connect to a resource that uses Windows Live ID. The resource redirects the user to Microsoft’s authentication authority. The connection to the authentication authority is protected by the SSL protocol to ensure that the authentication credentials are protected. The user sends the authentication credentials to the authentication authority. If the credentials are valid the authentication authority answers by sending three messages, so called cookies. These cookies are: [19]
Introduction

- Ticket cookie: This cookie contains a unique identifier corresponding to the user and a timestamp.
- Profile cookie: Stores all the user information such as name, email and address.
- Page cookie: Stores all the pages where the user is signed in.

When the user has been authenticated to the authentication authority, the authority redirects the user back to the resource. The resource requests the ticket cookie and the profile cookie for validation of the user. If the timestamp in the ticket cookie is outdated the resource could reject the authentication attempt. The profile cookie is used to transfer user data to the resource. The page cookie is only used by the authentication authority when the user signs out from the system. The page cookie then informs what other resources this user is signed into.

1.2.4.2 Security concerns

The central storage of the authentication credentials is a weakness to the system because the central storage can be a target for denial of service attacks [19]. To encounter the denial of service attacks Microsoft has several authentication resources to make the system less vulnerable to the denial of service attacks. The Windows Live ID could be target to a so-called phishing attack. If a malicious entity could change the address of the redirects in the Windows Live ID protocol the users could be tricked to reveal the authentication credentials [13].

1.2.5 Shibboleth

Shibboleth is an open source authentication and single sign-on solution for the Internet. Middleware Architecture Committee for Education (MACE), the Internet2 Middleware Initiative is developing the specification of the Shibboleth protocol. Shibboleth version 1.0 was released in June 2003 and version 2.0 was released in March 2008. The specification of Shibboleth consists of three parts: [6]
- The authentication provider: the part that makes the authentications of the users
- The relying part: the part that is using Shibboleth to authenticate the users.
- The discovery part: helps the relying part to find the users authentication provider.

The Shibboleth protocol uses the Security Assertion Markup Language (SAML) to transfer the authentication information between the different parts.

1.2.5.1 Authentication with Shibboleth

The user is the part that initialises the authentication by connecting to a relying part. If the user is already authenticated to an authentication provider the relying part checks for the Shibboleth authentication information and the user is authenticated. There are two possible scenarios if the user is not authenticated to an authentication provider. If the relying part has knowledge of the user’s authentication provider the relying part can redirect the user directly to the authentication provider. If the relying part does not have the knowledge of the user authentication provider, the relying part needs to redirect the user to the discovery part [6]. The user selects the correct authentication provider at the discovery part and the user is redirected to that authentication provider. At the authentication provider the user is authenticated. What type of authentication technology used at the authentication provider is not specified by the Shibboleth protocol. It is up to each authentication provider to select the best authentication technique for that provider. If the authentication of the user succeeds the user is redirected back to the relying part. If the relying part needs more information about the user, the relying part can establish a direct connection with the authentication provider and request the missing information. Based on the information that the relying part has the user is authenticated or rejected.

1.3 Evaluation of security protocols

Security protocols are used in many critical systems and it is very important that the security protocols are secure [5]. The protocols can be analysed in many ways for instance by protocol experts or by using automated tools. Analysis of a security protocol is a complex task. Many
security protocols are discovered to contain security flaws even after protocol experts have
evaluated the protocol [10]. Evaluation of security protocols can be divided into two different
parts, where one part is trying to prove that the protocol is correct and the other one is trying to
find flaws in the design of the protocol. These two parts are related to each other because when
proving that the protocol is correct flaws can be detected. To prove that a protocol is correct,
different approaches can be used. One way of proving the correctness of a security protocol is to
create cryptographic security proof of the protocol to mathematically verify the protocol. The
cryptographic security proof approach analyse the underlying cryptographic functions in the
security protocol and calculate the probability for an intruder to attack the protocol [30]. Formal
model verification of the protocol and computational models can also be used to detect flaws in
the design of the security protocol. There exist different categories of formal models such as
logic models or inductive models. Formal methods can make the protocol more understandable
[10]. One of the first logics for protocol analysis is the BAN logic invented by Burrows, Abadi
and Needham. The BAN logic has rules to define objects and how protocol specific tasks are
expressed. BAN logic assumes that encryption cannot be broken. The BAN logic does not find
all flaws that can exist in a security protocol due to the design of the BAN logic [22]. If
different runs of a protocol are interleaved some security issues may not be detected by the
BAN logic. Inductive verification of the protocol could also be used to analyse the security.
Induction is used to verify security protocols in the induction models [21]. The Automated
Validation of Internet Security Protocols and Applications (AVISPA) tool uses logic to analyse
security properties of Internet protocols. Models of the Internet protocols must be created for
analysis with the AVISPA tool. The models are created in the High-Level Protocol
Specification Language.

1.4 Evaluation of security critical software
Evaluation is one important part of the software development. Evaluation is to verify that the
software meets the requirements and that the software is functioning correctly. A goal of
software evaluation is to locate critical errors in the software. The evaluation process can be
divided into a validation part and a verification part [28]. The validation part of the evaluation
should verify that the software customer’s expectations are fulfilled. The verification part
evaluates the software with the respect to the requirements and specification on the software.
There are different ways to perform evaluations of the software. Various methods are based on
reviewing the software and other methods are using different types of runtime tests. The
security issues are very important when evaluating the software and the security issues are often
overlooked [16]. Security testing is different from other types of testing because the tester needs
to think like an attacker [23]. The first evaluation methodology for security system is The
Trusted Computer System Evaluation Criteria (TCSEC) or Orange Book [4]. The TCSEC was
developed by the government in the USA to evaluate security in commercial software products.
The TCSEC defines how secure a product is by applying different criteria which could be
formal analysis of the source code and what type of access control the system uses. There exist
multiple other evaluation methodologies to evaluate security critical software for instance FIPS
and The Common Criteria.

1.4.1 Software reviewing
The first formal software reviewing process was developed at IBM during the 1970s [28].
Multiple software reviewing processes have been developed during the years. Software
reviewing includes inspection of every readable document, for instance source code,
requirement documentation and manuals to discover errors. The source code inspection is
performed to find logical errors, unnecessary complex code paragraphs and how the code is
structured. The software reviews could be done by auditors or by automatic software reviewing
tools. Automatic software reviewing tools cannot detect every error that can exist but it can
detect the most common errors. Formal methods could be used as a method of software
reviewing, which involve building a mathematical model of the software. The model of the
software then is analysed mathematically to discover if the software has errors and verify that the software is implemented as the specification stated [28].

1.4.2 Software testing
There are multiple different methods of software testing. One method is called system testing, which performs tests of whole system. System testing evaluates how well the software functions operate together. There is another method that tests smaller parts of a complete system, which is called unit testing. Unit testing checks that functions and classes behaves as expected by the specification. The unit tests also need to check how the methods behave when unexpected input is passed to the methods.

1.5 Problem definition
A formal security analysis of the OpenID protocol is performed in this thesis. To analyse the OpenID protocol an automated protocol analysis tool should be used. An implementation of the OpenID protocol is also analysed. The implementation must implement both parts of the OpenID protocol, the Relying Party and the OpenID provider. The following questions are answered:

- Are there any important security issues in the OpenID protocol?
- If any security issues are found, can the OpenID protocol be improved to prevent these attacks?
- Does the implementation analysis detect any security vulnerabilities?

1.6 Project limitations
This thesis is not looking at each implementation of the OpenID protocol. The chosen implementation must consist of both part of the protocol. The user interface is not analysed. The performance of the implementation is not analysed. The OpenID protocol is only analysed by one formal analysis tool.

1.7 Contribution
The OpenID protocol is formally analysed by an Internet protocol analysis tool called the AVISPA tool. AVISPA is an abbreviation of Automated Validation of Internet Security Protocols and Applications and the analysis is done by building a model of the protocol. The models are built in the High-Level Protocol Specification Language HLPSL. The model is then analysed by the tool and the analysis of the OpenID protocol detects a possible attack. An implementation of the attack is done to verify that the attack is possible in a realistic environment.

A static code analysis of an OpenID protocol implementation is done to evaluate the security of the implementation. DotNetOpenID is the OpenID protocol implementation that is analysed with two analysis tools. These analysis tools are FxCop and Gendarme and these tools are chosen because they can analyse the programming language that the DotNetOpenID implementation is written in. Each tool has rules to detect different types of problems, for instance both tools has rules to detect security issues with the source code. The outcome of the analysis wit the two tools is that the tools security rules did not report any errors in the source code.
2 Theory and Background

2.1 Security protocol analysis with AVISPA

Automated Validation of Internet Security Protocols and Applications (AVISPA) is an automatic analysis tool for security protocols. To perform an analysis of a protocol the AVISPA tool uses models of the security protocols. The AVISPA tool uses High-Level Protocol Specification Language (HLPSL) as a model language. [31]. HLPSL is a formal language that can model the different properties of security protocols. Figure 2 gives an overview of the AVISPA tool.

![Overview of the different components of the AVISPA tool](image)

2.1.1 High-Level Protocol Specification Language

The HLPSL is based on Lamport’s Temporal Logics of Actions (TLA) [1]. The goal of HLPSL is to provide a human readable and powerful language to specify the security protocols. When specifying the protocols in HLPSL, the protocols are divided into multiple roles. There are different types of roles, basic roles that consist of the agents in the protocol and the composed role that uses the basic roles to compose sessions. The role that is needed when specifying the protocol is the Environment role. The Environment role is the top-level role and this role is used to specify how the sessions of the protocol is interacting with each other. The HLPSL contains multiple types to be used when specifying protocols. Examples of these types are agents, symmetric_key, public_key, text, nat, function and channel. Symmetric_key and public_key types symbolise cryptographic keys. “Text” is used to symbolise text messages in the protocol such as nonce. Nat is symbolising natural numbers. Function types are used to represent cryptographic functions and hash functions. The channel type is used to specify communication channels to be used to the communication between the different roles. The dy variable in the channel parameter, showed in the examples below, tells the AVISPA tool to use the Dolev-Yao attack model.
The attack model used by the AVISPA tool is the Dolev-Yao model, which is the only one supported. In the Dolev-Yao model the intruder has full access to all messages sent and can modify and send any type of message given that the intruder has knowledge about the cryptographic material. The Dolev-Yao model has the property that the intruder is not able to attack the cryptographic functions. The intruder needs to have knowledge of the right keys used.

The multiple types are composed to create a role. An example of a role is given below:

```plaintext
role alice (A:agent, Na:text, K:symmetric_key, SND, RCV: channel(dy))
    played_by A def=
    local State:nat
    init State := 0
end role
```

This role has alice as name and requires five different parameters, an agent A, the nonce Na, the symmetric key K and the two channel parameters SND and RCV. The played_by variable indicates the agent that plays this role. Local variables are only visible inside the role and this role has only one such variable, namely the State variable. Init State indicates that the State variable should be set to the value specified. The role could also use transitions to change the state of the role. A transition could look like this in HLPSL:

```plaintext
role alice (A:agent, Na:text, K:symmetric_key, SND, RCV: channel(dy))
    played_by A def=
    local State:nat
    init State := 0
transition
    1. State = 0 /\ RCV(start) =|>
    State' := 1 /\ SND({Na}_K)
end role
```

The transition is fired if the State variable is equal to zero and the RCV channel is set to start to indicate that this role starts the communication. The =|> symbol indicates that the transition should be executed right after the conditions are met. The := symbol indicates that the State variable should be set to one and that the nonce value Na should be encrypted with the key K and transmitted on the channel SND. Priming variables indicates that the variables should be set to a new value.

To specify the security goals for the protocols a special section is provided. The security goals can consists of two types, secrecy_of and authentication_on. The authentication property needs to be specified in the basic roles by specifying a request and a witness. An example taken from the HLPSL Tutorial [31]:

```plaintext
role alice(A,B : agent, K : symmetric_key, Hash : hash_func, SND,RCV : channel(dy))
    played_by A def=
    local
    State : nat,
    Na,Nb : text,
    K1 : message
    init
    State := 0
transition
    1. State = 0 /\ RCV(start) =|>
    State' := 2 /\ Na' := new()
    /\ SND({Na'}_K)
    2. State = 2 /\ RCV({Nb'}_K) =|>
    State' := 4 /\ K1' := Hash(Na.Nb')
```

To specify the security goals for the protocols a special section is provided. The security goals can consists of two types, secrecy_of and authentication_on. The authentication property needs to be specified in the basic roles by specifying a request and a witness. An example taken from the HLPSL Tutorial [31]:

```plaintext
role alice(A,B : agent, K : symmetric_key, Hash : hash_func, SND,RCV : channel(dy))
    played_by A def=
    local
    State : nat,
    Na,Nb : text,
    K1 : message
    init
    State := 0
transition
    1. State = 0 /\ RCV(start) =|>
    State' := 2 /\ Na' := new()
    /\ SND({Na'}_K)
    2. State = 2 /\ RCV({Nb'}_K) =|>
    State' := 4 /\ K1' := Hash(Na.Nb')
```
role bob(A, B : agent, K : symmetric_key, Hash : hash_func, SND, RCV : channel(dy))
played_by B def=
local
State : nat,
Nb, Na : text,
K1 : message
init
State := 1
transition
1. State = 1 /\ RCV({Na’}_K) =>
State’ := 3 /\ Nb’ := new()
/\ SND({Nb’}_K)
/\ K1’ := Hash(Na’.Nb’)
/\ secret(K1’, k1, (A, B))
2. State = 3 /\ RCV({Nb}_K1) =>
State’ := 5 /\ request(B, A, bob_alice_nb, Nb)
end role

role session(A, B : agent, K : symmetric_key, Hash : hash_func) def=
local SA, SB, RA, RB : channel (dy)
composition
alice(A, B, K, Hash, SA, RA)
/\ bob (A, B, K, Hash, SB, RB)
end role

role environment() def=
const
bob_alice_nb,
k1 : protocol_id,
kab, kai, kib : symmetric_key,
a, b : agent,
h : hash_func
intruder_knowledge = {a, b, h, kai, kib}
composition
session(a, b, kab, h)
/\ session(a, i, kai, h)
/\ session(i, b, kib, h)
end role

goal
secrecy_of k1
The example above is modelling an authentication protocol that is shown below:

A \to B: \{Na\}_K
B \to A: \{Nb\}_K
A \to B: \{Nb\}_{K1}, \text{ where } K1=Hash(Na.Nb)

A is modelled by the role alice and B is the role bob. The authentication goal is that A should be authenticated to B. To do that a pre-shared key is used and a new key is generated to validate that both A and B knows the shared key. This example has both the secrecy and the authentication goal specified as shown in the goal section of the protocol specification. The witness and request statements in the example is used to check that a role has right to believe that the role is speaking with a certain peer, that the peer has reach the specified state and that the peer agrees on a value [31]. The agent A has a witness statement on line 17 and the witness statement is written like this: \text{witness}(A,B,bob\_alice\_nb,Nb'). The witness should be interpreted as agent A agrees that the value Nb' should be used in an authentication attempt and that agent B wants to peer with agent A by the protocol id bob\_alice\_nb.

On row 36 the request(B,A,bob\_alice\_nb,Nb) parameter is used. The request should be interpreted as agent A trusts that the agent B exists and that the value Nb is agreed by both agent A and B.

The witness and request statement is then used to specify the authenticate_on goal at line 64.

When specifying the secrecy parameter the k1 in this example should not be known to any one except role A and B. The secret is specified at line 34. The secret function is used to specify secrecy goals. The session role is an example of a composite role and decides how the two basic roles act together. The environment role is the top-level role and it cannot take any arguments. The intruder_knowledge in the environment role specify what information the intruder knows. The i variable in the environment specification is the intruder.

In the example above both alice and bob roles are issuing the new() function on line 12 and 31. This function indicates that the variable should be set to a new value.

The AVISPA tool is translating the HLPSL into the Intermediate Format (IF) with the HLPSL2IF translator [32]. This is shown in the Figure 2. The Intermediate Format is a more low level language and is inserted into one of the four back-ends that the AVISPA tool uses. The protocol messages in the intermediate format language are modelled as atomic messages, pair messages or encrypted messages [3]. The intermediate format language specifies the protocol as an infinite-state transition system [2].

### 2.1.1.1 Logic behind HLPSL

The HLPSL is based on Lamport’s Temporal Logics of Actions and an example of how a HLPSL role is translated into the Temporal Logics of Actions is given below. The HLPSL role’s name is bob and the specification is given below:

```plaintext
role bob (A: agent, K: symmetric_key, SND, RCV: channel(dy))
  played_by A def=
  local State:nat, Na:text
  init State := 0
 transition
  1. State = 0 /\ RCV(start) /\ Na’ := new() =|>
     State’ := 1 /\ SND({Na’}_K)
end role
```
The role of Bob is not part of a protocol specification. The role is created to show how the HLPSL is translated to logic. The role of Bob consists of one transition there line 6 describe the start of the role and creates a new nonce value and then on line 7 sends the nonce encrypted with a key. The first step to translate this role into logic is to define the initial predicate. For Bob role the initial predicate is

\[
\text{Init}_\text{Bob}(\text{State}) \triangleq \text{State} = 0
\]

The initial predicate means that in the initial state the State variable is zero. The two transitions steps need to be translated into logic. The logic of the first transition is displayed below

\[
\begin{align*}
\text{Bob}_\text{Step1}(\text{State},\text{SND},\text{SND}_\text{flag},\text{RCV},\text{RCV}_\text{flag},\text{Na},K) & \triangleq \\
& \land \text{State} = 0 \\
& \land \text{RCV}_\text{flag}' \neq \text{RCV}_\text{flag} \\
& \land \text{RCV}' = \text{start} \\
& \land \text{Na}' \in \text{Msg} \\
& \land \text{Fresh(}\text{Na}'\text{)} \\
& \land \text{UsedNonces}' = \text{UsedNonces} \cup \{\text{Na}'\} \\
& \land \text{SND}' = \text{Crypt}(K,\text{Na}') \\
& \land \text{toggle(}\text{SND}_\text{flag}\text{)} \\
& \land \text{State}' = 1
\end{align*}
\]

The \&-symbol indicates that the different predicates are conjunctions. The three first predicates indicate the received message. The State variable must be set to zero. The RCV variable must be equal to the string start. The new Na variable, Na’, must be of the type message and a new value of the Na’ variable is generated by calling the Fresh function. The new generated Na’ value must not exist in the UsedNonces set because the generated nonce variable must be unique. The SND’ variable is set to the encrypted Na’ variable generated by the Crypt function. The State’ variable is changed to one in the last predicate.

The Next_Bob specify how the intruder affects the role and what the intruder can do and is shown below

\[
\begin{align*}
\text{Next}_\text{Bob}(\text{State},\text{SND},\text{SND}_\text{flag},\text{RCV},\text{RCV}_\text{flag},\text{Na},K) & \triangleq \\
& \land \text{Manipulate} \\
& \lor \text{Impersonate(}\text{RCV},\text{RCV}_\text{flag}\text{)} \\
& \land \text{Alice}_\text{Step1}(\text{State},\text{SND},\text{SND}_\text{flag},\text{RCV},\text{RCV}_\text{flag},\text{Na},K) \\
& \land \text{Divert(}\text{SND},\text{SND}_\text{flag}\text{)}
\end{align*}
\]

The Manipulate function specifies how the intruder could act. For instance the intruder could decrypt message if the intruder knows the key. The intruder could also generate new messages and apply functions to a message. The intruder always has full control over the different channels according to the Dolev-Yao intruder model. The logic is then translated into the Intermediate Language for analysis in one of the four back-ends.

### 2.1.2 AVISPA back-ends

As shown in Figure 2 the AVISPA tool uses four different back-ends that are able to check the protocol model. The four different back-ends are the Constraint-Logic-based Attack Searcher (CL-AtSe), the On-the-fly Model-Checker (OFMC), the SAT-based Model-Checker (SATMC) and the Tree Automata based on Automatic Approximations for the Analysis of Security Protocols (TA4SP).

The Constraint-Logic-based Attack Searcher uses a method called “lazy intruder” that uses symbolic terms to reduce the search space by limiting the amount of generated messages the Dolev-Yao intruder can generate [32]. The lazy intruder uses message variables and constraints to ensure that the changes do not affect the attacks or introduce new attacks. The CL-AtSe is modular to make the back-end simple to extend with more functionality.

The SAT-based Module checker takes the intermediate format language model and translates the model to a propositional formula. The propositional formula is then given to a SAT solver. A SAT solver is a tool that solves the satisfiability problem. The SATMC uses the DIMACS
interface to communicate with the SAT solver. DIMACS is a standard for SAT problems and this makes it possible to change the SAT solver used by SATMC.

The TA4SP back-end uses an approximation of the intruder knowledge to verify the protocol [32]. Rewrites and regular tree languages are used to analyse the protocol.

The OFMC back-end uses the IF language to analyse the protocol by traversing the transition system [3]. The transition system is modelled as a tree and the tree could be of infinite size. To handle this problem of infinity the OFMC uses a lazy infinite approach. The lazy infinite approach uses an infinite tree as a data-type in a lazy programming language, such as Haskell as the OFMC is written in [3]. The OFMC then searches the infinite tree to find an attack state. If an attack state is found, the OFMC stops the search and the attack trace is returned. As the CL-AtSe, the OFMC uses a lazy intruder.

2.2 Code analysis

If the source code successfully runs through the compiler the result is an executable program. The compiler checks the source code for errors, for example type mismatch and syntax errors. However the compiler cannot check all types of errors that can exist in the code. One example where the compiler does not detect errors is the gets method in the C language [23]. The gets method is constructed to read input into a provided buffer until an end-of-file or new line character is reached. The method does not check the size of the buffer and therefore this method can be exploited to do a buffer overflow. To solve this type of errors the code must be analysed further. Code analysts who audit the code can find these bugs. The code analysis takes time to perform and the analysts need to have special knowledge of common errors in the source code [8]. Therefore static code analysis tools have been developed. The static code analysis tools cannot find every error in the source code because the outcome depends on how the tools are constructed and what types of errors they detect. There are different types of analysis tools.

- One type of tools only analyse the methods in the source code. They do not analyse the connection between the different methods. This is called local analysis.
- Other tools can analyse the connection between classes and methods in the source code. This is called global analysis.

The downside of using more complex analysis is that it takes more time to evaluate.

2.2.1 FxCop

FxCop is a static code analysis tool developed by Microsoft. The FxCop is free to use and analyses assemblies written in the .Net framework. The assemblies contain Common intermediate language (CIL), which is generated when compiling a .Net program. Common intermediate language is the code that the virtual machine uses. When using the Common intermediate language the FxCop can analyse the code from different languages such as C#, VB.NET and all the other supported languages. This tool is mostly aimed for class library developers to detect errors against the .Net framework’s programming and design rules [17]. FxCop is intended to be an integrated part of the development cycle and to be used in an iterative way. To find errors as early as possible the tool should be executed every time new or updated source code is inserted into the assembly. The FxCop analysis tool generates a report after the analysis of the assembly. The report consists of all the errors the tool has discovered during the analysis. These warnings consist of information about what rule generated the warning, the importance of the warning, the cause of the warning and a description of the problem.

To analyse the assemblies FxCop uses introspection [14]. Introspection allows the tool to inspect the metadata in an assembly file. The information gathered from the assembly is divided into different nodes by the FxCop tool. These nodes are hierarchy ordered and are shown in the list below taken from [14]:

- Node
The introspection mechanism has one class for each of these nodes to build an abstract syntax tree representation of the assembly. The FxCop analysis tool traverses the nodes and executes each rule on each node during the analysis of an assembly. The rules are responsible for analysing the nodes to find the errors the rule is specified to detect in the assembly. FxCop could also build a call graph for a method if it is necessary when creating a rule. Kresowaty, J [14] has an example of what a rule could look like:

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.FxCop.Sdk;

namespace TutorialRules
{
    class UseGenericList : BaseIntrospectionRule
    {
        private TypeNode m_ArrayList;

        public UseGenericList() :
                typeof(UseGenericList).Assembly)
        {
        }

        public override void BeforeAnalysis()
        {
```
The goal of this rule is to detect the ArrayList variables and proposes to exchange them to generic lists instead. Every rule needs to inherit the class BaseIntrospectionRule as shown on line 9. On line 19 the rule overrides the BeforeAnalysis method, which is called once per thread during the analysis. In the BeforeAnalysis method everything that needs to be setup before the test could start should be done. In this example the parameter m_ArrayList is initialised in the BeforeAnalysis method. The other method in this example class is the Check method at line 26. The FxCop analysis tool calls the Check method when the rule should execute its test. If a problem is found it is added into the Problems parameter as shown in line 35 and 48. The rules are divided into multiple categories in the FxCop analysis tool. These categories are displayed in Table 1.
Categories in FxCop

Design category
Globalization category
Naming category
Mobility category
Performance category
Interoperability category
Security category
User category

Table 1 shows the different categories of rules in the FxCop tool.

**Design category** - The design category contains rules about code design. For instance there are rules to check that there should not be any empty interfaces that exceptions should be public and that indexer should be of the type integer or string.

**Globalization category** - The globalization group contains rules that analyse how differences between different countries should be handled. One rule checks that the file path should not be written directly into the source because different language versions of windows use different paths. Naming convention rules checks the names of the identifiers. The rules check if the names have the right case. There is also one rule that check if the names differ from each other by more than case.

**Naming category** - The naming category contains rules to detect errors regarding class names, variable names and method names. For instance one rule checks that no enums do have names that contain the word “Reserved”. Another rule checks that the class names do not match the namespace name.

**Mobility category** - The mobility category has rules that check different mobility errors. One rule checks that the process priority is not set to idle to prevent unnecessary CPU utilisation. Another rule checks how the different timers are used to keep the power consumption as low as possible.

**Performance category** - The performance category contains rules for checking if the source code has unnecessary statements that make an impact on performance. It checks if the assembly contains unused private code. One rule also checks the comparison between empty strings instead of using the length of the string, which is more effective.

**Interoperability category** - The interoperability rules are used to check if the assembly is compatible with other standards. To check if the assembly can be transferred to other platforms, such as 64-bit windows, the portability rules are used.

**Security category** - The security rules detect if the source code is violating security constraints. One rule checks that the unmanaged code contains only private pointers. Another rule checks for certain race conditions, which can occur in unmanaged code. In unmanaged code the user has more control of the code for instance the user can manipulate memory pointers. There are also rules to check how the different method calls are done with the respect to the security. For instance methods that require a security level need to do a stack walk to check the call structure.

**User category** - The user category, which is the last category, has rules to detect faulty behaviour when using the code. There is one rule that control if the return values are checked from a method call. One rule also checks if the index operators can perform an overflow of the array.

2.2.2 Gendarme

The Gendarme tool is developed to analyse the source code in the Mono project. The Mono project creates a .Net framework for Linux operating systems. Gendarme is open source and free to use. The user can write own rules to detect special errors in the source code. Gendarme uses, as FxCop, an introspection technique and Gendarme’s introspection framework is called Cecil [11]. Cecil is a framework to inspect the Common intermediate language. In Figure 3 a UML diagram is shown over the Cecil’s framework. The Cecil framework builds a Module Definition collection of the metadata in the assembly file. The rules are based on four different rule types, Assembly rules, Module rules, Type rules and Method rules. When creating a new rule, the new rule needs to inherit one interface from the rule type that matches best. An example rule is shown below:

```csharp
using System;
using Mono.Cecil;
using Gendarme.Framework;
using Gendarme.Framework.Rocks;

namespace Gendarme.Rules.Performance {
    public class AvoidReturningArraysOnPropertiesRule : Rule, IMethodRule {
        public RuleResult CheckMethod (MethodDefinition method) {
            if (!method.IsGetter)
                return RuleResult.DoesNotApply;

            if (!method.ReturnType.ReturnType.IsArray ()
                return RuleResult.Success;

            Runner.Report (method,
                Severity.Medium,
                Confidence.Total);
            return RuleResult.Failure;
        }
    }
}
```

**Figure 3 An overview of the Cecil’s framework [7]**

The Gendarme tool is developed to analyse the source code in the Mono project. The Mono project creates a .Net framework for Linux operating systems. Gendarme is open source and free to use. The user can write own rules to detect special errors in the source code. Gendarme uses, as FxCop, an introspection technique and Gendarme’s introspection framework is called Cecil [11]. Cecil is a framework to inspect the Common intermediate language. In Figure 3 a UML diagram is shown over the Cecil’s framework. The Cecil framework builds a Module Definition collection of the metadata in the assembly file. The rules are based on four different rule types, Assembly rules, Module rules, Type rules and Method rules. When creating a new rule, the new rule needs to inherit one interface from the rule type that matches best. An example rule is shown below:

```csharp
using System;
using Mono.Cecil;
using Gendarme.Framework;
using Gendarme.Framework.Rocks;

namespace Gendarme.Rules.Performance {
    public class AvoidReturningArraysOnPropertiesRule : Rule, IMethodRule {
        public RuleResult CheckMethod (MethodDefinition method) {
            if (!method.IsGetter)
                return RuleResult.DoesNotApply;

            if (!method.ReturnType.ReturnType.IsArray ()
                return RuleResult.Success;

            Runner.Report (method,
                Severity.Medium,
                Confidence.Total);
            return RuleResult.Failure;
        }
    }
}
```
This rule is taken from the source code of Gendarme and the rule is checking if a property is returning an array. This rule is inheriting the IMethodRule interface as shown on line 9. The CheckMethod method is used to execute the checking of the error in methods. On line 23 an error is detected in the source code and the result is returned as a failure.

Gendarme divides the different rules into categories, similar to FxCop. In Gendarme, there are more categories than in FxCop. In Table 2 the categories are listed.

**Table 2 show the names of the categories in Gendarme**

**Bad practice** - These rules try to detect bad practice in the source code. Examples of this can be that the source code calls potentially dangerous code. This can be `System.GC::Collect()`. Another example is that the user code creates a new exception without throwing it. There is also one rule to check that the constructor does not call a virtual method.

**Concurrency** - These rules deal with how to find potentially risks when multiple threads access data in variables and how to avoid deadlocks to appear. There are rules that check if the locks have been taken correctly. One rule check if there are any public static fields that has not been initialised and is not set to read-only. This can be a problem in a multithreaded environment.

**Correctness** - This rule category contains rules for checking that code is used in a correct way. One rule check the source code for classes that only declare static members because these types of classes should be declared as static or have private constructors. Another rules check that two floating point values are not compared because there could be precision problems. There is also one rule that check if some methods could be converted to static if they do not access anything from the instance. There are also rules to check variable assignment.

**Design** - This category has rules that check the design of methods and classes. One rule in this category checks if the class variables have accessor methods. Another rule checks that sealed classes do not have protected methods because sealed classes cannot be inherited. There is one rule to check that there are no empty interfaces. Another rule checks if operators are overloaded in pairs. For instance, if the subtraction operator is overloaded the addition should be overloaded as well.
**Design generic** - These rules check that the design of the generic types is correct. One example is that generic methods should take a generic parameter to make it easier for users of the method to understand what it does. Another rule check that the code uses generic event handles.

**Design linq** - In the last version of the .Net framework linq is included. Linq adds a new feature that is called extension methods. With extension methods a programmer could add new methods to old classes without change the original class. There is one rule in this category to check that there is no extension method that extends System.Object.

**Exception** - This category has rules to check that the exception handling of the code is correct. One example is that when writing a try-catch section the catch declaration should not catch a general exception. Instead the catch declaration should use a more specific exception. Another rule checks that reserved exceptions are not thrown. The reserved exceptions are ExecutionEngineException, IndexOutOfRangeException, NullReferenceException and OutOfMemoryException. These exceptions should only be thrown by the runtime.

**Interoperability** - These rules in the interoperability category check to see if the code is compatible with the 64-bit platform. For instance one rule checks that the code does not cast pointers into a 32-bits or smaller value.

**Maintainability** - This category has rules to check the complexity of the code and tries to make it easier to maintain the code. One rule in this category checks that the method IsNullOrEmpty in the string class is used instead of comparing the string object with null and comparing the length with zero.

**Naming** - This category has rules for checking that the names of types and namespaces are following the .Net framework standard. An example of a rule is that the names should not contain non-alphanumeric characters. There are also rules to detect redundancy in the naming of methods and in type names. Another rule checks the enumerable names so that they follow the standard.

**Performance** - The performance category has rules to detect errors that decrease the application performance. The rules can for instance detect problems that to see if a string is empty the programmer should check the length of the string. There are also rules to check that class variables and method parameters are used. Another rule checks that any private declared code, which is not used exist.

**Portability** - These rules check that the program can be transferred from one location to another. There are rules to check that the folder paths are not statically written into the code. Because Mono is used on Linux and Unix system there are some rules that check the portability for these systems.

**Security** - This category checks for security issues in the code regarding the different access categories. The access categories are public, private and protected. In this category there is one rule that checks if class variables are declared public because the class has not the control over them. Another rule checks that no array is declared public and read-only because the values in the array are still changeable.

**Security CAS** - Security CAS category checks that the code access attributes are correct. The code access attributes are set to specify the code rights. This means that the code can have security limits for example to access files on the hard drive. There are rules to check that a link security demands needs to have an inheritance demand if the type is not sealed.

**Smell** - This category has rules to check if it do exist duplicate code in the same class. There are also rules to check the size of the classes and methods. There is also one rule that checks if the parameter lists of the methods are not too many.

**Serialization** - The serialization category has rules to check serialization issues. One rule checks that the ISerializable interface is implemented correctly and checks that derived types call the base types constructor and method. There are also rules that checks if serialization attributes is set on correct methods.
ui - This category has rules to check the user interface. There are rules to check that the application is compiled with the right arguments for a windows form application. For instance when compiling Windows form applications the compiler should have the parameter – target:winexe. Another rule checks that the program has the right entry point.
3 The OpenID protocol

Brad Fitzpatrick took the initiative of creating the OpenID protocol and in 2005 the first version of the OpenID protocol was released [25]. The OpenID protocol is a single sign-on solution for the Internet. The first implementation of the OpenID protocol was made for the Internet site Livejournal.com. The second version of the protocol came in year 2007 and this version is analysed in this thesis. The OpenID protocol only uses standardised technologies, for instance the protocol uses HTTP to transfer the messages [18]. There are two parts of the OpenID protocol, the OpenID provider and the Relying Party. The OpenID provider performs the authentication of the user. The Relying Party is an OpenID enable service that uses OpenID protocol to authenticate the users. This separation of the OpenID protocol allows the users to choose their own OpenID provider to handle their identity.

3.1 Description of the protocol

To use the OpenID protocol the user needs to choose their OpenID provider and create an account at that OpenID provider [18]. To begin the authentication process the user accesses a Relying Party, which the user wants to authenticate to. At the Relying Party the user enters an OpenID identifier. The OpenID identifier must be an URL or an extensive resource identifier (XRI) and the OpenID identifier must be unique for the user. The Relying Party uses the OpenID identifier to discover the OpenID provider associated with the OpenID identifier. The discovery process uses the Yadis protocol or HTML documents. When the Relying Party locates the OpenID provider, the user is redirected to that provider. To redirect the user, the Relying Party sends an OpenID authentication request as a HTTP redirect message. The OpenID provider receives the OpenID authentication request and authenticates the user. The authentication method is not specified by the OpenID protocol. The OpenID provider is able to choose the authentication method. The OpenID provider redirects the user back to the Relying Party with an OpenID authentication response message, which tells the Relying Party if the authentication of the user succeeded. The Relying Party needs to verify the signature in the authentication response. To do that the Relying Party could use a pre-exchanged key that is negotiated with the OpenID provider. If there is no key exchange the Relying Party needs to send the authentication response to the OpenID provider. If the authentication response message is verified the user is authenticated.

3.2 Discovery process

The Relying Party performs the discovery process. The Relying Party can use three different methods to discover the OpenID provider. The three discovery methods are:

- If the user has entered an extensive resource identifier (XRI) an extensible resource descriptor sequence (XRDS) document should be used to do the discovery.
- If the user has entered an URL the Yadis protocol should be used.
- If the user has entered an URL but the Yadis protocol cannot be used the Relying Party should use the HTML document located at the URL.

If none of these methods are available the Relying Party cannot find the OpenID provider and the authentication process cannot go through. The extensible resource descriptor sequence (XRDS) is an XML document that stores the OpenID provider information. If the HTML document is used, the document must have special information about the OpenID provider. The information should be stored in the head section of the document.
3.2.1 Yadis protocol

The Relying Party uses the Yadis protocol to find the OpenID provider from an identifier. HTTP is used by the Yadis protocol as transfer protocol [33]. A Yadis ID is an identifier that can be resolved to an URL and the Yadis ID is necessary to start the Yadis protocol. When a service, such as the Relying Party, wants to use the Yadis protocol to get the XRDS document, the service needs to send an HTTP request to the Yadis ID. To the HTTP request there could be three different responses. The response could contain the XRDS document, or the response contains an HTML document with a link to the XRDS document or the response could contain a header field with the link to a XRDS document. If the response contains a HTML document, the HTML document needs to have a meta element in the header section. The meta element needs to have the http-equiv attribute set to X-XRDS-Location and the content attribute specified to a location address to the XRDS document. If the response contains a HTML document, the XRDS document in the HTTP response header needs to be X-XRDS-Location. If the response contains the link to an XRDS document the service that uses the Yadis protocol needs to do a new HTTP request to the XRDS document’s location. When the XRDS document is received the Yadis protocol is finished.

3.2.2 XRI

Extensive resource identifier (XRI) is an infrastructure like the Domain Name System (DNS). The XRI is used to identify objects such as users and computers [25]. The XRI address is based on the so-called I-names. An I-name begins with a global context symbol, which are the equal sign (=), the at-sign (@) and the addition symbol (+). The equal sign should be used for individuals, the at sign should be used for companies and the addition symbol is used for identify general objects. The XRI addresses can be transformed into three different forms. If the XRI I-name is @company the address could take the form of only the I-name, the fully qualified form xri://@company or in the proxy resolution form http://xri.net/@company. Xri.net is a global resolver for the XRI I-names. XRI uses the XRI resolution protocol to find the XRDS document. The XRI resolution protocol uses the Yadis protocol to locate the XRDS document.
The OpenID protocol

1. Access RP
   Client (C) → Relying Party (RP)

2. RP does the OP discovery
   Relying Party (RP) → OpenID provider (OP)

Optional steps

3. Association Request
   AssocReq(T,P,G,DH_{op})

4. Association Response
   AssocRes(AH,T,DH_{op}, H)

5. Authentication Request
   AuthReq(OP,RP,ID)

6. C authenticate to OP
   (Not specified by the OpenID protocol)

7. Positive Assertion
   PosAssert(RP,OP,ID,N,{Sig})

Optional steps. They are required if the association step is left out.

8. Verification Request

9. Verification Response

Figure 4 Message overview of the OpenID protocol
3.3 Abstract description of OpenID

As shown in Figure 4 there are three different parts involved during an authentication: the client (C), the service provider called Relying Party (RP) and the identity provider called OpenID provider (OP). C is in most cases a web browser controlled by a user. The user wants to get access to a service provided by the RP. RP asks C to authenticate to OP. OP then tries to authenticate C and issue a response to RP. RP receives the response and grants C access. RP initialises the OpenID protocol. There are two message types for RP-to-OP and OP-to-RP communication. RP could communicate with the OP directly or by redirects depending on what type of message it sends. Direct messages are always responded to by direct messages from OP and the same with the redirect messages.

In step 1, showed in Figure 4, C request to be authenticated to a service located at RP and in step 2 RP tries to locate the OP by using one of the discovery techniques described. Step 3 and 4 in Figure 4 are optional but recommended by the OpenID specification [18]. A key is exchanged in step 3 and 4 with the Diffie-Hellman algorithm. More information can be found in RFC 2631. In step 3 the RP sends an Association Request as a direct message to OP. The Association Request message has the following form AssocReq(T,P,G,DHOP) where T is the type of association algorithms used, P is the modulus prime number, G is the Diffie-Hellman generator and the DHOP is the RP’s public key. OP responds with an Association Response, which is shown in step 4 in Figure 4. The Association Response has the following form AssocRes(AH,T,DHOP,H), where AH is the identifier of the association, T is the type of association algorithms used, DHOP is the OP’s public key and H is a hash value of the created key XOR with the MAC key to transport the secret MAC key encrypted. The association steps are done to generate the key K, to be used as a session key between RP and OP. The AH value is stored at both the OP and the RP to be used to locate the associated key to generate the signature of the positive assertion message.

Step 5 in Figure 4, RP sends a redirect message to C, which is called Authentication Request of the form AuthReq(OP, RP, ID) where ID is the user’s identity. C redirects the message to the OP. In step 6 the OP tries to authenticate the user. If the authentication succeeds the OP sends a positive assertion message, which is shown in step 7 in Figure 4. The positive assertion message is also a redirect message as the authentication request message. The positive assertion message has the form PosAssert(RP, OP, ID, N, {Sig}) where {Sig} is an generated signature by hashing the concatenation of the RP, OP, ID, N and the key K. If the association steps have been left out the key K is generated by the OpenID provider. N is a random value called nonce. This message is redirected to RP. RP then needs to verify the signature.

If there was a key exchange in the beginning RP can verify the signature. If there was no key exchange, RP needs to send the signature back to OP for verification. In step 8 in Figure 4 the RP sends a Verification Request to the OP. OP must verify the signature of the message. Step 9 is executed after the OP has analysed the signature of the message to inform the RP the outcome of the verification of the message.

3.4 OpenID authentication example

To clarify the OpenID authentication process an example is given. The example explains all the different messages sent between the Relying Party and the OpenID provider. In the example the user has the following identifier http://user.example.com. The Relying Party uses the following address http://application.example.com and the OpenID provider has the following address http://www.example.com/openid.

The discovery process is the first step the Relying Party takes to authenticate the user, which is step 2 in Figure 4. To perform the discovery the identifier needs to be normalised before the discovery process can start. If the identifier starts with an XRI global context symbol the string should be treated as an XRI identifier. If the string starts with xri:// prefix the xri:// must be erased. If the string does not start with a XRI global context symbol or xri:// prefix the string should be treated as a URL. After the discovery process is finished and the Relying Party has a
location to an OpenID provider, the Relying Party could do an association with the OpenID provider by sending an association request. The association between the Relying Party and the OpenID provider is optional but recommended by the OpenID 2.0 specification [18].

3.4.1 Association request message

Step 3 in Figure 4 the association request message is sent to the OpenID provider. The association helps the OpenID provider to reduce the computations because the OpenID provider does not need to generate a new key for every authentication request and the authentication process is performed faster because the Relying Party does not need to ask the OpenID provider to verify the authentication response message. The association request message has the form AssocReq(T,P,G,DH_RP) as described in section 3.3. This message must be sent as an HTTP post message as it is a direct communication between the Relying Party and the OpenID provider. The information in the message must be sent as a HTTP post message to the OpenID provider. An example of the association request message is shown in Figure 5.

The first four lines in Figure 5 are parameters of HTTP message. The first line indicates that the message has the Post type. The first line also indicates that the path at the host should be /openid and the version of the HTTP message is 1.1. The second line in Figure 5 contains the Host parameter, which store the host name of the OpenID provider. The third line in Figure 5 specifies what type of content the message contains. The association request message content must be URL-form-encoded. The fourth line in Figure 5 specifies the length of the message content.

The next three lines in Figure 5 contain the OpenID protocol parameters. The association request message has seven different OpenID parameters and the first two must be included in every message. The first parameter, openid.ns, is used to indicate the version of the OpenID protocol that the Relying Party uses. The value of the openid.ns variable in Figure 5 indicates that the version of the protocol is 2.0. The second variable is the openid.mode and this variable indicates the type of the OpenID protocol message. The value of the openid.mode variable indicates that the message type is an association request.

The next five OpenID protocol parameters are specific to the association request message. The T parameter in the AssocReq(T,P,G,DH_RP) shown in step 3 in Figure 4 consists of two different parameters, the openid.assoc_type and openid.session_type.

The openid.assoc_type parameter describes the type of algorithm that the Relying Party wants to use to sign the message. In Figure 5 the openid.assoc_type is the third parameter and this variable has two defined values HMAC-SHA1 and HMAC-SHA256. The fourth parameter is openid.session_type, which is used to indicate what algorithm the Relying Party uses when generating the keying material. The openid.session_type parameter has also two defined values, DH-SHA1 and DH-SHA256. These defined values indicate that the Diffie-Hellman algorithm should be used to generate the key and the SHA1 or SHA256 hash function should be used to send the MAC key encrypted.
The fifth parameter is the openid.dh_modulus parameter which is also the P parameter in the AssocReq(T,P,G,DH_RP). The openid.dh_modulus parameter value is a big prime number and is used to generate the key with the Diffie-Hellman algorithm.

The G parameter in the AssocReq(T,P,G,DH_RP) consists of the openid.dh_gen parameter, which is the sixth parameter in Figure 5. The openid.dh_gen also contains a number used by the Diffie-Hellman algorithm to generate the key.

The seventh and last parameter in Figure 5 is the openid.dh_consumer_public, which is also symbolised by the DH_RP parameter in the AssocReq(T,P,G,DH_RP). The openid.dh_consumer_public variable contains the public part of the Relying Party’s Diffie-Hellman key exchange.

HTTP/1.1 200
Content-Type: text/plain
Content-Length: 188
ns:http://specs.openid.net/auth/2.0
assoc_handle:a6567G!hask
session_type:DH-SHA1
assoc_type:HMAC-SHA1
expires_in:5000
dh_server_public:g^xb mod p
enc_mac_key:h(g^{(xa*xb) mod p})XOR MAC key

Figure 6 Example of an association response message

3.4.2 Association response message

When the OpenID provider receives an association request the provider needs to complete the key exchange and respond to the Relying Party with an association response. The association response message is sent in step 4 in Figure 4 and Figure 6 show an example message of an association response. The association response must be an HTTP response message and the message body needs to be in the Key-Value form [18]. The Key-Value form is constructed by writing the parameter name and then a semicolon and then the value. The separation between the different parameters must be a new line character (\n). The association response message has the form AssocRes(AH,T,DH_OP,H) as shown in section 3.3.

As in the association request message the first three lines in Figure 6 is used for HTTP parameters. The next eight lines contain the seven OpenID protocol parameters. As in the association request the ns parameter, which is the first parameter located at line four in Figure 6, indicates the version of the OpenID protocol.

The second OpenID protocol parameter is the assoc_handle, which contains an ASCII string that is used as a reference for this association between the Relying Party and the OpenID provider. The assoc_handle corresponds to the AH parameter in the AssocRes(AH,T,DH_OP,H) discussed in section 3.3.

The T parameter corresponds to two parameters, session_type and assoc_type, which are set to the same value as in the association request message if the OpenID provider supports the algorithms proposed. The session_type and assoc_type are displayed as parameter third and fourth in Figure 6.

The fifth parameter in Figure 6 is expires_in, which is used to set a lifetime of the association in seconds.

The DH_OP parameter in section 3.3 is the sixth parameter in Figure 6, which is called dh_server_public. The dh_server_public parameter contains the public key of the OpenID provider.

The seventh and last parameter is enc_mac_key in Figure 6 and corresponds to the H parameter in section 3.3. The value of the enc_mac_key is composed by hashing the secret Diffie-Hellman key and then performs an exclusive or function with a secret MAC key generated by the OpenID provider.
When the association has been established the secret key is stored at the Relying Party and the OpenID provider to be used during the authentication phase of the protocol.

HTTP/1.1 302 Found
Location:
http://example.com

Figure 7 Example of an authentication request message

3.4.3 Authentication request message

When the association has been completed the Relying Party begins the authentication process by sending an authentication request message. Step 5 in Figure 4 shows an authentication request and the form of the message is AuthReq(OP, RP, ID). The authentication request message is transmitted as a HTTP redirect message via the user to the OpenID provider. This type of request is called indirect request. Figure 7 shows an example of an authentication request message.

The first line in Figure 7 is the HTTP status parameter and in the example this parameter is set to 302, which indicates that the message is a redirect message. The second line in Figure 7 is also a HTTP protocol parameter called location. The value of the location parameter is the URL to the OpenID provider and then the OpenID protocol parameters are inserted after the question mark. The OpenID provider address is the OP parameter in section 3.3. The authentication request message consists of seven OpenID parameters.

The first two OpenID parameters in Figure 7 are the openid.ns and openid.mode. The openid.ns parameter is as in the previous messages containing the version of the OpenID protocol used to generate the message. The openid.mode parameter has two defined values, checkid_setup or checkid_immediate. The checkid_setup mode should be used if the Relying Party wants the user to interact with the OpenID provider. The checkid_immediate is used to perform the authentication without allowing the user to interact with the OpenID provider and should be used for instance together with AJAX technology.

The third and fourth OpenID parameters in Figure 7 correspond to the ID parameter in the AuthReq(OP, RP, ID) described in section 3.3. The third parameter has the name openid.claimed_id and the value of the parameter is set to the identifier that the user is claiming to own. The fourth parameter has the name openid.identity and the parameter is used if the user uses an own identifier. Then the OpenID provider needs to have a local identifier to identify the user. If there is no need for a local identifier the openid.identity parameter should have the same value as the openid.claimed_id parameter.

If the user has entered an address to an OpenID provider instead of using the identifier the openid.claimed_id parameter and the openid.identity parameter should be set to the value http://specs.openid.net/auth/2.0/identifier_select. This value indicates that the OpenID provider must identify the user and the user’s identifier.

The fifth parameter in Figure 7 is the openid.assoc_handle. This parameter has the value of the association between the OpenID provider and the Relying Party exchanged in the association response message described in section 3.4.2. If no association is established between the Relying Party and the OpenID provider this parameter is omitted.

The openid.return_to parameter is the sixth parameter in Figure 7 and corresponds to the RP parameter in the AuthReq(OP, RP, ID) message described in section 3.3. The openid.return_to
The parameter is set to the Relying Party’s location address. If this parameter is omitted the OpenID provider cannot respond to the authentication request message.

The seventh and last parameter in the message showed in Figure 7 is the openid.realm parameter. The parameter is used to inform the user at the OpenID provider, which domain the authentication is valid for.

When the Relying Party sends the message it stores a copy of the user identifier and the openid.return_to parameter value to be used for the verification step after the Relying Party receives a positive assertion message.

```
HTTP/1.1 302 Found
```

**Figure 8 Example of a positive assertion message**

### 3.4.4 Positive assertion message

The positive assertion message is the OpenID provider’s response to the Relying Party if the authentication succeeded. Step 7 in Figure 4 shows the positive assertion message on the form PosAssert(RP, OP, ID, N, {Sig}). The OpenID provider should send the positive assertion message as an HTTP redirect message via the user [18]. Figure 8 shows an example of the positive assertion message.

As in the authentication request message the first two lines in Figure 8 show the HTTP parameters. The next eight lines consist of the OpenID protocol message. Precisely as in the authentication request the OpenID parameters are appended to the location address after the question mark with the different that the location address is set to the Relying Party’s address. The Relying Party’s address is taken form the openid.return_to parameter in the authentication request. The location address corresponds to the RP parameter in the PosAssert message described in section 3.3. The positive assertion message contains ten OpenID protocol parameters.

The first OpenID parameter in Figure 8 is as in every other message the openid.ns. The second parameter is the openid.mode and the value is set to id_res, which indicates that this message is a positive assertion message.

The openid.op_endpoint parameter is the third parameter in Figure 8. The openid.op_endpoint parameter contains the OpenID provider’s address and corresponds to the OP parameter in the message PosAssert(RP, OP, ID, N, {Sig}) showed in section 3.3. The Relying Party then knows which OpenID provider post the message.

The fourth and fifth parameter in Figure 8 corresponds to the ID parameter described in section 3.3. The fourth parameter is the openid.clamed_id and the fifth is the openid.identity and these parameters should have the same value as in the authentication request. If the Relying Party used the http://specs.openid.net/auth/2.0/identifier_select in the authentication request message the OpenID provider should set the openid.clamed_id and openid.identity to the correct values.

The sixth parameter in Figure 8 has the name openid.return_to and the value of this parameter is copied from the authentication request sent by the Relying Party.

The openid.response_nonce parameter is the seventh parameter in Figure 8. The openid.response_nonce parameter contains a maximum of 255-character string. The string must
be unique to every assertion message generated by this OpenID provider. The current Coordinated Universal Time (UTC) and an optional random value compose the string [18]. The random value and the time stamp must be separated by the character z.

The eighth parameter is the openidassoc_handle. The openidassoc_handle parameter should specify the association handle to the association, which is signing the positive assertion message. If there was an association between the Relying Party and the OpenID provider and the association is still valid the parameter value is a copy form the authentication request message.

The ninth and tenth parameters in Figure 8 are used for the signature of the message. The ninth parameter is the openid_signed that is used to indicate which parameters are signed. The value of the openid_signed parameter contains a comma separated list of the signed parameters. This list must contain openid_op_endpoint, openid_return_to, openid_response_nonce, openid_clamed_id and openid_identity parameters from the positive assertion message.

The last and tenth parameter in Figure 8 is the openid_invalidate_handle. The openid_invalidate_handle parameter is used if the Relying Party entered an invalid association handle in the authentication request.

3.4.5 Verifying the assertion message

The Relying Party needs to verify the positive assertion message to ensure that the assertion is valid. The Relying Party must check that the openid_return_to parameter in the positive assertion message matches with the openid_return_to parameter in the authentication request message sent by the Relying Party. To verify that the discovery information is correct, the Relying Party needs to confirm that the claimed identifier has the same value as the Relying Party discovered for the user. If the Relying Party used the http://specs.openid.net/auth/2.0/identifier_select as the identifier a new discovery process needs to be performed with the new clamed identifier in the positive assertion message to ensure that the OpenID provider has authority to execute the authentication for the identity. The nonce value in the openid_response_nonce must be validated by the Relying Party to ensure that the nonce is unique form the OpenID provider. The positive assertion message has a signature that can be verified by the Relying Party if an association has been performed with the OpenID provider and the association has not expired. The Relying Party uses the key exchanged in the association to recalculate the signature of the message. If the Relying Party does not have an association with the OpenID provider, the Relying Party must ask the OpenID provider to verify the message signature. Step 8 and 9 in Figure 4 show how the verification with the OpenID provider is done.

To ask the OpenID provider to verify the positive assertion message signature the Relying Party sends a direct message to the OpenID provider as discussed in the abstract description of OpenID section.

The direct message is a copy of the positive assertion message and the message is sent directly to the OpenID provider and not through the user. The openid_mode field is set to the value check_authentication to inform the OpenID provider about the message type. The OpenID provider is verifying the signature and sends a response back to the Relying Party. The response to the Relying Party includes three parameters, openid_ns, openid_is_valid and the openid_invalidate_handle. The openid_is_valid parameter is set to either true or false depending on the signature verification. The openid_invalidate_handle is optional and contains the openid_assoc_handle value sent in the positive assertion message. If the openid_is_valid is set to true and the openid_invalidate parameter is included in the response message, the Relying Party should not use the association in another authentication request.
3.5 Security concerns

According to the OpenID protocol specification [18] there are some security concerns that must be considered. The different OpenID messages are not protected by any encryption by default and an eavesdropper is able to read them in clear text. The assertion response message is the only message that is integrity protected. The authentication method is not specified in the OpenID specification and the Relying Party does not know how the user is authenticated. The OpenID foundation is working on a specification to enable the Relying Party to request a specific authentication method [24].

3.5.1 Eavesdropping attack

No encryption is used in the default specification of the OpenID protocol. An eavesdropper could track the authentications a user is performing and in that way know which services the user is using. The user could also change every message that is not integrity protected without the notice of the Relying Party, the user and the OpenID provider.

3.5.2 DNS attacks on OpenID

The OpenID protocol uses the Domain Name System (DNS) protocol for the discovery process and the redirection of the user. If someone could alter the information in the DNS system the OpenID provider could be impersonated [18]. This can be achieved by altering the DNS cache at the victim to change the URL to IP address mapping [15].

3.5.3 Other types of attacks

If the Relying Party is malicious it can be used in so called phishing attacks. The Relying Party could redirect the user to the wrong OpenID provider that tries to steal the user’s authentication details. The discovery process could also be used to falsify an authentication. The information is not integrity protected and could be changed to alter the OpenID provider that should do the authentication.

3.5.4 Counter methods against the attacks

The OpenID specification recommends that the Secure Socket Layer (SSL) protocol should be used. The SSL protocol encrypts the communication between the user, Relying Party and the OpenID provider. This hides the information in the messages for an eavesdropper and makes it more difficult to perform a man in the middle attack against the OpenID protocol.
4 Security analysis of the OpenID protocol

The OpenID protocol is formally analysed by the AVISPA tool. The AVISPA tool is a state of the art analysis tool for security protocols and the AVISPA tool is selected because the tool is built to analyse protocols like the OpenID protocol. When analysing the OpenID protocol with the AVISPA tool, the OpenID protocol is checked for security errors. One attack found by the AVISPA tool is implemented to show that the attack is working in a realistic environment.

4.1 Experiment setup

The tool used to analyse the OpenID protocol is called Secure Protocol ANimator for AVISPA (SPAN), which is a graphical user interface for the AVISPA tool. The version of SPAN used in this experiment is 1.5. The SPAN tool is installed on a Macintosh with the operating system Mac OS X version 10.5.6.

4.2 Formal analysis of OpenID protocol

To analyse the OpenID protocol with the AVISPA tool, the protocol must be modelled in the HLPSL. To create the model the OpenID protocol is split up into multiple basic roles to describe each part in the protocol. One role is symbolising the user, one is symbolising the Relying Party, one is the OpenID provider and one is symbolising the discovery agent. The model of the OpenID protocol has excluded the association between the Relying Party and the OpenID provider.

4.2.1 The OpenID model

The OpenID model written in the HLPSL is explained in this section. The lines beginning with the percent sign is a comment line. The user model in HLPSL is shown below:

```
%The user role of the OpenID protocol
role user(U, R, O, L : agent,
    SND_RU, SND_OU, RCV_RU, RCV_OU : channel(dy),
    K : symmetric_key, %Key used in the authentication
    Hash : hash_func)
played_by U
def=
local State : nat,
    Na : text, % Nonce used by the OpenID protocol
    X : message, % Message containing
        % variables unknown to the user role
    Nb : text, % Nonce for the authentication part
    Nc : text, % Nonce for the authentication part
    Ks : message % New key for the authentication part
init State := 0
transition
1. State = 0 /\ RCV_RU(start) =|>
    % Contact the relying party
    State' := 2 /\ SND_RU(U.R.L)
2. State = 2 /\ RCV_RU(U.O.R.L) /\ RCV_OU(start) =|>
    % Receives the redirect to the OpenID provider
    % the OpenID provider with authentication information included
    State' := 4 /\ Nb' := new() /\ SND_OU(O.U.R.L.(Nb')_K)
```
% Transition 3 is used for the authentication of the user to
% the OpenID provider

3. State = 4 \ RCV_OU({Nc'}_K) =|>
   State' := 6 \ Ks' := Hash(Nb.Nc') \ SND_OU({Nc'}_Ks') \ /
   witness(U,O,op_user_authentication, Nc')
   % Send the redirect information from the OpenID provider to
   % the Relying Party

4. State = 6 \ RCV_OU(U.O.R.L.Na'.X') =|>
   State' := 8 \ SND_RU(U.O.R.L.Na'.X')
end role

As shown on line 18 the transition one begins by the start of the RCV_RU channel and then the user start the authentication procedure by sending a message to the Relying Party. This message is step 1 in Figure 4. The second transition showed from line 22 to line 25 is modelling the authentication message redirected by the user agent. This is step 5 in Figure 4. The \{Nb'}_K variable on line 25 is used for the authentication between the user and the OpenID provider, which is step 6 in Figure 4. The authentication between the user and the OpenID provider is modelled by an authentication method that is using a shared key between the user and the OpenID provider. This step is not included in the OpenID specification. This authentication protocol is taken from the HLPSL tutorial and a message description is shown below [31].

\[ A \rightarrow B : \{Na\}_K \]
\[ B \rightarrow A : \{Nb\}_K \]
\[ A \rightarrow B : \{Nb\}_K1, \text{ where } K1=Hash(Na.Nb) \]

This authentication protocol is found to be secure by the AVISPA tool and therefore this protocol is used in the OpenID protocol model to ensure that the authentication of the user does not affect the outcome of the analysis. A is assigned to the user role in the OpenID model and line 28 to line 30 show the model code for the authentication. B is the OpenID provider.

The last and fourth transition of the user role is the redirection of the positive assertion message from the OpenID provider to the Relying Party, which is step 7 in Figure 4.

The Relying Party is modelled in HLPSL as shown below.

% The Relying Party role of the OpenID protocol
role relying_party(R : agent,
    SND_UR, SND_OR, SND_L, RCV UR, RCV OR, RCV_L :
    channel(dy))
played_by R
def =
local State : nat,
    U,O,L : agent,
    Na : text, % The nonce variable in the OpenID protocol.
    X : message % Signature of the OpenID protocol
init State := 1
transition
% Relying Party receives the authentication request from the user
1. State = 1 \ RCV_UR(U'.R.L') =|>
   % Relying Party ask for the users OpenID provider to the discovery
   % process
   State' := 3 \ SND_L(R.L')
   % Relying Party receives the OpenID provider

2. State = 3 \ RCV_L(R.L.O') =|>
   % Send the authentication request to the user for redirection to
   % the OpenID provider
   State' := 5 \ SND_UR(U.O'.R.L)
   % Receives the positive assertion message from the OpenID provider
   % redirected from the user

3. State = 5 \ RCV_UR(U.O.R.L.Na'.X') =|>
   % Sends the positive assertion message back to the OpenID provider
   % for verification
   State' := 7 \ SND_OR(O.R.L.Na'.X')
Security analysis of the OpenID protocol

The first transition is started when the Relying Party receives the request from the user, which is step 1 in Figure 4. On line 17 the Relying Party model sends a message to the discovery agent and this is symbolised with step 2 in Figure 4. The discovery agent is modelled in the HLPSL OpenID model to distinguish the request the Relying Party does to discover the OpenID provider. The model of the discovery agent is shown in Appendix A on line 111. The second transition begins on line 19 with the receiving of the message from the discovery agent. An authentication request is then sent to the user for redirection to the OpenID provider, which is step 5 in Figure 4. In the third and last transition the Relying Party receives the positive assertion message showed in step 7 in. Then the Relying Party role sends a verification message to the OpenID provider as step 8 in Figure 4 shows. Then a request is issued to make the authentication based on the message signature which is symbolised.

The last role is the OpenID provider. This role is displayed below.

```plaintext
role openid_provider(O, U, L: agent,
                    SND_UO, SND_PO, RCV_UO, RCV_PO : channel(dy),
                    K : symmetric_key,
                    Hash : hash_func)
played_by O

def =
local State : nat,
R : agent,
H : hash_func,
Ko : symmetric_key,
Na : text,
X : message,
Nb : text,
Nc : text,
Ks : message
init State := 11

transition
% Receives the authentication request from the User with the
% authentication information included
1. State = 11 \ RCV_UO(O.U.R'.L. {Nb'}_K) =|>
   State' := 13 \ Nc' := new() \ SND_UO({Nc'}_K) \ Ks' :=
   Hash(Nb'.Nc') \ secret(Ks',user_session_key,(U,O))
% Receives the authentication information from the User
2. State = 13 \ RCV_UO((Nc)_Ks) =|>
   % The request is set to the authentication process between the user
   % and the OpenID provider
   State' := 15 \ Na' := new() \ Ko' := new()
   \ SND_UO(U.O.R.L.Na'.H(O.R.L.Na'.Ko'))
   \ request(O,U,op_user_authentication,Nc)
% Receives the verification message from the Relying Party
   % Witness the signature generated by the OpenID provider
   State' := 17 \ witness(U,O,o_r_signature, H(O.R.L.Na.Ko))

end role
```

The first transition on line 21 begins when the OpenID provider receives the authentication request from the user, which is step 5 in Figure 4. Then the OpenID provider role sends a message back to the user as shown in step 6 in Figure 4 containing authentication information for the authentication method. The OpenID provider receives the last authentication message from the first message in the second transition on line 26. The positive assertion message is then constructed and sent to the user for redirection to the Relying Party, which is step 7 in Figure 4. The OpenID provider also issues a request for the authentication method between the user and
Security analysis of the OpenID protocol

the OpenID provider shown on line 32. At line 34 the OpenID provider receives the verification message from the Relying Party and the OpenID provider issues a witness statement on line 36 for the OpenID protocol, which is step 8 and 9 in Figure 4.

The complete code of the OpenID protocol is shown in the Appendix A. The protocol is then analysed by the AVISPA tool.

4.2.2 Result of the AVISPA analysis

The implemented model has been analysed by OFMC backend and the OpenID protocol is found to have a possible attack.

<table>
<thead>
<tr>
<th>Intruder</th>
<th>Relying Party</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The Intruder access the Relying Party</td>
<td></td>
</tr>
<tr>
<td>2. The Relying Party send a discovery message to the Intruder</td>
<td></td>
</tr>
<tr>
<td>3. The Intruder respond the discovery message</td>
<td></td>
</tr>
<tr>
<td>4. The Relying Party sends the authentication request message</td>
<td></td>
</tr>
<tr>
<td>5. The Intruder sends the positive assertion response message</td>
<td></td>
</tr>
<tr>
<td>6. The Relying Party sends the verification message</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 9 The message chart for the discovered attack*

The Figure 9 shows the generated attack trace for the OpenID protocol. In step 1 the intruder establishes contact to the Relying Party. In step 2 the Relying Party does the discovery by sending a message back to the intruder and the Intruder answers in step 3 and tells the Relying Party to contact the intruder for the user authentication. To enable this in a realistic way the DNS infrastructure or the routing infrastructure must be attacked. In step 4 the Relying Party sends the authentication request message to the intruder and in step 5 the intruder sends a positive assertion message back to the Relying Party. The Relying Party must send a verification request to the intruder because no association has occurred between the Relying Party and the intruder. The intruder is then authenticated to the Relying Party without the users’ knowledge. The attack is based on the fact that the intruder could play all the roles except the
Relying Party. As mentioned earlier, this means that the DNS or routing infrastructure must be attacked to direct all the messages to the intruder or the OpenID provider must be compromised.

### 4.3 Implemented attack

An attack similar to the one found by the AVISPA tool is implemented to confirm that the OpenID protocol can be attacked in a realistic environment. The setup is shown below in Figure 10 and is based on a rough User agent and a man in the middle attack against the OpenID provider.

The User agent is a simple HTTP client to send messages to a specific Relying Party. In this case, the User agent is constructed to send messages to the DotNetOpenID’s Relying Party. The attack begins by the special User agent that sends a message to the Relying Party. This is shown as step 1 in Figure 10. Included in the message is the user’s URL posted to the Relying Party. The Relying Party is performing a discovery of the OpenID provider and establishes an association with the OpenID provider shown as step 2 in Figure 10.

The Relying Party then sends an authentication request message to the special User agent for redirection to the OpenID provider shown in step 3 in Figure 10. The special User agent does not redirect to the OpenID provider.

Instead, the special User agent takes the authentication request message and builds a positive assertion response message. The special User agent edits the positive assertion response message and sets the assoc_handle parameter to a new value to force the Relying Party to send out an assertion message validation. The special User agent sends the forged positive assertion message back to the Relying Party as shown in step 3 in Figure 10.

When the Relying Party receives the positive assertion message, the validation process takes place. Because of the change of the association handle, the Relying Party needs to send an assertion message validation to the OpenID provider. The assertion message validation is then sent as a direct message to the OpenID provider shown in step 4 in Figure 10. The OpenID provider is checking the positive assertion response message included in the assertion message validation. The OpenID provider has not generated the positive assertion response message and...

![Figure 10: An overview of the implemented attack](image-url)
is returning a negative validation response. The negative validation response is passed through an eavesdropper as shown in step 5 in Figure 10.

The eavesdropper is an application called Ettercap and is available at http://ettercap.sourceforge.net/. This program performs a so-called address resolution protocol spoofing attack against the OpenID provider. The address resolution protocol is a protocol that glues the network level to the data link level in the OSI network layer stack. The address resolution protocol matches mac-addresses with corresponding IP-addresses. The data link level uses the mac-addresses and the network layer uses the IP-addresses. The Ettercap program change the mapping of the victim’s IP-addresses to make the traffic pass through the eavesdropper. The eavesdropper is performing a man in the middle attack by changing the negative validation response into a positive validation response and sends the forged message to the Relying Party shown in step 6 in Figure 10. A filter in the Ettercap tool does the changing of the message by analysing the messages for negative validation response. The User agent is then signed in to the Relying Party without authentication to the OpenID provider. Depending on where the man in the middle is located there are two different scenarios. If the man in the middle is located at the OpenID provider as in the implemented attack, every user associated with that OpenID provider could be authenticated to all Relying Parties. If the man in the middle attack is performed at the Relying Party, the attacker could authenticate a user from all the existing OpenID providers.
5 DotNetOpenID, an OpenID implementation

The DotNetOpenID implementation is written in the C# language. The implementation uses ASP.NET as the server language for the web pages. DotNetOpenID is an open source implementation of the OpenID specifications. The DotNetOpenID implementation is split into different components that match the OpenID specification parts. The main modules are the Relying Party and the OpenID provider.

5.1 Relying Party

The Relying Party consists of multiple classes to handle the functionality. Some of the classes are shown in the Figure 11. The OpenIdLogin class starts the authentication of the user. The OpenIdLogin class act as a web control placed on the page where the users should enter their identities. When the web page with the control is requested, the OpenIdLogin control checks if the request contains an OpenID message or if the user has entered an identifier. If the user entered an identifier the OpenIdLogin control starts the authentication process. During the authentication the discovery process is completed and the creation of the authentication request message is done. The association with the OpenID provider is also done during the authentication phase. The requests and responses are modelled as their own classes as shown in the Figure 11. The positive assertion message is also received and processed by the OpenIdLogin control.

![Class Diagram](image)

Figure 11 A class diagram of the most important classes in the Relying Party module

5.2 OpenID provider

The OpenID provider in this implementation consists of a server endpoint, which is used to handle all the requests from the Relying Party and sends the responses to the requests. The most important classes that make up the OpenID provider are shown in the Figure 12. The OpenID provider is also responsible for creating the XRDS documents for the users. The implementation has an XRDS controller to create the XRDS documents. The authentication method in this implementation is username/password based. As in the Relying Party there are classes to model
the different types of responses and requests. When the endpoint receives an authentication request, the endpoint parses the request and presents a login page if the user is registered at the provider.

![Class diagram of the most important classes in the OpenID provider module](image)

**Figure 12** A class diagram of the most important classes in the OpenID provider module

### 5.3 Discovery process

The Relying Party module uses the discovery process part to parse the discovery information. There are classes that represent the different parts of the XRDS document to model the information. There is also a class to parse the HTML document if the discovery process finds such documents.

### 5.4 Diffie-Hellman

To perform the key exchange between the Relying Party and the OpenID provider there is a Diffie-Hellman module. The Diffie-Hellman module contains classes to generate the keying material to perform a key exchange. Both the Relying Party and the OpenID provider modules need to use the Diffie-Hellman classes. There are also classes to handle the big integers that the Diffie-Hellman needs to represent the big numbers. To generate the prime numbers there are classes to do the primality tests. The developers of the DotNetOpenID project are not the developers of the Diffie-Hellman module.

### 5.5 Extensions

This part of the implementation has classes to handle the extensions of the OpenID protocol. There is one extension, the registry information request, which is used to send user registry information from the OpenID provider back to the Relying Party. The registry information request is useful when the Relying Party want information about the user and the user does not need to re-enter the same information twice. There is also an attribute exchange extension to the OpenID protocol. The attribute exchange extension makes it possible for the Relying Party and
the OpenID provider to exchange information. The last extension that is implemented is the provider authentication policy extension. This extension makes it possible for the Relying Party to request a certain authentication method at the OpenID provider or the OpenID provider can inform the Relying Party about the authentication method.
6 Security analysis of the OpenID implementation

The DotNetOpenID implementation of the OpenID protocol is selected for evaluation by two independent static analysis tools. The method of using static analysis tools is used because the implementation already has implemented unit tests. It is also interesting to analyse an OpenID implementation with the static analysis tools to see if various problems occur. The static analysis tools that are used are FxCop and Gendarme. The most important goal of the static code analysis is to determine if the implementation of the OpenID protocol contains any security flaws.

6.1 Experiment setup

The experiments are performed on a Windows machine running Windows XP with service pack 3. The FxCop static analysis tool used in this experiment has version number 1.36. This version is the latest version of the program that is released from Microsoft. The second analysis tool used is the Gendarme version 2.2.0.0. The both programs are chosen because they can analyse the Common Intermediate Language code generated from the C# source code of the implementation. Both analysis tools have multiple rules that analyse the source code to find errors.

6.1.1 DotNetOpenID

The implementation analysed in this experiment is DotNetOpenID. The reason for choosing the DotNetOpenID implementation is that the implementation implements both the Relying Party and the OpenID provider. The version of the DotNetOpenID implementation is 2.4.3. The .Net framework’s version is 3.5 but the implementation is compiled with the 2.0 specifications because the DotNetOpenID implementation is built to be compatible with that version.

6.1.2 The Experiment execution

The analysis tools can only analysed CIL code and therefore the implementation needs to be compiled. By using the compiler provided in the .Net framework software development kit the DotNetOpenID source code can be compiled to the CIL. The compiled code is stored in a file, which is called an assembly. The assembly file is loaded into each analysis tool and the tool run the different tests. The tests are split into different categories and each of these categories is enabled separately. Each category has a set of rules, which is executed when that category is selected. When the test of one category is finished the result of the test is displayed.

6.2 Result

The result from the analysis of the DotNetOpenID implementation is shown in Table 3.
Security analysis of the OpenID implementation

<table>
<thead>
<tr>
<th>Categories in FxCop</th>
<th>Number of errors found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design category</td>
<td>49</td>
</tr>
<tr>
<td>Globalization category</td>
<td>3</td>
</tr>
<tr>
<td>Naming category</td>
<td>29</td>
</tr>
<tr>
<td>Mobility category</td>
<td>0</td>
</tr>
<tr>
<td>Performance category</td>
<td>20</td>
</tr>
<tr>
<td>Interoperability category</td>
<td>0</td>
</tr>
<tr>
<td>Security category</td>
<td>0</td>
</tr>
<tr>
<td>User category</td>
<td>38</td>
</tr>
<tr>
<td>Total number of errors</td>
<td>147</td>
</tr>
</tbody>
</table>

*Table 3 Result from the FxCop tool analysis of DotNetOpenID*

In Table 4 the analysis result from the Gendarme tool is shown.

<table>
<thead>
<tr>
<th>Categories in Gendarme</th>
<th>Number of errors found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad practice</td>
<td>65</td>
</tr>
<tr>
<td>Concurrency</td>
<td>13</td>
</tr>
<tr>
<td>Correctness</td>
<td>31</td>
</tr>
<tr>
<td>Design</td>
<td>54</td>
</tr>
<tr>
<td>Design generic</td>
<td>9</td>
</tr>
<tr>
<td>Design linq</td>
<td>0</td>
</tr>
<tr>
<td>Exception</td>
<td>18</td>
</tr>
<tr>
<td>Interoperability</td>
<td>0</td>
</tr>
<tr>
<td>Maintainability</td>
<td>36</td>
</tr>
<tr>
<td>Naming</td>
<td>126</td>
</tr>
<tr>
<td>Performance</td>
<td>200</td>
</tr>
<tr>
<td>Portability</td>
<td>24</td>
</tr>
<tr>
<td>Security</td>
<td>0</td>
</tr>
<tr>
<td>Security CAS</td>
<td>0</td>
</tr>
<tr>
<td>Smell</td>
<td>77</td>
</tr>
<tr>
<td>Serialization</td>
<td>16</td>
</tr>
<tr>
<td>Ui</td>
<td>0</td>
</tr>
<tr>
<td>Total number of errors</td>
<td>669</td>
</tr>
</tbody>
</table>

*Table 4 Result from the Gendarme tool analysis of DotNetOpenID*

The numbers in both the Gendarme and FxCop table shows all occurrences of the errors. This includes multiple errors found by the same rule but in different locations in the source code.

The most important result was that FxCop and Gendarme did not find any errors in the source code when running the security category rules. The Security category has rules to detect the most important security threats in the source code. However other categories found errors in the implementation and the most critical is described in this section.

### 6.2.1 Examples of errors detected by the Gendarme and FxCop tool

Both the FxCop tool and the Gendarme tool react on the method RealmUrl in the source code of the class OpenIdTextBox. The rule that triggers is checking if results from methods are ignored.
At line 13 in the code there is a declaration of a new object of the type Realm. The new instance of the Realm is not stored in a variable in the method and the result from the constructor method is therefore never used. The developer is checking if the value provided to the method is a correct realm URL. At line 24 there is also a similar situation but in this case an Uri object is created. This means that new objects is created and not used. The garbage collector needs to delete these objects, which can take some time.

The FxCop and Gendarme find an error in the BigInteger class located in the Diffie-Hellman package. The analysis shows that there is one parameter, which is not used in one of the constructors in the BigInteger class.
Security analysis of the OpenID implementation

```csharp
    { this.data = new uint [len];
    this.length = len;
  }

The sign variable at line 1 is never used in the constructor.

The FxCop analysis tool reports an error in the OpenIdLogin class. The rule called “operation should not overflow” is triggered by the following code.

```csharp
public override short TabIndex {
    get { return base.TabIndex; } } } 
    set {
        WrappedTextBox.TabIndex = (short)(value + textBoxTabIndexOffset); 
        loginButton.TabIndex = (short)(value + loginButtonTabIndexOffset); 
        rememberMeCheckBox.TabIndex = (short)(value + rememberMeTabIndexOffset); 
        registerLink.TabIndex = (short)(value + registerTabIndexOffset); 
    }
}
```

The rule triggers on lines 4, 6, 8 and 10 because on these lines different TabIndex variables’ values are increased without testing the new values. The variable is declared as short and if the value is larger than 32767 or smaller than -32768 there is an argument out of range exception thrown. The Gendarme analysis tool does not detect this overflow error.

According to FxCop the string comparison should be done with upper case if modification is needed. The UriIdentifier class is using the ToLowerInvariant method.

```csharp
static bool TryCanonicalize(UriBuilder uriBuilder, out Uri canonicalUri) {
    uriBuilder.Host = uriBuilder.Host.ToLowerInvariant(); 
    canonicalUri = uriBuilder.Uri;
    return true;
}
```

However this example the method is legal because the URI containing the URL should be in lower case format. Therefore this error is not correct. The same thing happens in the IdentityEndpoint class where the method bestGuessNormalisation also calls the ToLowerInvariant method.

The Gendarme analysis tool also detects errors that are not discovered by the FxCop analysis tool. The Gendarme AvoidUncalledPrivateCode rule detects private source code that is never called. The uncalled code is declared as private or internal and the Gendarme is discovering multiple places in the DotNetOpenID implementation that has uncalled private code. Another rule that detects errors by the Gendarme analysis tool is named AvoidUnneededCallsOnStringRule. One example of this error is shown in the code below:

```csharp
if (Email != null) {
    fields.Add(Constants.email, Email.ToString());
}
```

This code is a part of the ClaimsResponse class and the Email parameter is a string and the call to the ToString method is unnecessary because this call is only returning the same string.

According to Table 3 and Table 4 multiple other errors are discovered by the analysis tools but the errors explained are the most interesting ones.
7 Conclusion

The OpenID protocol has been analysed with the AVISPA tool, which detected security flaws in the protocol. The attack found by the AVISPA tool is possible due to the fact that the messages are not integrity protected and therefore anyone could change the information in the messages. The information cannot be trusted in these messages. Also the information in the messages is not confidentially protected by default and an eavesdropper can read every data in clear text. An attacker would be able to collect data about what service providers a user uses. A version of this attack is implemented in a real environment, which shows that an attack against the OpenID protocol can be performed. To prevent the implemented attack the messages should at least be integrity protected to avoid modification of the protocol’s messages. To integrity protect and encrypt the authentication messages the Relying Party and the OpenID provider can use the cryptographic SSL/TLS protocols.

Two code analysis tools have been applied on the DotNetOpenID implementation of the OpenID protocol. Neither Gendarme nor FxCop did report any errors when applying the rules in the security category. To conclude that the implementation does not have any security errors, more analyses are needed. During the course of the project it became clear that the DotNetOpenID development team is using the FxCop on the source code. The FxCop still detects errors in the source code but the errors are not critical in a security perspective. It is interesting that the DotNetOpenID development team uses FxCop and does not correct the errors detected.

During the analyses of the DotNetOpenID some rules in the FxCop analysis tool reports false negatives in the source code. Many of these rules are located in the analysis tool’s naming category. The reason for many of the false negatives is that the tool uses dictionaries to check the spelling of the names in the code and the names used by the DotNetOpenID implementation contains abbreviations. The FxCop rule NormalizeStringsToUppercase reports false negatives in the source code. The DotNetOpenID uses the ToLowerInvariant method correctly because the URL should be lowercased. This suggests a refinement of the FxCop rule to eliminate this false negative. There are errors found regarding performance issues that could be critical due to heavy load on the web application using this implementation. The performance errors should be taken into considerations and should be corrected. There are multiple similarities and differences between the both analysis tools used in this project. Gendarme and FxCop have similar rules that detect the same types of errors but they do also differ at certain categories. The rules that are similar in both tools are regarding the .Net framework design and programming rules. Gendarme has more categories and it also has rules that are bound to the Mono framework. For instance the portability category contains rules that check the implementation for compatibility with the Mono framework. The FxCop does not contain these types of rules.

7.1 Future Work

One potential area of further work is to analyse how to protect the OpenID protocol from the attack created in this report and evaluate how different types of cryptographic affect the OpenID protocol.

One interesting task to perform is to evaluate other aspects of the implementation of OpenID for example evaluates the implementation’s performance or to compare multiple implementations. Another approach could be to investigate how the authorisation of the users could be federated. Today the access control is done at the Relying Party and the OpenID protocol does not specify how the access control should be made. The Relying Party may want to delegate the access control back to the OpenID provider or to another entity. How to enable that with OpenID could be an interesting task to perform.
Bibliography


Appendix A OpenID HLPSL model

%The user role of the OpenID protocol
role user(U, R, O, L : agent,
    SND_RU, SND_OU, RCV_RU, RCV_OU : channel(dy),
    K : symmetric_key, %Key used in the authentication
    Hash : hash_func)
played by U
def= _
local State : nat,
    Na : text, % Nonce used by the OpenID protocol
    X : message, % Message containing variables unknown to the user
    Nb : text, % Nonce for the authentication part
    Nc : text, % Nonce for the authentication part
    Ks : message % New key for the authentication part
init State := 0
transition
1. State = 0 \ RCV_RU(start) =>
    % Contact the relying party
    State' := 2 \ SND_RU(U.R.L)
    % Receives the redirect to the OpenID provider
2. State = 2 \ RCV_RU(U.O.R.L) \ RCV_OU(start) =>
    % Send the redirect message to the OpenID provider with
    % information included
    State' := 4 \ Nb' := new() \ SND_OU(O.U.R.L.{Nb'}_K)
    % Transition 3 is used for the authentication of the user to the
    OpenID provider
3. State = 4 \ RCV_OU({Nc'}_K) =>
    State' := 6 \ Ks' := Hash(Nb.Nc') \ SND_OU({Nc'}_Ks') \ /
    witness(U,O,op_user_authentication, Nc')
    % Send the redirect information from the OpenID provider to the
    Relying Party
4. State = 6 \ RCV_OU(U.O.R.L.Na'.X') =>
    State' := 8 \ SND_RU(U.O.R.L.Na'.X')
end role

%The Relying Party role of the OpenID protocol
role relying_party(R : agent,
    SND_UR, SND_OR, SND_L, RCV_RU, RCV_OR,
    RCV_L : channel(dy))
played by R
def= _
local State : nat,
    U,O,L : agent,
    Na : text, % The nonce variable in the OpenID protocol
    X : message %Signature of the OpenID protocol, unknown to the
    Relying Party
init State := 1
transition
1. State = 1 \ RCV_UR(U'.R.L') =>
% Relying Party ask for the users OpenID provider to the discovery process
State' := 3 /\ SND_L(R.L')
% Relying Party receives the OpenID provider
2. State = 3 /\ RCV_L(R.L.O') =|>
% Send the authentication request to the user for redirection to the OpenID provider
State' := 5 /\ SND UR(U.O'.R.L)
% Receives the positive assertion message from the OpenID provider redirected from the user
3. State = 5 /\ RCV UR(U.O.R.L.Na'.X') =|>
% Sends the positive assertion message back to the OpenID provider for verification
State' := 7 /\ SND OR(O.R.L.Na'.X')\ /
 request(U,R,o_r_signature,X')
end role

% The OpenID provider role
role openid_provider(O, U, L: agent,
SND_UO, SND_PO, RCV_UO, RCV_PO : channel(dy),
K : symmetric_key,
Hash : hash_func)
played_by O
def=
local State : nat,
R : agent,
H : hash_func,
Ko : symmetric_key,
Na : text,
X : message,
Nb : text,
Nc : text,
Ks : message
init State := 11
transition
1. State = 11 /\ RCV_UO(O.U.R'.L. {Nb'}_K) =|>
% Send the authentication information back to the User
State' := 13 /\ Nc' := new() /\ SND_UO({Nc'}_K) /\ Ks' :=
 Hash(Nb'.Nc') /\ secret(Ks',user_session_key,{U,O})
% Receives the authentication information from the User
2. State = 13 /\ RCV_UO({Nc}_Ks) =|>
% Send the positive assertion response back to the user.
% The request is set to the authentication process between the user and the OpenID provider
State' := 15 /\ Na' := new() /\ Ko' := new() /\
 request(O,U,op_user_authentication,Nc)
% Receives the verification message from the Relying Party
% Witness the signature generated by the OpenID provider
end role

% This role symbolizes the discovery of the OpenID provider.
role url (L, O : agent,
SND_PL, RCV_PL : channel(dy))
played_by L
def=
local State : nat,
  R : agent
init State := 19

transition
  % Receive the request from the Relying Party
1. State = 19 \ RCV_PL(R'.L) =>
  % Respond to the Relying Party
  State' := 21 \ SND_PL(R'.L.O)
end role

%The session role binds the different parts of the OpenID protocol
together
to

role session(O, R, U, L : agent,
  K : symmetric_key,
  H : hash_func)
def=
local SND_RU, SND_OU, RCV_RU, RCV_OU,
  SND_UR, SND_OR, SND_L, RCV UR,
  RCV OR, RCV L, SND UO, SND PO,
  RCV UO, RCV PO, SND PL, RCV PL : channel(dy)

composition
  %The different roles binds together.
user(U, R, O, L, SND_RU, SND_OU, RCV_RU, RCV_OU, K, H)
/\openid_provider(O, U, L, SND UO, SND PO, RCV UO, RCV PO, K, H)
/\relying_party(R, SND UR, SND OR, SND L, RCV UR, RCV OR, RCV L)
/\url(L, O, SND PL, RCV PL)
end role

role environment()
def=
const o,r,u,l : agent,
o_r_signautre, user_session_key, op_user_authentication :
  protocol_id,
  kuo, kio, kui : symmetric_key,
  h : hash_func
intruder_knowledge = {o,r,u,l, kio, kui, h,i}
composition
  session(o,r,u,l, kuo, h)
/\ session(o,r,u,i, kuo, h)
/\ session(o,r,i,l, kio, h)
/\ session(o,i,u,l, kuo, h)
/\ session(i,r,u,l, kui, h)
end role

goal
% The secrecy goal of the authentication procedure
secrecy_of user_session_key
% The authentication goal of the authentication procedure
authentication_on op_user_authentication
% The authentication goal of the OpenID protocol
authentication_on o_r_signautre
end goal

environment(
Appendix B Abbreviation

AJAX – Asynchronous JavaScript and XML is and technology to update web pages dynamically with the help of JavaScript and XML.

HTML – Hyper Text Markup Language is the language that is used for construct web pages.

HTTP – Hyper Text Transfer Protocol is the protocol to transfer information mostly web pages.

MAC – Message Authentication Codes is used to generate signatures.

XML – Extensive Markup Language is a file format to store data in a hierarchy.