



Norwegian University of  
Science and Technology

# Security in Single Sign-On Web Applications

An Assessment of the Security in and Between Web Applications  
Sharing a Common Single Sign-On User Session

**Jo Grimstad**

Master of Science in Communication Technology

Submission date: July 2010

Supervisor: Svein Johan Knapskog, ITEM

Co-supervisor: Øystein Sekse Øie, Kantega AS  
Nils Tesdal, Signicat AS

Norwegian University of Science and Technology  
Department of Telematics



# Problem Description

Single Sign-On (SSO) is a mechanism which gives users the control over their usernames across multiple Web sites. This way, they do not have to create a new account with a username/password combination at every Web site they wish to use. There are many benefits of only having to sign in using one digital identity on the Internet, like simple registration, less passwords to remember, reduced information maintenance (e.g. postal and e-mail address), and reduced time spent on the actual login to Web applications. Today, more and more Web sites support SSO. However, using a SSO protocol also has its drawbacks, like phishing weaknesses and being a single point of failure. Also, the fact that a user will be logged in at many places at once is not good, at least not from a Web security point of view.

The work will include investigation of security vulnerabilities and threats related to the usage of SSO in Web applications. As part of the thesis, the student will implement two Web applications using OpenID, which is a decentralized authentication protocol. These applications will act as OpenID relying parties, i.e. providing user registration and login via OpenID providers (e.g. Google or myOpenID). Experimenting will be performed on both sample applications individually, but the main focus will be on how information security is safeguarded between two Web applications that belong to the same SSO user session. In this case, the student will examine the feasibility of attacks that are of relevance, like Cross-Site Request Forgery (CSRF). Security assessments will be performed for the various phases of a user's SSO session; mainly where the user is logged in, but also for the login and logout phase (single logout).

Assignment given: 15. February 2010

Supervisor: Svein Johan Knapkog, ITEM



## Abstract

*Single Sign-On* (SSO) is a solution where the authentication process is taken care of once by a third-party Web site rather than at each of the the Web sites providing services to their users. This new way of separating user identities from the service-providing Web sites leads to different security requirements. As an approach towards assessing the security of Web applications utilizing SSO, this thesis investigates the concepts and functionality of *OpenID*, a decentralized authentication protocol. The assessment addresses vulnerabilities and threats related to SSO, using real Web applications as examples. Development of an OpenID-enabled Web application is a part of the security assessment.

The thesis includes experimenting with various OpenID-enabled Web sites and *Identity Providers* (IdPs), and observing how they are affected by different kinds of Web security threats. The results of the thesis shows how security weaknesses were discovered at two major IdPs by performing Clickjacking attacks. Also, the thesis outlines some attacks that are threatening the concept of SSO in general.



## Preface

This thesis was written as part of the 5 year MSc in Communication Technology (Department of Telematics) with a specialization in Information Security at the Norwegian University of Science and Technology (NTNU) in the spring semester of 2010. The Master's thesis was written with the support from Kantega AS, a consulting, technology services and application outsourcing company located in Trondheim, providing me with co-supervising of the work.

I would like to thank my supervisor at NTNU, professor Svein Johan Knapskog, for valuable feedback and guidance during the work. Furthermore, I wish to thank my co-supervisors for valuable input throughout my work; Øystein Sekse Øie at Kantega AS, and Nils Tesdal at Signicat AS.

Trondheim, July 12, 2010

Jo Grimstad

jogrimst@stud.ntnu.no





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Related Work . . . . .	2
1.3	Refinements and Limitations . . . . .	2
1.4	Thesis Outline and Methodology . . . . .	3
<b>2</b>	<b>Theoretical Background</b>	<b>7</b>
2.1	Goal . . . . .	7
2.2	Single Sign-On (SSO) . . . . .	7
2.2.1	Components . . . . .	8
2.2.2	Possible Applications . . . . .	9
2.3	Single Logout (SLO) . . . . .	10
2.4	Identity Providers (IdPs) . . . . .	12
2.4.1	Multi-Factor Authentication . . . . .	15
2.5	Web Security Threats . . . . .	16
2.5.1	Cross-Site Request Forgery (CSRF) . . . . .	17
2.5.2	Clickjacking . . . . .	19
2.5.3	Man-in-the-Middle (MITM) . . . . .	19
2.5.4	Phishing . . . . .	21
2.6	Summary . . . . .	23
<b>3</b>	<b>OpenID</b>	<b>25</b>
3.1	Goal . . . . .	25
3.2	Introduction to OpenID . . . . .	25
3.3	Basic Elements in OpenID 2.0 . . . . .	28
3.4	Creating an OpenID . . . . .	30
3.4.1	OpenID Providers (OPs) . . . . .	30
3.5	Using OpenID . . . . .	32
3.5.1	Simple Registration Extension (SREG) . . . . .	40
3.6	Multi-Factor Authentication . . . . .	42
3.6.1	One-Time Password (OTP) . . . . .	42
3.6.2	Digital Certificates . . . . .	47
3.6.3	Other Methods . . . . .	49
3.7	Provider Authentication Policy Extension (PAPE) . . . . .	49
3.7.1	Defined Authentication Policies . . . . .	50
3.8	Phishing Protection . . . . .	53
3.9	Certification of OPs . . . . .	55
3.10	Summary . . . . .	56

<b>4</b>	<b>Security Assessment</b>	<b>57</b>
4.1	Goal . . . . .	57
4.2	Application Development . . . . .	57
4.2.1	Java Library: openid4java . . . . .	58
4.2.2	Spring Security . . . . .	59
4.2.3	Functionality . . . . .	60
4.3	Security Assessment . . . . .	62
4.3.1	Attacking an RP Using CSRF . . . . .	62
4.3.2	Countermeasures Against CSRF . . . . .	70
4.3.3	Attacking an OP Using Clickjacking . . . . .	70
4.3.4	Countermeasures Against Clickjacking . . . . .	78
4.4	Summary . . . . .	81
4.5	Discussion . . . . .	81
<b>5</b>	<b>Conclusion and Further Work</b>	<b>83</b>
5.1	Evaluation and Conclusion . . . . .	83
5.2	Further Work . . . . .	85
<b>A</b>	<b>OpenID</b>	<b>89</b>
A.1	Signature Calculation in OP Response . . . . .	89
<b>B</b>	<b>eXtensible Resource Descriptor Sequence</b>	<b>89</b>
B.1	Supported Authentication Policies at VeriSign . . . . .	89
<b>C</b>	<b>YubiKey</b>	<b>90</b>
<b>D</b>	<b>Cross-Site Request Forgery (CSRF)</b>	<b>93</b>
D.1	Countermeasures . . . . .	93
<b>E</b>	<b>Clickjacking</b>	<b>94</b>
E.1	Clickjacking Attack Against Google OP . . . . .	94
E.2	Protection Mechanisms . . . . .	96
<b>F</b>	<b>Web Application</b>	<b>97</b>
F.1	Log Messages During Sign-On . . . . .	97
F.2	Attached Source Code . . . . .	97

## List of Figures

1	Sign-on at two SPs using a FEIDE identity . . . . .	10
2	SLO example . . . . .	11
3	A selection of some popular IdPs . . . . .	13
4	CSRF scenario . . . . .	18
5	Clickjacking [15] . . . . .	19
6	Man-in-the-Middle [16] . . . . .	20
7	Original IdP: <a href="https://pip.verisignlabs.com/login.do">https://pip.verisignlabs.com/login.do</a> . . . . .	22
8	IdP phishing: <a href="http://pip.verisignlabsc.com/login.do">http://pip.verisignlabsc.com/login.do</a> . . . . .	22
9	Example of an OpenID URL . . . . .	27
10	Basic elements of OpenID . . . . .	29
11	VeriSign Labs PIP - Create Account . . . . .	33
12	VeriSign Labs PIP - My Information . . . . .	34
13	<a href="https://sourceforge.net/account/login.php">https://sourceforge.net/account/login.php</a> . . . . .	34
14	VeriSign Labs PIP - Sign In . . . . .	35
15	VeriSign PIP requesting verification . . . . .	36
16	New account created using an OpenID . . . . .	38
17	Signing into a <i>second</i> OpenID-enabled Web site . . . . .	39
18	VeriSign VIP on Sony Ericsson mobile phone . . . . .	43
19	YubiKey USB-key connected to key chain . . . . .	44
20	Using the YubiKey at <a href="http://clavid.com">clavid.com</a> . . . . .	45
21	Multi-factor authentication using a digital certificate . . . . .	48
22	Multi-factor authentication with CallVerifID . . . . .	49
23	Yahoo! OP with and without sign-on seal . . . . .	54
24	Web application sign-on page . . . . .	60
25	Agreement to share information at OP . . . . .	61
26	RP account details . . . . .	62
27	“Manage My OpenIDs” . . . . .	64
28	CSRF attack . . . . .	66
29	Results of CSRF attack using HTTP GET . . . . .	67
30	Results of CSRF attack using HTTP POST . . . . .	69
31	Authentication with Google OP . . . . .	71
32	Authentication with Google OP located inside an <code>iframe</code> . . . . .	72
33	Fake page presented to the victim . . . . .	73
34	Reducing opacity of OP authentication form layer . . . . .	74
35	Successful Clickjacking attack . . . . .	75
36	Password settings at <a href="http://clavid.com">clavid.com</a> . . . . .	76
37	A fake page with a commenting form . . . . .	77
38	Partially opaque <code>iframe</code> containing <a href="http://clavid.com">clavid</a> page . . . . .	78
39	YubiKey control flow [14] . . . . .	92

40	The HTTP X-FRAME-OPTIONS header . . . . .	96
----	---	----

# List of Tables

- 1 OpenID SREG fields . . . . . 40
- 2 NIST authentication mechanism levels . . . . . 52
- 3 The Same Origin Policy . . . . . 63
- 4 OP comparison of Clickjacking vulnerabilities . . . . . 80

## List of Listings

1	CSRF using an <code>img</code> element . . . . .	17
2	XRDS document example . . . . .	51
3	Spring Security configuration example . . . . .	59
4	Form used for associating additional OpenIDs . . . . .	65
5	CSRF using HTTP <code>GET</code> . . . . .	65
6	CSRF using HTTP <code>POST</code> . . . . .	68
7	Inclusion of Google OP in an <code>iframe</code> . . . . .	71
8	Frame busting code . . . . .	79
9	The contents of VeriSign PIP's XRDS file . . . . .	90
10	CSRF examples . . . . .	93
11	Usage of unpredictable token to avoid CSRF . . . . .	94
12	Code used during Clickjacking attack against Google OP . . . . .	94
13	Logging produced during sign-on . . . . .	97

## Abbreviations

**AES** Advanced Encryption Standard

**AJAX** Asynchronous JavaScript and XML

**ARP** Address Resolution Protocol

**CA** Certification Authority

**CSP** Content Security Policy

**CSRF** Cross-Site Request Forgery

**ECB** Electronic Codebook

**HMAC** Hash-based Message Authentication Code

**IdP** Identity Provider

**MITM** Man-in-the-Middle

**NIST** National Institute of Standards and Technology

**OIDF** OpenID Foundation

**OP** OpenID Provider

**OTP** One-Time Password

**PAPE** Provider Authentication Policy Extension

**RP** (OpenID) Relying Party

**OWASP** Open Web Application Security Project

**SAML** Security Assertion Markup Language

**SLO** Single Logout

**SOP** Same Origin Policy

**SP** Service Provider

**SPOF** Single Point of Failure

**SREG** Simple Registration Extension

**SSL** Secure Socket Layer

**SSO** Single Sign-On

**VIP** VeriSign Identity Protection

**XRDS** eXtensible Resource Descriptor Sequence

**XRI** eXtensible Resource Identifier



# 1 Introduction

## 1.1 Motivation

A large problem of today's Internet is how many of its users are forced to handle a large number of user accounts spread around at different domains. A common way for Web sites to create such accounts is to make the user choose a username accompanied by a password. The username normally consists of an e-mail address, or simply a nickname. Web sites often require more pieces of information during registration than these, but what the user needs to remember to be able to log in at a later time is basically the username/password combination.

Some users have only a few such accounts, and for them it is relatively easy to remember the username/password combinations. For other users, however, the total amount of accounts might grow so large that even remembering just the usernames at every Web site becomes too hard. Imagine a user having 40-50 different Web site accounts. It is obvious that remembering a completely different password for each site in such a situation is something that very few people can handle.

Some people solve this problem by using the *same password* at every Web site they use. Obviously, this is not a secure way of managing passwords. Most information security experts encourage users **not** to reuse identical passwords for several Web sites. The reason for this is that once it is compromised on one of the sites, it is easy for an attacker to login with any of the other accounts belonging to the user. Also, one cannot trust that every Web site has implemented password handling in a secure manner. Some sites even store passwords in plaintext!

Another solution is to pick a relatively strong password, and use *variants* of it for each user account. As an example, let us say that a user has chosen P8\$Tr7@mP as a default password. Now, if this user wants to register a new account at google.com, a variant could be made by picking the leading character and substituting the first and the last character, hence producing the password G8\$Tr7@mG. Another example would be Y8\$Tr7@mY for an account at yahoo.com. Following this rule, the user would have only one fixed password to remember, and at least it would not be identical across every Web site (except for those starting with the same letter, of course). However, it is still not impossible for an attacker to figure out this way of doing it. By stealing two or more passwords from a user, it would eventually be possible to figure out the adopted pattern.

As we see, there are many ways people can manage their passwords. And some people choose safer methods than others. However, there will always

be those who choose weak passwords and those who use the same password with every Web site. In addition, there are different password requirements at different Web sites. Some sites actually allow weak passwords like `asdf` or `1234`, while other sites have requirements like a minimum-length of 8 characters and the inclusion of both numbers and letters. For users with little experience using the Internet, writing down complex passwords instead of memorizing them might seem like the easiest solution. But having dozens of post-its floating around with different username/password combinations is not exactly the most secure way to go either.

## 1.2 Related Work

Several articles and theses addressing information security related to SSO exist. One example is [12], a thesis which discusses various threats against SSO mechanisms. It focuses on which types of Web security threats that can be realized, or that become more apparent, as a result of adoption of SSO. Three threats were identified:

1. Exploitation of IdPs as Single Point of Failure (SPOF).
2. Lacking or insufficient implementation of Single Logout (SLO).
3. Increased danger of phishing.

Note that terms like IdPs, SLO, and phishing will be described in Section 2 (*Theoretical Background*).

Another related work, [16], consists of a security evaluation of the OpenID protocol,<sup>1</sup> which is an implementation of SSO. The work puts emphasis on evaluating the protocol using an analysis tool, and attempts to discover security weaknesses. During the analysis, a security weakness of the OpenID protocol was discovered. This weakness makes it possible for an attacker to impersonate a user during authentication (Man-in-the-Middle). The thesis concluded that the OpenID protocol messages need integrity protection to avoid the attack, which can be done by applying the Secure Socket Layer (SSL) protocol.

## 1.3 Refinements and Limitations

The practical part of the thesis (*4. Security Assessment*) includes implementation of example applications utilizing SSO, more specifically the OpenID protocol. In this case, the implementation only includes development of

---

<sup>1</sup><http://openid.net>

OpenID *Relying Parties* (RPs), and **not** development of OpenID *Providers* (OPs). Instead, existing providers available on the Internet are used. Also note that a security assessment directed towards the OpenID protocol itself is out of scope for this thesis, as the focus is rather directed at SSO-functionality in general.

The thesis will not cover details about every single attack that can possibly be performed against an SSO Web application. Instead, some of the most relevant threats will be chosen, and later used for attack scenarios in the practical part. Also, note that the thesis focuses on the investigation and identification of security threats, but not as much on their corresponding countermeasures. That is, countermeasures for each attack *will* be mentioned, but using a relatively low level of details. Instead, the main efforts will be put into showing how to actually perform the attacks.

Also, it is well worth pointing out that certain discrepancies based on the problem description have arisen during the process of writing this thesis. While the original problem description states that the thesis is “An Assessment of the Security in and Between Web Applications Sharing a Common Single Sign-On User Session”, the focus switched gradually to Web applications not necessarily belonging to the same SSO session as the Web application under attack. The original idea was defined as the following, according to the text in the problem description:

“[...] the main focus will be on how information security is safeguarded between two Web applications that belong to the same SSO user session. [...]”.

Hence, it should be noted that other scenarios are investigated as well, e.g. a situation where the same user first logs into an SSO-enabled Web application, and later visits a malicious Web site that is **not** utilizing SSO. So, it should be taken into account that there are certain contradictions between the problem description and the contents of the thesis, as the work has progressed and focus has changed throughout the process. These new directions of focus have been made in accordance with guidance from the supervisor.

## 1.4 Thesis Outline and Methodology

The primary content of the thesis is roughly divided into three parts:

1. Theoretical Background
2. OpenID
3. Security Assessment

The main part of the thesis begins with theoretical background material (Section 2). This part explains the concept of SSO and IdPs. Additionally, an introduction to relevant Web security threats is given.

The part that follows is dedicated to the OpenID protocol alone (Section 3). OpenID is an example of a popular implementation of SSO, and this section gives an introduction to the way it works and what elements it is comprised of. As part of the introduction, this part starts by giving a step-by-step explanation of how to create an OpenID identity and how to use it. Also, this part addresses various parts of the OpenID specification, e.g. two extensions; *Simple Registration Extension* (SREG) and *Provider Authentication Policy Extension* (PAPE). Also, the part describes how *Diffie-Hellman* (DH) key exchange is used during authentication. A known challenge for the OpenID protocol is that it is vulnerable to phishing attacks. For this reason, a section is dedicated to demonstrating ways IdPs can prevent such attacks. OpenID is also the protocol that will be used for the application implementation in the section that follows.

The last part (Section 4) deals with the practical pieces of the thesis. Here, there is given a description of a Web application developed as part of the thesis, and this implementation is used for security assessment of SSO. The application enables users to sign on using an OpenID identity. The security assessment consists of experimenting with Web attacks like *Cross-Site Request Forgery* (CSRF) and *Clickjacking*. The attacks will mainly address the following two scenarios:

1. A malicious Web site attacks an SSO-enabled Web site, i.e. a *Service Provider* (SP).
2. A malicious Web site attacks an IdP.

In both of these cases, the malicious Web site might itself be utilizing SSO. The assessment also includes looking at phishing attacks, as well as looking at how information is exchanged between SSO Web applications and IdPs.

**Methodology** The main results in this thesis have been achieved through Web application implementation and experimentation, being backed up by theoretical background material. In the beginning of the thesis, basic theory about SSO is presented, as well as the motivation behind the work. Some time was reserved for gathering reference material related to SSO, with a focus on finding information about potential attacks.

A significant amount of time has also been used to get acquainted with the OpenID protocol and its specifications. Additionally, quite a lot of effort has been put into testing different types of IdPs available on the Internet.

Since a part of the thesis consists of assessing information security between Web applications and IdPs, it was considered necessary to register with and use a relatively high number of different providers in order to get an overview of the most known providers that exist today.

A central part that has been focused on throughout the thesis is a collection of questions defined at the very beginning of the writing process. Some of the questions can be considered as assumptions, looking for confirmation (or falsification) as the level of knowledge were to increase throughout the work. Other questions are more open, addressing topics that are pretty much unknown, i.e. things that are desired to be clarified underway. These are all the questions that were defined (presented in groups):

- New users of SSO
  - Is it easy to make security mistakes as a new user of SSO?
  - Is it necessary with a thorough education of new users?
- Information disclosure
  - Will SSO Web applications be able to access private information from the IdP where the user is currently logged in?
- SSO-related technology
  - How is user information (e.g. name/e-mail) transferred from an IdP to an SSO Web application?
  - Which technologies are being used?
- Security requirements
  - Is SSL encryption required by most IdPs, or is this optional?
  - If so, are there any requirements regarding the strength of SSL encryption?
  - Might poor handling of security at one IdP affect other IdPs?
- Provider filtering
  - Can an SSO Web application choose to allow only certain IdPs?
  - Is it possible to filter those that fulfil a certain degree of security requirements during authentication (e.g. two-factor authentication)?
- Differences from traditional Web applications

- When a user signs out of an SSO Web site, is she signed out of all the others at the same time?
- Does signing out of an SSO Web applications entail a sign-out from the IdP?
- SSO security threats
  - What kind of Web security threats are SSO Web applications typically suffering from?
  - Are traditional attacks like Man-in-the-Middle (MITM), phishing, and CSRF still relevant?

Note that each question is not necessarily covered using an equal amount of text and time. As an example, Web security attacks have received more attention than topics like user-friendliness and user-education. The goal has been to prioritize themes most relevant for the thesis, while still including information that is useful in order to gain a good overview.

## 2 Theoretical Background

### 2.1 Goal

This section will give an introduction to some theory relevant to SSO. This will provide the background information necessary for the sections that follow (Section 3 *OpenID* and Section 4 *Security Assessment*). First of all, this part will describe the concept of SSO and how it can be applied. Next, *Single Logout* (SLO) will be discussed, as well as *multi-factor authentication* at IdPs.

The section will then continue with a part that addresses security threats on the Internet, i.e. those that are most relevant to SSO systems. Theoretical background will then be given for attacks like CSRF and Clickjacking. The section ends with a description of phishing.

### 2.2 Single Sign-On (SSO)

As explained in the introduction, there is obviously a craving need for better management of user accounts on the Internet. This is where *Single Sign-On* (SSO) comes into play. SSO is a solution where a user only needs to login with a username/password combination once for each browser session, no matter how many different Web sites are being visited. The idea is that the Web sites supporting SSO do not store the passwords of its users; they rather obtain user information from an IdP. When a user wishes to login to an SP, a redirection to an IdP is performed. If the user has not already created an identity with the IdP, she has to go through the registration process before proceeding. Next, the user types the username/password combination<sup>2</sup> to login with the IdP. During this step, the user is prompted for which pieces of information to share with the requesting Web site. Next, the user is redirected back to the original Web site, and provided information like nickname and e-mail address can be handled. At this point, the Web site can consider the user as authenticated.

If the same user later decides to use another SP, then the login process will be even easier. Since the user has already signed on with the IdP, no username/password is needed since she is considered to be authenticated for the current session. So, for subsequent logins at Web sites supporting SSO, all that is needed is confirmation from the user that the requesting Web site is being signed into using her identity. This is why it is called Single Sign-On; the user only has to provide login details *once* for each browser session.

---

<sup>2</sup>The authentication can be performed using other methods than a username/password combination, e.g. usage of digital certificates. This depends on the current IdP.

A goal of SSO is to improve user-friendliness. It is beneficial for users to be presented with a login and registration process that is faster and easier. The frustration of not remembering username/password combinations can also be reduced. And as mentioned above, the password management will be a less demanding task for the average user. Not only the users might benefit from the usage of SSO; also the Web sites themselves might experience advantages while offering their services:

- The conversion of *visiting* users into *registered* users might increase its frequency due to a simpler registration procedure.
- Reduced number of inquiries from users/customers regarding forgotten passwords (i.e. lower help desk costs).
- Users can easily share as much of their existing personal profile as they want with the Web sites. This increases the probability that the Web site will obtain additional information about a user (e.g. date of birth).
- The Web site can provide a more personalized user experience (as a result of the gathering of additional user data).

Obviously, there are not only advantages related to the usage of SSO; several drawbacks exist as well. One of them is that IdPs might become bottlenecks in the systems, and SPs are depending on a relatively high up-time with regards to their functionality. This means that IdPs might represent Single Points of Failure (SPOFs). And, of course, if they were to be compromised, this might result in enormous damage. Security threats against IdPs and SPs will be addressed throughout the rest of the thesis.

### 2.2.1 Components

Jan De Clercq (Security Consultant, HP) defines SSO as the following [4]:

“SSO is the ability for a user to authenticate once to a single authentication authority, and then access other protected resources without re-authenticating.”

According to this quoted definition, and as indicated earlier, an SSO system consists of various components. We can say that there is a “circle of trust” [3], containing the various participants. This is a federation of SPs and IdPs that have business relationships, and with whom End Users can transact business. Basically, each of the classified participants can be described like this:



1. **End User** - An End User is an actual human being, whose goal is to prove his/her identity to an SP.
2. **Service Provider (SP)** - An SP (also known as “Consumer”) is a service-providing Web site that is looking for proof that a given End User owns the provided Identifier.
3. **Identity Provider (IdP)** - An IdP (also referred to as “Server”) is the authentication server that an SP contacts for proof that the End User owns the claimed identity.

### 2.2.2 Possible Applications

The concept of SSO spans a wide area of technologies and application areas. Companies might choose to implement SSO as part of their Intranet, while others focus on the usage of SSO on the Internet (being the case in this thesis). Some people have only one SSO identity, and others might choose to use several different identities; one for each category of Web sites. Different Web sites might not have equal security requirements, so using one identity system for all the sites might lead to problems because of different needs. So a user could decide to choose between several identities, depending on the current usage area. As an example, an End User might pick an IdP with a high security level for usage with some SPs, while choosing a less secure, but more user-friendly, IdP for SPs that handle less sensitive information.

SSO systems can be distinguished between two main types; *pseudo-SSO* and *true SSO* [5]. In a pseudo-SSO system, there are still being used multiple username/password combinations, one for each SP. The difference is that there is a component dedicated to storing every combination for later use. So, the first time a user visits a Web site requiring authentication, she types in her username/password for the current site. Depending on the SSO system, the credentials are stored at a given location, e.g. on a server or locally on the computer (obviously, the storage needs to be done in a secure manner). And when the user re-visits one of the SPs stored in the system, the sign-on procedure is performed automatically. This means that the user needs to know the SP credentials for the first time she signs on, but not afterwards. To gain access to the whole system, the user needs to authenticate herself. This step is called *primary authentication* [5]. A pseudo-SSO system can be a browser plugin, OS software, etc. One example is *LastPass*,<sup>3</sup> which is a password manager that works with multiple browsers and operating systems. Other alternatives exist as well.

---

<sup>3</sup><http://lastpass.com>

Hence, pseudo-SSO does **not** implement a complete authentication solution [16]. The *true SSO* system, however, requires that the participants all use the same authentication solution when interacting with each other. This is the type of SSO that is addressed in this thesis. In such a system, the service-providing Web sites do not need to have their own authentication system; instead only **one** set of credentials is needed. I.e., no passwords are shared with SPs in a real SSO system. Examples of true SSO systems are *OpenID*, *Kerberos*, and *Microsoft Windows Live ID*.

## 2.3 Single Logout (SLO)

As it is closely related to SSO, it is also important to mention the concept of *Single Logout* (SLO). When a user has finished her task at an SP, she might want to log out from that particular Web site. Imagine a scenario where an End User has used an account at an IdP in order to sign on at two different SPs. If the current SSO system supports SLO, a logout from the IdP or one of the SPs would entail a logout from the other RP and the IdP itself.

An example of an SSO system supporting SLO is *FEIDE*,<sup>4</sup> a Norwegian IdP. First, consider a scenario where a user signs on at two different SPs utilizing an identity from FEIDE, as demonstrated in Figure 1.

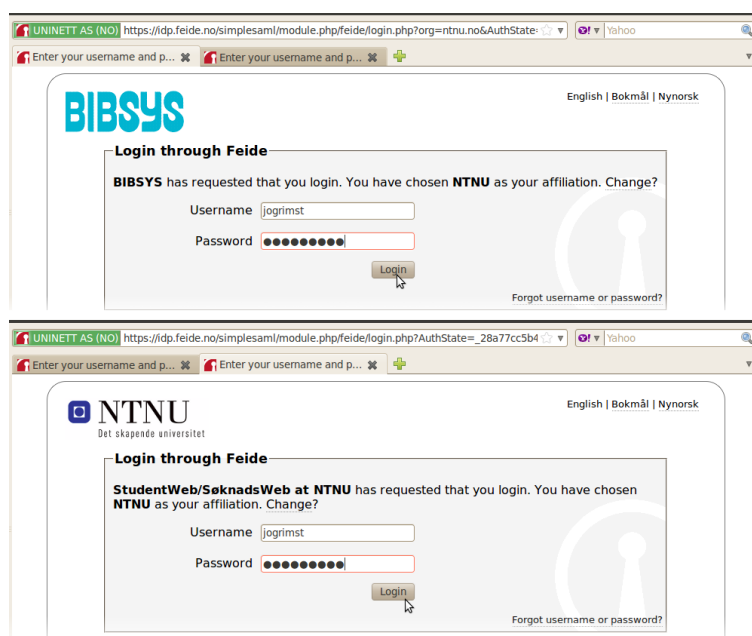


Figure 1: Sign-on at two SPs using a FEIDE identity

<sup>4</sup> <http://www.feide.no/>

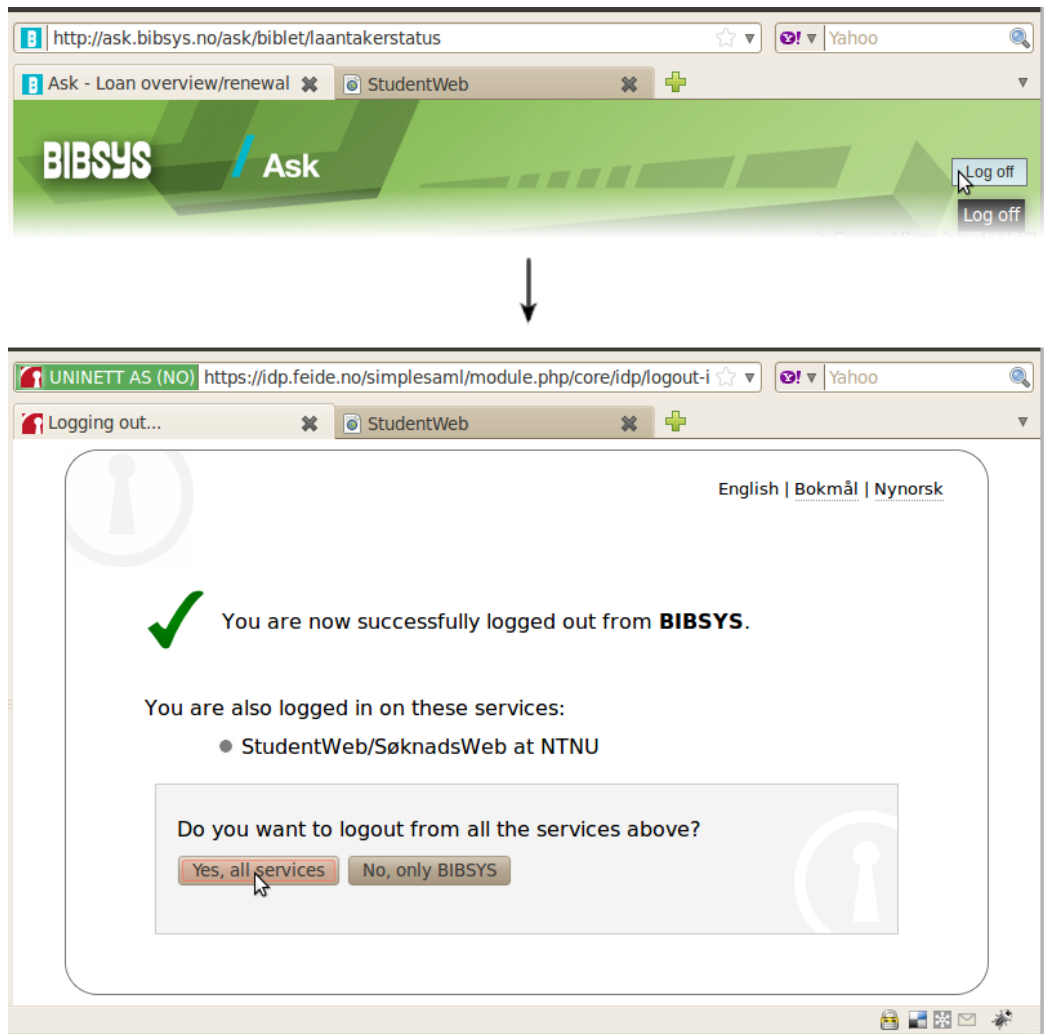


Figure 2: SLO example

When signed on, the user can choose to log out from one of the Web sites. Figure 2 shows that after clicking the “Log off” button at the BIBSYS Web site, the user is presented with the following logout alternatives:

1. Log out from *all* the services.
2. Log out from the current SP only.

If the user chooses “Yes, all services”, she will be logged out from both of the SPs and from the IdP (FEIDE). If she chooses “No, only BIBSYS”, she will be logged out from BIBSYS and the IdP, but **not** from NTNU StudentWeb.

Actually, SLO functionality depends on the SSO system that is being used. As an example, OpenID does **not** support SLO, while others do (e.g.

OpenSSO<sup>5</sup>). With OpenID, the logout scenario above (a user being signed on at two SPs via an IdP) would require the user to act in a completely different way. As above, she would have to start by logging out of the current Web site. However, this does not mean that the user should be satisfied and leave the computer. Actually, when using OpenID, she would still be signed on to the other SP, as well as the IdP! Hence, for a complete logout, she would have to manually visit the other SP and the IdP in order to log out, or close the browser.

Obviously, a lack of SLO functionality might be confusing for users that are not familiar with the concept of SSO. And the fact that some SSO systems support it while others do not, is also a problem. When logging in at traditional Web sites using username/password credentials, most users are aware that they should log out from the sites when they are finished. But with the spread of SSO Web applications, users might feel confused since SLO functionality might be implemented in some systems, while missing at others. A possible scenario is the usage of SSO at public computers. A user might use her OpenID identity to sign on with an SP, then log out from the SP, and leave the computer. In this case, the session at the IdP remains active, available for misuse. Hence, it is clear that the transition from traditional Web applications to SSO Web applications requires some form of education of new users, and that a lack of support for SLO might lead to security threats [12].

## 2.4 Identity Providers (IdPs)

When someone wants to use SSO in order to authenticate with an SP, an identity must have been created first. An identity is provided by an *IdP*, so the first thing that is needed to be done, is to choose which IdP to use. Here, there are a lot of different possibilities. Figure 3 shows a selection of various IdPs. This is a screenshot grabbed from a social widget interface called *Engage*, created by *Janrain*<sup>6</sup> - a company specializing in user management for the social Web. It illustrates how End Users can choose amongst different IdPs when signing on at an SP. An advantage of this solution is that there is a very large propability that the users already have a registered account at one or more of those providers, thus eliminating the need for going through with a new registration.

However, if a user has not registered with any of the IdPs, she needs to choose one. This might be difficult for a completely new user with little

---

<sup>5</sup>OpenSSO: <https://opensso.dev.java.net>

<sup>6</sup>Janrain: <http://www.janrain.com>

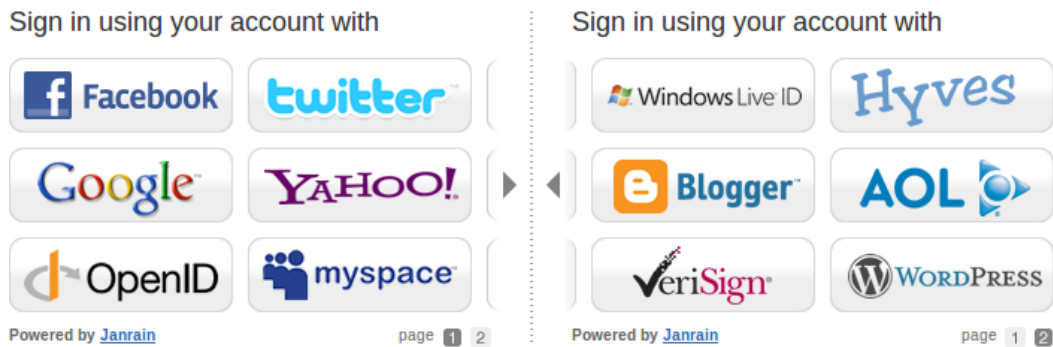


Figure 3: A selection of some popular IdPs

or no experience with SSO systems (or Internet in general). At least, with Janrain’s solution, the user is presented with a range of alternatives to choose between. If using OpenID, on the other hand, the End User has to look for OpenID IdPs herself in order to obtain an identity. Considering the fact that OpenID IdPs vary quite a lot with regards to the offered security level, this might be experienced as problematic for a fresh user.

Many Web sites have also chosen to introduce sign-on buttons for only a few, but well-known, IdPs (e.g. *Facebook*). It is entirely up to the SP in question which IdPs that should be allowed. Consider



a Web site containing blog articles, where readers are allowed to post comments. In this case, the site owner might wish to only allow Facebook users to post comments, thus removing the problem of spam produced by anonymous users (and robots). So, if visiting End Users are already signed on with their Facebook account, they can post comments immediately. Otherwise, they are typically provided with a login popup when clicking the “Login with Facebook” button. Similar functionality is available at other IdPs, like Yahoo! and Twitter.

IdPs for governmental services also exist. As an example, in Norway, a provider named *MinID*<sup>7</sup> (eng: MyID) has been established by the government. Norwegian citizens above the age of 13 can register with this IdP in order to gain access to various public services, like student loan, medical services, and tax services. The number of End Users registered with this IdPs is currently approaching half of Norway’s population.

*FEIDE*<sup>8</sup> is another Norwegian IdP, and stands for *Felles Elektronisk IDEntitet* (eng: Joint Electronic Identity). The Ministry of Education has

<sup>7</sup>MinID: <http://minid.difi.no>

<sup>8</sup>FEIDE: <http://www.feide.no>

decided that every school in the country is to be offered usage of FEIDE by the end of year 2010. Using this IdP, students can gain access to SPs like library services, gradings, etc. The previous section (*2.3 Single Logout*) showed an example where FEIDE was used.

When picking an IdP, a user's *trust* in the provider is essential. After all, the chosen IdP gets a lot of information about its End Users:

- User data like name, e-mail, postal address, etc.
- Information about which sites the users visit, i.e. which SPs users sign on with using their identity provided by the IdP.
- User habits, e.g. how frequently and at what time of the day SPs are visited.

With such information in hand, the IdP can create a collection of complete user profiles to be used as recipients of marketing, spam, etc. IdPs also holds sign-on credentials belonging to their users, so a security breach would obviously be fatal, as this might enable attackers to sign on at Web sites associated with the identity at the compromised IdP. Also, if sufficient user information is stored at the IdP (e.g. date of birth, postal address, and social security number), a security breach might result in identity theft [1].

For this reason, End Users face a trust issue when they find themselves in the process of choosing an IdP. First of all, they need to feel confident that the IdP will not misuse the registered user information, nor share it with any third-party companies. Also, many users expect their choice of IdP to be of one that takes security seriously, i.e. one that has implemented protection mechanisms sufficient enough to withstand common attacks.

When speaking of trust, many users might choose an IdP that is a large and well-known company over a smaller, and relatively unknown, IdP. For instance, people might decide to use Google as IdP instead of VeriSign, even though VeriSign is the only one of them which provides multi-factor authentication (multi-factor authentication will be described in the following section). In this case, it can be said that Google is picked because the user knows that Google is a large company, while VeriSign is a company that she has never heard about. VeriSign actually provides better security during authentication than Google, but that does not help if users have never heard about the better alternative, or if they do not have any trust in it.

So, a good rule is to pick an IdP that has a good reputation from a security perspective [1]. This, however, depends on user knowledge about the many different alternatives that exist all over the Internet. Also, for those who have little or no trust in third-party IdPs, it is possible to run your own

IdP. This way, privacy issues related to third-parties would be eliminated, but the responsibility of securing the data in a proper manner would then be transferred to you, as being the owner of the identity server.

### 2.4.1 Multi-Factor Authentication

When signing on with an IdP, a user needs to be authenticated before she is given access to her account. There are several different ways users can be authenticated; it is up to the IdP how to implement it. However, it is natural to categorize some *factors* of authentication [1]:

1. **Ownership factor**

Something the user *has*, e.g. mobile phone or hardware token.

2. **Knowledge factor**

Something the user *knows*, e.g. a password or a PIN-code.

3. **Inherence factor**

Something the user *is* or *does*, e.g. fingerprint or voice.

Each of these three authentication factors cover their range of elements that can be used for authentication of users. As of today, the *knowledge factor* is the factor that is most used on the Internet, normally in the form of a username/password combination.

To gain access to applications that contain sensitive information (e.g. banks), usage of the knowledge factor alone is normally not enough. In this situation, applications may require a combination of two authentication factors. An example is to combine a password (knowledge factor) with a hardware token (ownership factor), which is an application of the term *two-factor authentication*. Basically, *multi-factor authentication* is the situation in which two or more authentication factors are combined. Multi-factor authentication can also be replaced with *strong authentication*, which can be defined as the following:<sup>9</sup>

“Layered authentication approach relying on two or more authenticators to establish the identity of an originator or receiver of information.”

Some real-life examples of uses of multi-factor authentication will be given in Section 3.6, where authentication techniques of various OpenID IdPs will be presented.

---

<sup>9</sup>CNSS National Information Assurance (IA) Glossary: [http://www.cnss.gov/Assets/pdf/cnssi\\_4009.pdf](http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf)

## 2.5 Web Security Threats

The benefits of using SSO are many, but what about the *security*? Does the increased user-friendliness affect the security handling in any way? And are there any Web security threats that become more severe if SSO is in use? These questions form the basis of this Master thesis.

SSO can make the life easier for many End Users, but there are also criticisms. The fact that all of a user's accounts can be accessed through one single authentication mechanism means that this particular entrance becomes more vulnerable to attacks. Because a user's SSO identity functions as a way to access *all* of her SSO-enabled Web site accounts, this "Single Point of Failure" (SPOF) should be secured in a good manner. When comparing with a single (traditional) Web site account, the login using an SSO identity should be protected using *stronger authentication*. As an example, a two-factor authentication method using password and one-time passwords via SMS might be useful in some situations where improved security is desirable. The degree of security depends, however, on what type of Web sites the identity is used for. An account used for blog commenting does not require as much protection as an account used for e-commerce.

Additionally, as of today the concept of SSO is still not widely known to the average user. Assumably, this will change during the next years, but until then there will be many users who might react in a skeptical way to this new way of signing on to Web sites. Some might hesitate because of the fact that someone with access to their SSO identity can access all their Web sites. For people who are using the same password for every site this would not be a security degradation, since an attacker would still be able to access all the user's accounts by obtaining the password from one of the sites. But for people who are more security-aware, it might be harder to find a way to inform how using an SSO identity might be a better solution than having several (strong) passwords spread around. In order for these users to feel secure, it is more important to inform that the IdP in question is trustworthy and that it supports strong authentication, which was described in the previous section (*2.4.1 Multi-Factor Authentication*).

This also leads us to the problem with user-friendliness contra security. Higher security requirements normally leads to degraded user-friendliness and vice versa. For SSO to be widely adopted, it is necessary to find solutions that cover the security needs for most users without being too invasive. To be able to determine what security measures that should be employed in SSO Web applications, it is first necessary to get an overview of the types of attacks that are relevant. In the following, some Web security threats related to SSO Web applications will be outlined.



### 2.5.1 Cross-Site Request Forgery (CSRF)

One of the Web security threats that SSO Web applications are facing is *Cross-Site Request Forgery* (CSRF). This is an attack that tries to exploit the fact that users are authenticated and logged in at other pages [10]. As an example, assume that an innocent user has been authenticated to access her account at `http://bank.com/account.php`. For the simplicity of the example, let us assume that a request to the following URL would execute a money transfer:

```
http://bank.com/account.php?transferTo=1234.12.1234&amount=100
```

If the user goes to visit a malicious page (without logging out of the bank), great damage can be done if the bank has failed to handle CSRF attacks. An attacker could simply trick the user into requesting the URL above, e.g. by including an image on the page:

Listing 1: CSRF using an `img` element

---

```
1 
```

---

This would have caused a transfer to be executed without the victim knowing about it, given that no countermeasures were employed beforehand. Also, since the width and height of the image is set to zero, the user would not see anything, only a blank page.

Such an attack is possible when Web applications wrongly assume that *all* requests coming from authenticated users are to be trusted. As shown in Figure 4, the attack basically requires the following steps to happen, in the given order:

1. A user authenticates with a Web site, `bank.com` (which will be attacked later on).
2. Using the same browser, the user visits another Web site, `attacker.com`, without first logging out of `bank.com`, thus leaving an active user session.
3. The current Web site, `attacker.com`, forces the user's browser to make a request to `bank.com` (without the user knowing about it).

The visit to `attacker.com` can be made either voluntary by the user, or it can be forced. Assuming that `attacker.com` is hosted on some server controlled by the attacker, the goal is to somehow make the user visit this site. As an example, the user might be directed to this site via hyperlinks placed on

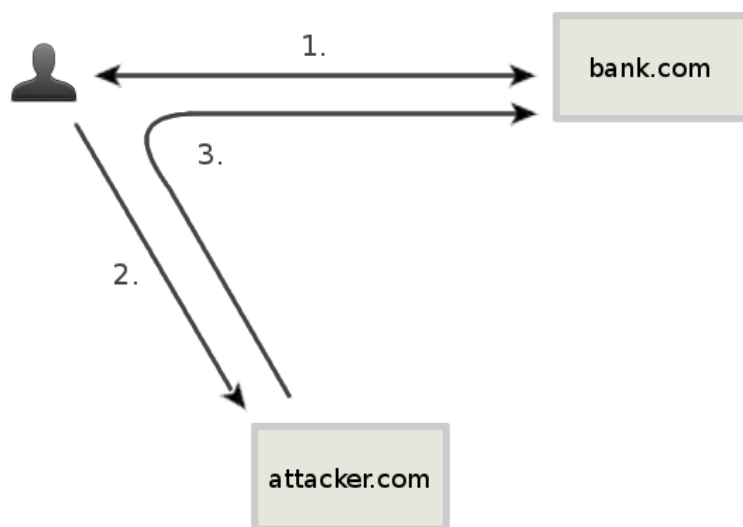


Figure 4: CSRF scenario

numerous sites, appearing to be a site being harmless to visit (the attacker might even fill it with ordinary content, making it appear more trustworthy). Another way to make users visit a page, is to place scripts at other Web sites, forcing redirection of users to the wanted URL. Also, an attacker might be able to send e-mail to a large number of recipients, hoping for some of them to click a link in the e-mail leading to the malicious Web site. Sending users links via Instant Messenger (IM) programs is another option. These are just a few examples of numerous ways to trick people into visiting a particular Web site.

Note that it is not only images (as demonstrated above) that can be used to forge requests; also `iframes`, scripts, and stylesheets are potential means (see Appendix D for code examples). And CSRF attacks are not exclusively related to SSO Web applications; it has been a threat on the Internet for several years already. However, the situation typically found with SSO, where users are signed on at multiple Web sites at once, makes CSRF a highly relevant attack vector.

Section 4.3.1 will provide real-life examples of how CSRF attacks might be performed against an OpenID-enabled Web site. For an introduction to countermeasures that can be used against CSRF, please take a look at Appendix D.1.

### 2.5.2 Clickjacking

As with CSRF, *Clickjacking* is another attack that can be used to force a user's Web browser into making a request to a site without the user's knowledge. While CSRF can be executed as a page loads its contents, Clickjacking is actually not initiated until the user submits the request herself. Using a combination of Cascading Style Sheets (CSS) and `iframes` (i.e. sub-frames), Clickjacking is able to *redress* the User Interface (UI) presented to the user, hence the synonym "UI redress attack" [11]. Clickjacking can also be used to circumvent the CSRF nonce protection mechanism (this concept is explained in Appendix D.1).

What happens in a Clickjacking attack is that the site under attack (the victim site) is included in a transparent `iframe` and put on top of a page that is apparently normal [15]. This is visualized in Figure 5; when users believe



Figure 5: Clickjacking [15]

that they are interacting with the normal page, they are in fact interacting with the overlapped victim site.

Theoretically, this attack might fool users into interacting with SSO Web applications without knowing about it. If a user has already signed on to her IdP, it might be possible to use Clickjacking in order to make her sign into and share data with a malicious SP. Section 4.3.3 of the practical part of the thesis will address this possibility in detail, followed by a description of countermeasures against Clickjacking in Section 4.3.4.

### 2.5.3 Man-in-the-Middle (MITM)

Another type of attack that is threatening SSO Web applications is *Man-in-the-Middle* (MITM). Unlike phishing (described in Section 2.5.4), an MITM attack does not attempt to imitate an SP or an IdP; instead the attacker's

focus is directed towards *intercepting data* that is sent between the End User and the IdP.

One way to achieve an MITM attack is through spoofing of the Address Resolution Protocol (ARP) in order to change the mapping of the IP addresses belonging to the victim [16]. This way it is possible to perform eavesdropping of traffic that pass through. Figure 6 illustrates the scenario. First, the user sends her identifier to the RP. Next, the RP establishes an

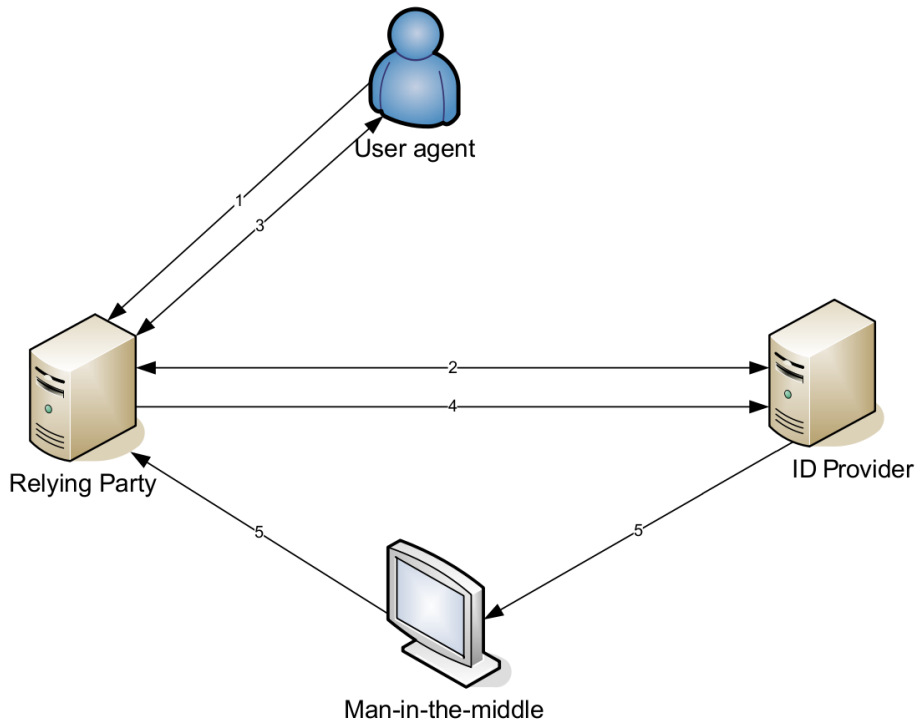


Figure 6: Man-in-the-Middle [16]

association with the IdP, and an authentication request message is sent. At this point, the IdP will either return a negative validation response or a positive validation response. Here, the attacker (the Man-in-the-Middle) can intercept a negative response, change it into a positive one, and then sending it to the RP. This would make the user signed in to the RP without actually authenticating with the IdP.

There are two different scenarios of an MITM attack in a SSO system; the MITM could be located at the RP, or it could be located at the IdP. If the MITM is located at the RP, it is possible to sign in a user at that particular RP using any IdP as a provider. Otherwise, if the MITM is located at the IdP, then users registered with that particular IdP could be authenticated to any RP [16].

### 2.5.4 Phishing

Because an SSO identity is the only entry point a user would have to her Web sites, attackers might consider it as lucrative to perform *phishing* (also known as *Web site spoofing*), i.e. misleading users into believing that they are in fact visiting the login page of an original IdP Web site when they are not. Depending on the implementation, this could also be considered as some kind of an MITM attack; the phishing site could simply forward the provided user credentials to the genuine IdP after saving them. This way of doing it would also make it less probable that the End Users would notice the attack.

A phishing site normally has a similar (or identical) design as the original, while the URL is slightly different from the original, yet hard to spot if not paying attention. When a user is tricked into visiting a fake site, she might end up providing login information that should only be shared with the genuine IdP.

The process of creating a Web page that looks exactly like the page that is being imitated is actually very easy. It can be done by performing the following steps:

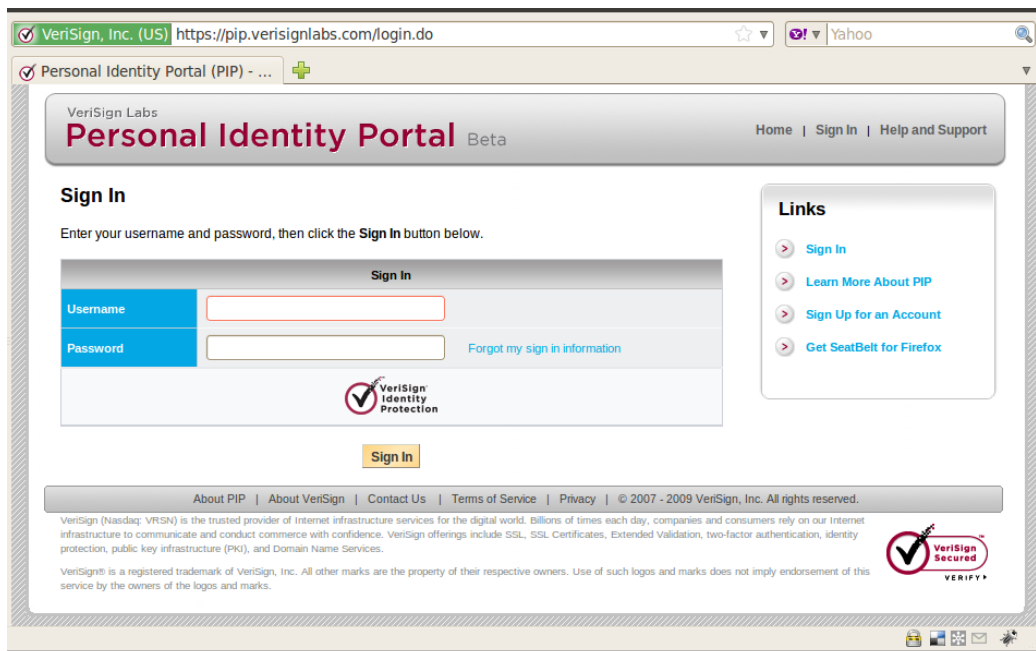
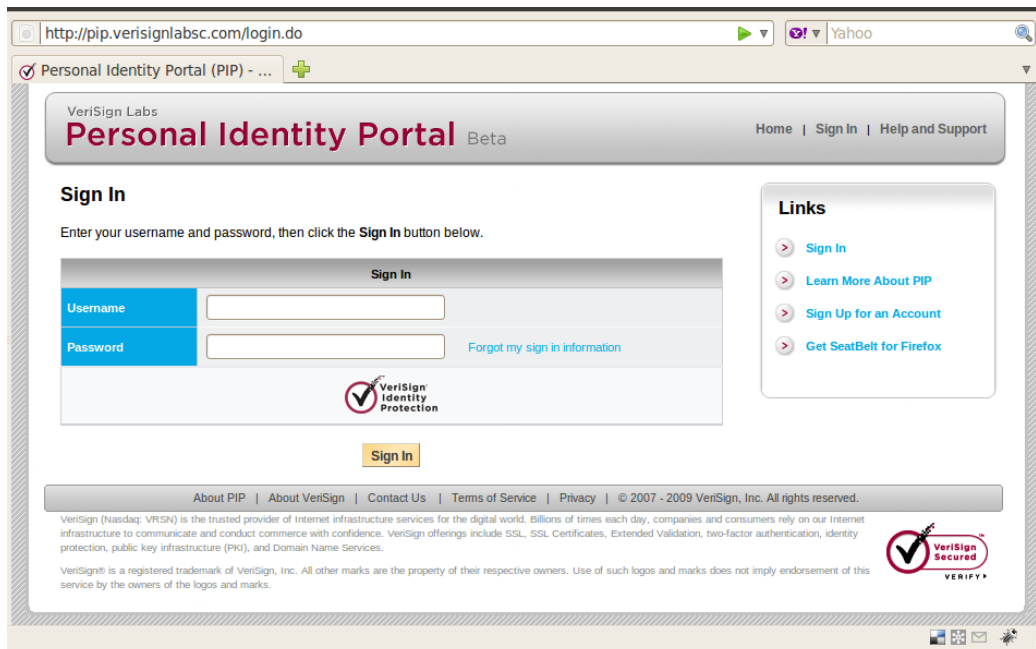
1. Copy the HTML source code of the genuine IdP login page.
2. Copy images, stylesheets, etc., needed for the design and layout to look complete.

It is that simple. It is a really quick process, and requires a minimum of technical skills. Note, however, that the attacker also probably would like to have the page located at a domain name looking very much alike the original one, e.g. something like this:

Original IdP: `https://pip.verisignlabs.com/login.do`  
Spoofed IdP: `http://pip.verisignlabsc.com/login.do`

Note that if the spoofed IdP uses HTTPS as protocol at the login page (which most IdPs do), then the attacker should also obtain a digital server certificate in order to be able to present the login page at a URL starting with `https://`. This would make it even harder for the user to become aware that something is wrong.

Figure 7 shows the original VeriSign IdP, while Figure 8 shows a phishing attempt. In this example, the phishing site lacks encryption, which might make users suspicious. But other than that, it is easy to understand that it can be hard to see the difference from the original. (Note that for this example, I did not actually register the domain `verisignlabsc.com`; for

Figure 7: Original IdP: `https://pip.verisignlabs.com/login.do`Figure 8: IdP phishing: `http://pip.verisignlabsc.com/login.do`

the simplicity of the example, I rather created a file locally, opened it in a browser, and simply changed the URL in the navigation bar into `http://pip.verisignlabsc.com/login.do` before taking the screenshot.)

For the attack to actually succeed, the attacker also needs to create a way to collect the credentials provided by victims of the attack, e.g. by saving them in a database or using some other storage mechanism. This, and domain registration and configuration, would probably be to most time-consuming part of such a phishing attack.

As will be described in Section 3.8 (*Phishing Protection*), there are things Web sites can do in order to make it easier for their users to protect themselves against phishing.

## 2.6 Summary

The traditional way of registering with a new username/password combination at every Web site requiring authentication is stuck in time; this way of user account management has not changed in over a decade, while the number of new services we use online has increased rapidly. Obviously, another technique for signing on at Web sites is needed, and *SSO* might be the solution.

However, there is a lack of user-familiarity with *SSO*. It is still not common place on most Web sites out there. This might change in the future, but no matter what, it is still necessary to assess the security of *SSO* Web applications. If the information security is not handled properly, the solution should never become common place. One security issue is the fact that an *SSO* identity would function as a single entry point to many accounts, thus requiring a higher level of security.

This part has provided a theoretical background to some of the concepts related to *SSO*. First, there was given an introduction to the functionality of *SSO* itself (Section 2.2). Then, a description was given for each of the components that are part of an *SSO* system; the End User, SP, and IdP. Also, different variants of *SSO* were described, as well as the functionality of *SLO*. Section 2.4 was dedicated to giving a description of IdPs. And *multi-factor authentication*, a way to increase the security during IdP authentication, was explained as well.

In Section 2.5, various Web security threats related to *SSO* were outlined. First, *CSRF* was explained, a technique for forging requests to Web sites where a user has already been authenticated. And then the concept of *Clickjacking* was clarified, which is an attack that can be used to make a user click something without actually being aware of it. Next, *MITM* was explained, as well as *phishing*, which is a significant threat to *SSO* systems.





## 3 OpenID

### 3.1 Goal

The goal of this part is to give an introduction to the OpenID protocol, as it will be used later in the practical part (Section 4). This introduction is aimed at people with no previous knowledge of the protocol, so first of all, a walk-through will be given of how to create and use an OpenID identity.

Following such a basic introduction, the focus will switch to a more technical level, describing various parts of the protocol's specification. As an example, there will be given a detailed explanation of what happens during an OpenID authentication request/response. This includes an overview of the types of messages that are being exchanged during authentication.

There will be given explanations of various extensions to the OpenID protocol that are used during authentication. Also, real-life examples of multi-factor authentication will be given, as well as a part dedicated to phishing protection. And another goal is to answer questions that were outlined in Section 1.4 (*Thesis Outline and Methodology*). Note that the questions are not answered in this part alone; they are addressed in Section 2 and Section 4 as well.

### 3.2 Introduction to OpenID

As part of the application development in this thesis, the utilization of SSO will be implemented using a protocol called *OpenID*. OpenID is a decentralized authentication protocol which was first introduced in 2005,<sup>10</sup> which allows users to use an existing account to sign on to multiple Web sites without the need to create a new password for each site. This is the concept of SSO, as described previously (in Section 2.2). As of today, there have been published two final specifications of OpenID; OpenID Authentication 1.1 and OpenID Authentication 2.0.



A user's OpenID may have various pieces of associated information (e.g. name and e-mail address), and each time a user signs on to a new Web site, this information can be shared. It is up to the user to decide how much of the information to share with a Web site. Normally, the user controls this by being presented by a list of requested information and then tick the fields that she allows. This functionality will be described further in Section 3.5.1 (*Simple Registration Extension*).

---

<sup>10</sup><http://openid.net/get-an-openid/what-is-openid>

The OpenID IdP confirms the identity of a user each time she signs on to a Web site. The IdP is the only party with which the user needs to share a password. This means that no password is transferred between an OpenID-enabled Web application and IdPs; passwords are only given to IdPs. Thus, the user does not have to worry about unscrupulous password management by the Web sites.

**OpenID Prevalence** According to *Janrain*, a company offering OpenID solutions, there are today over 1 billion OpenID-enabled users and more than 9 million OpenID-enabled Web sites (as of December 2009). The number of Web sites accepting OpenID is still growing fast, and one of the reasons why the growth is so strong, is that many of the world's largest Internet companies are adopting (or planning) support for OpenID. As giants like America Online (AOL), Twitter, MySpace, and Yahoo! integrate OpenID support for their enormous user bases, it is obvious that the potential growth increases dramatically. Google also supports OpenID. Note that some large actors have only joined the OpenID foundation without yet implementing the protocol (e.g. PayPal).

When users see that they actually have an existing identity present on the Web that also can be used as an OpenID, the threshold for adopting OpenID functionality becomes much lower. Obviously, the future prevalence of SSO strongly depends on the help from companies holding large user bases.

**The OpenID Foundation** OpenID is not owned by anyone; it is created by an open source community. The protocol is decentralized, and the idea is that anyone who wishes to can use their OpenID at Relying Parties (RPs) or become an OpenID Provider (OP) without paying for it. If someone wants to host their own OP, there are no particular approval procedures to pass, nor any form of registration.

The *OpenID Foundation* (OIDF) was created to assist the open source model of OpenID. Basically, this is done by:

- Providing needed infrastructure
- Promoting and supporting expanded adoption of OpenID

The goal of the foundation is to promote, protect, and nurture the OpenID community and the OpenID technologies.<sup>11</sup> The foundation consists of both private individuals and companies. It is a non-profit organization, and it is

---

<sup>11</sup>OpenID Foundation: <http://openid.net/foundation>

backed by a group of sponsoring members, like Google, Yahoo!, Microsoft, VeriSign, PayPal, and Facebook.

The foundation itself will not interfere with the technical decisions made by the OpenID community; it will rather help to enable and protect what the community creates. The OI DF does not dictate the technical direction of the OpenID protocol, so its development is purely community-driven.

**OpenID URL** Just like a person can use her driver's license as a form of identity, OpenID lets you use a *URL* (Web address) as the identity. An OpenID URL can look something like `username.myopenid.com`, which is pointing to a page that the user controls. Obviously, knowledge of the URL alone is not sufficient in order to be gained access to an OpenID-enabled Web site; the user has to provide further proof of ownership (like passwords, digital certificates, or hardware tokens). There is a large number of different OPs on the Internet. So if a user wants to use more than one OpenID (e.g. one for work and one for private purposes), that is also possible.

When a user wants to sign in at an OpenID-enabled Web site, he types the OpenID URL into the sign-in form, as shown in Figure 9 (the whole process will be described in detail in Section 3.4 and 3.5). Actually, there are



Figure 9: Example of an OpenID URL

no restrictions on how an OpenID URL should be composed [1]. This means that it does not matter if a URL looks like

```
http://john.myopenidprovider.com/
```

or

```
http://myopenidprovider.com/some-long-path/?username=john
```

It is the Identity Provider who decides its composition. However, it is still important that the providers try to follow some guidelines when deciding how the identity URLs should be created:

- It should be *consistent* (i.e. not dynamically changing over time)
- It should be *user-friendly* (i.e. easy to remember). This normally means that it is better to have a short URL. As an example, it seems quite a lot easier to remember `username.myopenid.com` than to remember `username.pip.verisignlabs.com`.

When speaking of user-friendliness, many OPs today use identity URLs that are long and quite hard to remember, e.g. Google's `https://www.google.com/profiles/<username>`. However, it is not always necessary for the End Users to actually remember the whole URL; many OpenID-enabled Web sites have added the possibility for users to click the logo of the company that provides their identity, as shown in Figure 3 and Figure 13. This way, users can simply click on their provider and then type in the username, instead of having to remember the whole Identifier URL.

### 3.3 Basic Elements in OpenID 2.0

The OpenID 1.1 specification and the OpenID 2.0 specification are slightly different in terms of terminology [1]. The definitions in this thesis will match those set for the OpenID 2.0 specification (see [6]). There are basically three main elements in OpenID 2.0:<sup>12</sup>

1. **End User** The *End User* is a real user who is using an OpenID to sign on to one or more Web sites. The credentials of this user are stored with the OP.
2. **Relying Party (RP)** The *RP* in an OpenID system is the Web site where the End User is logging in using her OpenID. Such a site wants proof that a user controls an OpenID URL, so the RP asks the user to provide a URL and then consumes it. This is why RPs also can be called *Consumers*.
3. **OpenID Provider (OP)** The *OP* is the server where the OpenID credentials of users are stored. This is where an OpenID URL points to when an End User attempts to login at an RP. As an example, in the URL `john.myopenid.com`, the OP is `myopenid.com`. When an RP consumes an OpenID URL, messages are exchanged with the OP. Depending on the response returned from the OP as part of the authentication process, the RP will know if the given ID was valid or not. An OP can also be called *OpenID Server*.

---

<sup>12</sup>Note the resemblance with the terms outlined in Section 2.2.1.

While *SP* and *IdP* were used in Section 2 (*Theoretical Background*), the OpenID-specific terms *RP* and *OP* will be used in this part and in Section 4 (*Security Assessment*). In addition to those mentioned above, there are other terms that should be taken note of:

**User-Agent** A *User-Agent* is simply what the End User interacts with when accessing RPs and OPs, i.e. a Web browser. This is a browser which implements the HTTP/1.1 protocol [6].

**Identifier** An *Identifier* is the OpenID URL. The Identifier identifies the digital identity of an End User. This URL must begin with `http` or `https`, or it can be an Extensible Resource Identifier (XRI) [6].<sup>13</sup>

**Endpoint URL** An OpenID *Endpoint URL* is the URL which accepts OpenID authentication protocol messages. This must be an absolute URL beginning with `http` or `https`, and it is obtained by performing discovery on the Identifier supplied by the user. *Discovery* is a process where the RP uses the user-supplied Identifier to look up information that is necessary for initiating authentication requests to the OP [6].

Figure 10 illustrates how the various elements interact with each other. The

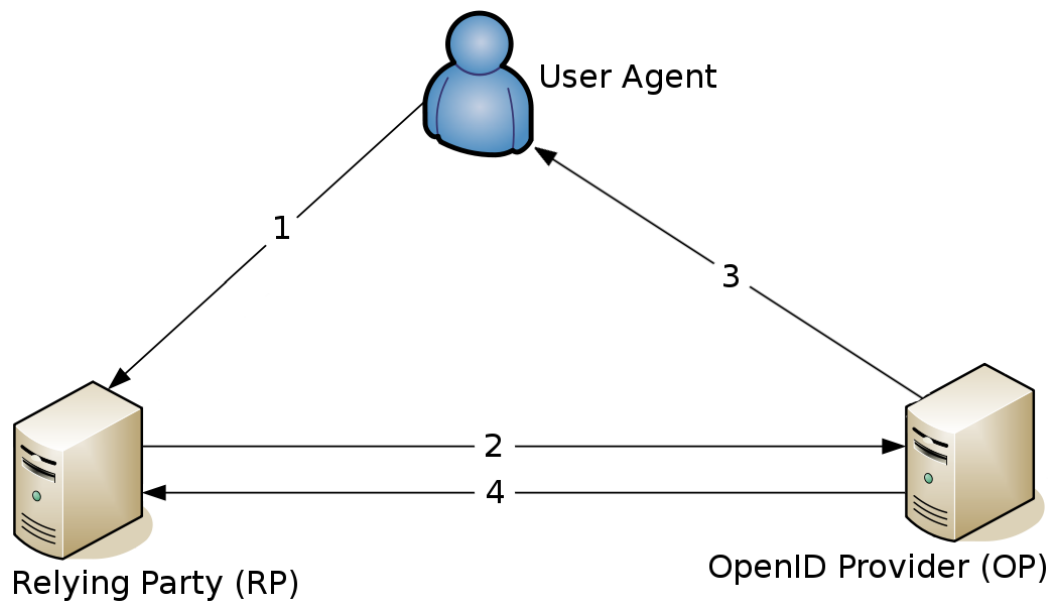


Figure 10: Basic elements of OpenID

following explains each step:

<sup>13</sup>Refer to <http://www.oasis-open.org/committees/xri/faq.php> for more information about XRI.

1. A User-Agent is used to sign on at the RP with an OpenID URL as an Identifier.
2. The RP contacts the OP in order to verify ownership of the Identifier.
3. The OP requests authentication from the User-Agent (unless the User-Agent is already signed on at the OP).
4. If/when the user authenticates, the OP sends requested user data (e.g. name and e-mail) to the RP, and the user is signed on at the RP Web site.

### 3.4 Creating an OpenID

Now that the basic terminology is in place, a description will be given of how to create an OpenID. Before a user on the Internet can enjoy the advantages of SSO, a unique OpenID URL must exist at some OpenID server for RPs to communicate with. This OP will authenticate the user. The first step will be to choose an OP, as will be described in the following.

#### 3.4.1 OpenID Providers (OPs)

As mentioned earlier, there is a large number of OpenID Providers to choose from. As OpenID emerges as a standard for SSO on the Internet, more and more OPs will be created, and it might seem like a daunting task to pick the best one.

The OpenID protocol does not mandate that any particular form of authentication should be supported by OPs. For this reason, there is a large span of different OP implementations to choose from; some might use simple username/password credentials, while others might require smart cards, etc.

The following are some, of many more, examples of OPs that a user on the Internet can choose to register with:

- VeriSign Labs - Personal Identity Portal (PIP)  
<https://pip.verisignlabs.com>
- myOpenID  
<https://www.myopenid.com>
- Yahoo!  
<http://openid.yahoo.com>
- Google  
<https://www.google.com/profiles>

- Certifi.ca  
<https://certifi.ca>
- clavid (Swiss provider)  
<https://www.clavid.com/portal>
- nettid.no (Norwegian provider)  
<https://nettid.no>
- Feide OpenIdP (Norwegian provider)  
<https://openidp.feide.no>

These providers vary among themselves in the way they work. As an example, it is possible to sign on with a *Yahoo!* OpenID using only a username/password combination, while the *Certifi.ca* OP uses SSL certificates instead of passwords. An additional comparative example is *myOpenID*, which supports multi-factor authentication (this will be demonstrated in Section 3.6.3) and *Google*, which does **not**.

It is also possible for anyone to become an OP themselves. This, of course, requires a certain degree of technical skills. The complexity varies, depending on the chosen way of implementation:<sup>14</sup>

1. The simplest approach is to outsource the development to a *third-party*, e.g. Janrain's *Engage* solution.<sup>15</sup> This might suite enterprises which have existing user management systems.
2. A presumably more time-consuming approach is to develop and host an OP solution yourself using OpenID-capable libraries, plugins, or software packages, e.g. *Java OpenID Server* (JOS).<sup>16</sup>
3. The most difficult way of becoming an OP would be to implement support for the OpenID protocol directly, without any usage of libraries or plugins. This is a risky approach, and is not recommended for developers without heavy experience within Web security [1].

For most people, simply looking for a way to obtain their own OpenID, the easiest solution will be to just choose one of the existing OPs on the Internet, rather than setting up their own.

The way the OpenID system works, users are completely free to choose any OP of their own choice. This means that it is entirely up to the user

---

<sup>14</sup><http://openid.net/add-openid/become-a-provider>

<sup>15</sup><http://www.janrain.com/products/engage>

<sup>16</sup><http://code.google.com/p/openid-server/>

to decide how security-aware her OP should be. An Internet novice might not care much how an OP handles her data, while another user might have requirements like encryption, multi-factor authentication, etc. A common factor, however, is the fact that most people would prefer to register with OPs that have an *established reputation* on the Internet. It is reasonable to believe that the majority of users would rather create an OpenID with a large and well-known company (e.g. Yahoo! or VeriSign) than with some company they have never heard about.

**VeriSign Labs PIP** As an example, the following will demonstrate how to create an OpenID with *VeriSign Labs PIP*. VeriSign is most known as a Certification Authority (CA) for digital certificates and has currently an established reputation on the Internet.

The first step of registering with VeriSign's PIP consists in typing in a username, password, and e-mail address, just like any registration form on the Internet (see Figure 11).

Next, after verifying the registered e-mail address, the user can sign in. Once signed in, it is possible to navigate to a page displaying *My Information* (see Figure 12). This page shows a list of information associated with the OpenID account. All of these fields, if containing a value, can easily be shared with OpenID-enabled Web sites (RPs). Note, however, that this information will *not* automatically be shared with sites a user signs on to; the user first needs to confirm that it is ok to share certain pieces of information. For this reason, it is no problem to fill out all of the fields under My Information (unless the OP itself is not trustworthy). After all, one of the major advantages of using an OpenID is to *not* having to fill in commonly requested pieces of information over and over again for each new Web site registration. Note that these fields match those listed in the OpenID *Simple Registration Extension* (SREG). This extension will be described further in Section 3.5.1.

## 3.5 Using OpenID

With an OpenID at hand, let us see how it can be utilized. There are many OpenID-enabled Web sites out there, one of which is `sourceforge.net`, the world's largest open source software development Web site. For the sake of demonstration, the user created earlier has been signed out from the VeriSign PIP OP before proceeding.

As shown in Figure 13, the OpenID URL `jogrimst.pip.verisignlabs.com` is typed into the login page of SourceForge.net. When clicking "Log in",



VeriSign Labs  
**Personal Identity Portal** Beta

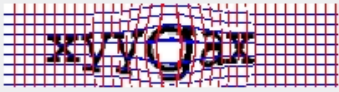
### Create Your Account

Use PIP to protect your information, share it with sites you trust anywhere you see OpenID, and safely sign in with only one username across the Internet; all protected by VeriSign.

\* Required Information

Account Information		
* Username	<input style="width: 90%;" type="text" value="jogrimst"/>	The username that you enter will be used to create your OpenID.
* Password	<input style="width: 90%;" type="password" value="●●●●●●"/>	Minimum of 6 characters and must contain a number and a letter.
* Confirm Password	<input style="width: 90%;" type="password" value="●●●●●●"/>	
* Email	<input style="width: 90%;" type="text" value="jogrimst@stud.ntnu.no"/>	Only used for account verification, recovery and infrequent service update notices.
<input checked="" type="checkbox"/> Keep me informed of changes to PIP -please send me email updates!		


  

Verification Code		
* Verification Code	<input style="width: 90%;" type="text" value="xyygax"/>	For security purposes, type the code seen in the image below.
		

By creating my account I have read and agree to the [Terms of Service](#) and [Privacy Policy](#)

Create Account

Figure 11: VeriSign Labs PIP - Create Account



## My Information

The table below lists the information you entered when an OpenID Web site requested additional data from you. Use the links in the action column to manage the information associated with your account.

Add a Field

Show Tag: All

My Information			
Field Name ▼	Value	Tags ?	Action
Full Name	Jo Grimstad		<a href="#">Edit</a>
Nickname	jogrimst		<a href="#">Edit</a>
Email Address	jogrimst@stud.ntnu.no		<a href="#">Edit</a>
Language	Norwegian		<a href="#">Edit</a>
Gender	Male		<a href="#">Edit</a>
Date of Birth	1984-06-16		<a href="#">Edit</a>
Country	Norway		<a href="#">Edit</a>
Postal Code	7012		<a href="#">Edit</a>
Time Zone	Europe/Oslo		<a href="#">Edit</a>

Figure 12: VeriSign Labs PIP - My Information

### Log in with OpenID

---

Please click your account provider:

Enter your OpenID.

Remember Me ?

Figure 13: <https://sourceforge.net/account/login.php>

the user is eventually redirected to `http://pip.verisignlabs.com/server`, and the following parameters are passed along in an HTTP GET request:

```
http://pip.verisignlabs.com/server
?openid.ns=http://specs.openid.net/auth/2.0
&openid.mode=checkid_setup
&openid.identity=http://jogrimst.pip.verisignlabs.com/
&openid.claimed_id=http://jogrimst.pip.verisignlabs.com/
&openid.assoc_handle=a8843e90-4d6c-11df-a273-d702551e809e
&openid.return_to=https://sourceforge.net/account/openid_verify.php
&openid.realm=https://sourceforge.net
&openid.ns.sreg=http://openid.net/extensions/sreg/1.1
&openid.sreg.optional=nickname,email,fullname,country,language,timezone
&openid.sreg.policy_url=http://p.sf.net/sourceforge/privacy
```

In this case, `sourceforge.net` is the RP, and `pip.verisignlabs.com` is the OP. By sending the request above, the goal of the RP is to obtain an assertion from the OP, which can be a *negative assertion* or a *positive assertion*. When a user authorized with the OP wishes to complete the authentication, a *positive assertion* should be returned to the RP [6].

Next, as long as the given OpenID (the value of the `openid.identity` parameter) is found in the system of VeriSign Labs PIP, a sign in form appears, requesting a username/password combination (see Figure 14). After

VeriSign Labs  
**Personal Identity Portal** Beta

### Sign In

Enter your username and password, then click the **Sign In** button below.

Sign In	
Username	<input type="text" value="jogrimst"/>
Password	<input type="password" value="●●●●●●●●"/> <a href="#">Forgot my sign in information</a>



  
Sign In

Figure 14: VeriSign Labs PIP - Sign In

supplying the required credentials, the user is logged in at the VeriSign PIP account and is presented with a page where it is possible to choose which information to share with the RP (see Figure 15). Also, the user is able to choose for how long time this trust relationship should last. The user is



### Sign In with Your OpenID

The Web site, <https://sourceforge.net> is requesting verification that **jogrimst** is your OpenID.

Complete the following form, select when you want the trust relationship for this site to expire and click **Allow**.

Click **Deny** to deny this request and return to <https://sourceforge.net>.

\* Required Information

**OpenID Information**

Use the **My Information** section on the right to help complete the form

Full Name	<input type="text" value="Jo Grimstad"/>
Nickname	<input type="text" value="jogrimst"/>
Email Address	<input type="text" value="jogrimst@stud.ntnu.no"/>
Country	<input type="text" value="Norway"/>
Language	<input type="text" value="Norwegian"/>
Time Zone	<input type="text" value="Europe/Oslo"/>

My Information

Click to copy the information to the associated field on the left.

- Full Name: Jo Grimstad
- Nickname: jogrimst
- Email Address: jogrimst@stud.ntnu.no
- Postal Code: 7012

Trusted Site Expiration

Expiration	<input checked="" type="radio"/> Never Expire <input type="radio"/> Expire on: <input type="text" value="Apr"/> <input type="text" value="21"/> <input type="text" value="2010"/> <input type="radio"/> Expire After Signing In
------------	---

Figure 15: VeriSign PIP requesting verification

presented with 3 options for expiration:

1. Never expire
2. Expire on a given future date

### 3. Expire immediately after signing in to the RP

If the user chooses “Never expire”, this particular verification step will be skipped the next time the user signs in to the Web site `sourceforge.net`.

Once the user has agreed to the RP’s request, the OP returns the user to the URL that was specified as part of the request:

```
openid.return_to=https://sourceforge.net/account/openid_verify.php
```

This is the location where the OP will indicate the status of the RP’s request for user authentication. In this example, the VeriSign PIP OP passed along the following data to the SourceForge RP when the user was sent back to the `openid.return_to` URL:

```
https://sourceforge.net/account/openid_verify.php
?openid.sreg.fullname=Jo Grimstad
&openid.sreg.timezone=Europe/Oslo
&openid.sreg.language=nor
&openid.assoc_handle=a8843e90-4d6c-11df-a273-d702551e809e
&openid.response_nonce=2010-04-21T17:08:49ZjN+sHA==
&openid.sreg.email=jogrimst@stud.ntnu.no
&openid.sreg.country=NO
&openid.sreg.nickname=jogrimst
&openid.ns=http://specs.openid.net/auth/2.0
&openid.mode=id_res
&openid.op_endpoint=http://pip.verisignlabs.com/server
&openid.pape.auth_policies=http://schemas.openid.net/pape/policies/[...]
&openid.claimed_id=http://jogrimst.pip.verisignlabs.com/
&openid.sig=QA1pZ+Y9GNmr+mrwvO2wlsGTJ5eNepHEQFIibWGs1e0=
&openid.identity=http://jogrimst.pip.verisignlabs.com/
&openid.ns.pape=http://specs.openid.net/extensions/pape/1.0
&openid.pape.auth_time=2010-04-21T14:02:47Z
&openid.signed=assoc_handle,identity,response_nonce,return_to,claimed_id,[...]
&openid.ns.sreg=http://openid.net/extensions/sreg/1.1
&openid.return_to=https://sourceforge.net/account/openid_verify.php
```

Through this positive assertion, the RP knows that the user was successfully authenticated. And since the RP has been supplied with personal info like name, e-mail, nickname, etc, it is possible to automatically create an account for the user.

And that is also exactly what has happened in this example. As the user clicked “Allow” at the VeriSign PIP verification page, the Consumer Web site (SourceForge) performed some operations behind the scenes:

1. Created a new user account (assigned user id, nickname, etc).
2. Logged in the new user.

Figure 16 shows the Web page that is presented to the user after authenticating with the Identity Provider. Note that the RP has picked information

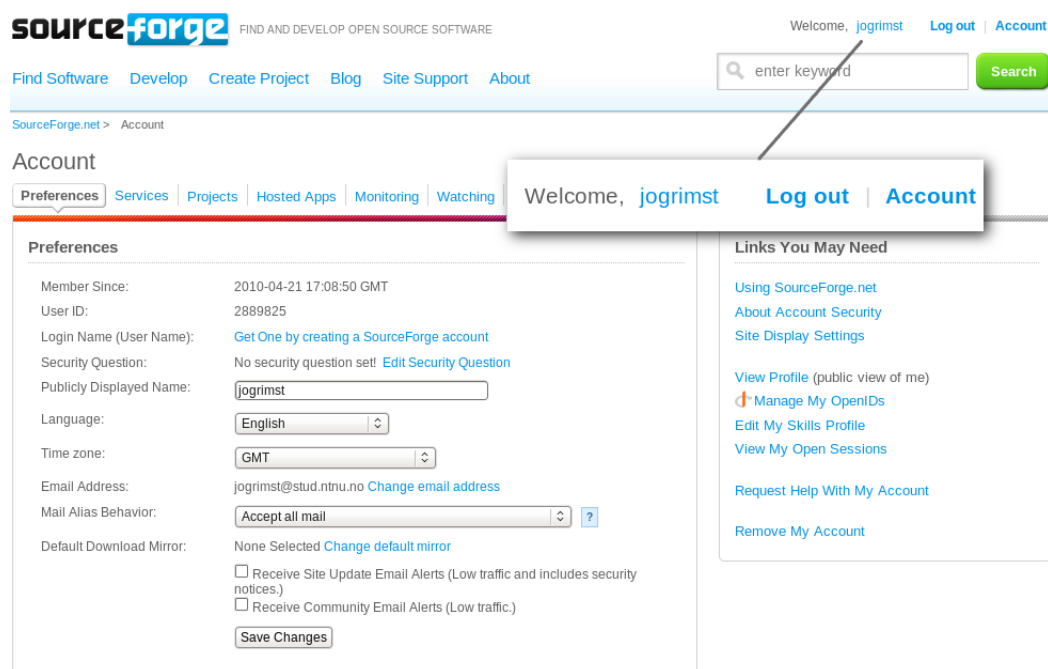


Figure 16: New account created using an OpenID

like nickname and e-mail address returned from the OP in order to create a new user. As the figure shows, the user is immediately given the possibility to modify account details, as well as logging out of the initiated user session.

**Using OpenID when Already Signed On** Now, consider the situation in which the user has signed on to the OP (Verisign PIP) and wishes to access another Consumer Web site. As an example, assume that the user would like to login with another OpenID-enabled Web application located at <http://demand.openid.net>.<sup>17</sup> First, the user types in the OpenID URL, as shown in Figure 17. The next thing that happens, is that the user is taken directly to the VeriSign PIP verification page. What is important to note in this case, is that the user did **not** need to sign on with the OP (i.e. no

<sup>17</sup>This is a Web page where people (with OpenIDs) can cast their vote of which Web sites they want to add OpenID as a login option.

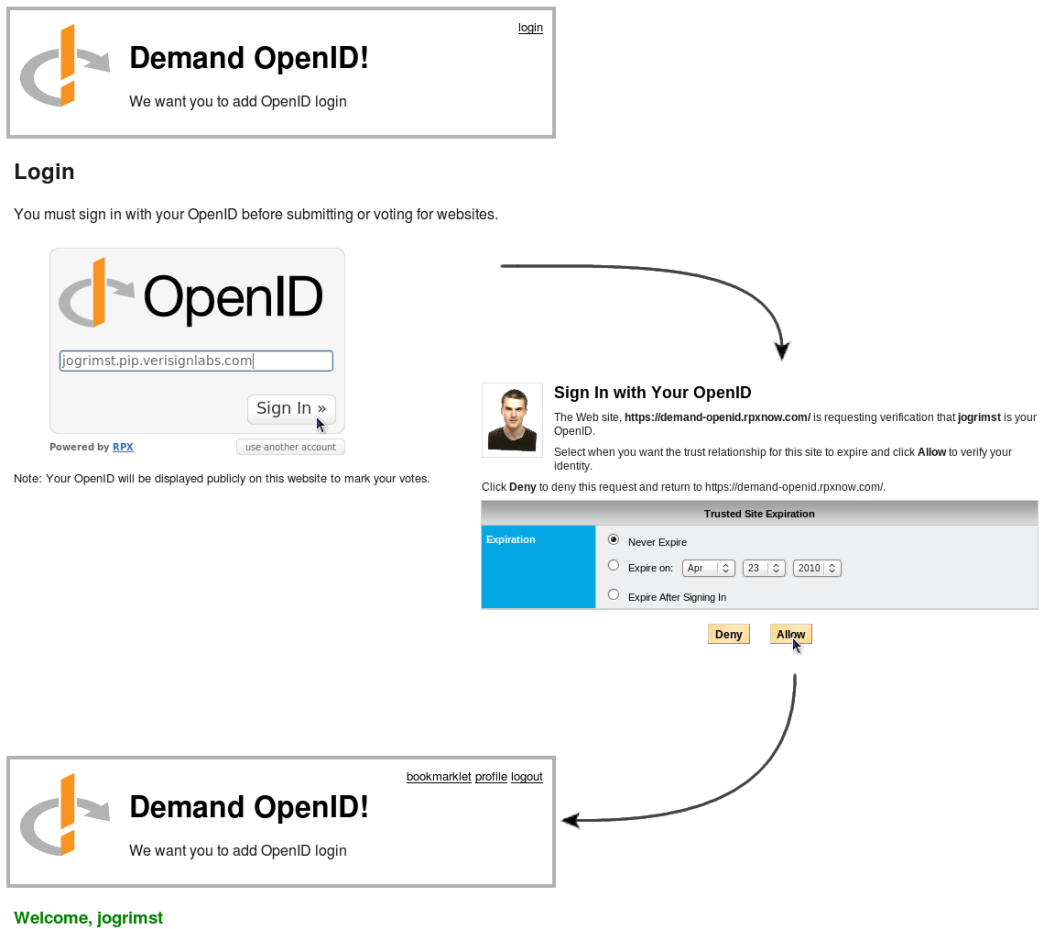


Figure 17: Signing into a *second* OpenID-enabled Web site

username/password combination was requested) once more. Since a sign-on had been performed with a RP earlier (SourceForge), the user did not have to repeat this step with the second RP. This is an important part of the OpenID system, and gives a clear understanding of why it is called *Single Sign-On*. It is not necessary to provide the OP with login credentials once more when already signed on.

The steps above have basically shown how easy it is to create and use an OpenID. Obviously, it simplifies the life of End Users, being both a time-saver and a user-friendly solution. In the following, we will dig a little deeper into the technicalities; first of all, there will be given explanations of some extensions to the OpenID protocol that are used during authentication.

### 3.5.1 Simple Registration Extension (SREG)

The OpenID protocol offers several extensions. Among these is one called *Simple Registration Extension* (SREG). The purpose of SREG is to allow for light-weight profile exchange [7]. When Internet users sign up with various Web sites, they are often asked to provide more information about themselves than just a username and a password. Information like full name, e-mail address, and country are typically requested as well.

The SREG extension has defined a collection of 9 such commonly requested pieces of information (see Table 1). When an RP performs an authentication request towards an OP on behalf of a user, it must pass along either `openid.sreg.required` or `openid.sreg.optional` parameter values. These fields both contain a comma-separated list of field names matching those in Table 1. As an example, consider the request parameters described

Parameter Name	Information
<code>openid.sreg.nickname</code>	Nickname
<code>openid.sreg.email</code>	E-mail address
<code>openid.sreg.fullname</code>	Full name
<code>openid.sreg.dob</code>	Date of birth (“YYYY-MM-DD”)
<code>openid.sreg.gender</code>	Gender (“M”/“F”)
<code>openid.sreg.postcode</code>	Postcode
<code>openid.sreg.country</code>	Country
<code>openid.sreg.language</code>	Language
<code>openid.sreg.timezone</code>	Time zone

Table 1: OpenID SREG fields

in Section 3.5, where the SourceForge RP sent an authentication request



to the VeriSign PIP OP. There, the SREG parameter defined the following fields:

```
openid.sreg.optional=nickname,email,fullname,country,language,timezone
```

In this particular case, the user was asked by the RP to provide nickname, e-mail address, full name, country, language, and timezone, as was shown in Figure 15. None of the fields were compulsory to fill out, so if the user would have left them all blank, it would still be possible to proceed. However, the authentication request could also consist of SREG attributes looking something like this:

```
openid.sreg.required=nickname,email
openid.sreg.optional=fullname,country,language,timezone
```

In such a case the user would have been presented with an asterisk (‘\*’) next to the nickname and e-mail fields, symbolizing that these are compulsory to fill. If the user is not willing to share the required pieces of information, it is impossible to proceed the authentication procedure. For this reason, RPs should only mark user profile information as required if it is absolutely necessary for account creation.

In addition to the request parameters defining required and optional data fields, there is an SREG parameter called `openid.sreg.policy_url`. During an OpenID verification, the RP can pass along a URL pointing to a privacy policy document. Here, the RP would inform about how profile data retrieved from the OP will be used. In the example above, the SourceForge RP sent the following URL value:

```
openid.sreg.policy_url=http://p.sf.net/sourceforge/privacy
```

The idea is that the OP will display a link to this policy document at the same page as where the OpenID verification happens. This way, the user gains easy access to the policies of the Web site she is signing up with, without having to browse back to the RP and search for it manually. However, even though an RP includes the `openid.sreg.policy_url` parameter, it is still up to the OP whether to display the URL or not; it is not a mandatory part of the OpenID protocol. As an example, the VeriSign PIP OP does not display policy URLs, while others (like myOpenID) do.

In an authentication *response* coming back from an OP, the requested SREG fields defined in `openid.sreg.required` and/or `openid.sreg.optional` are returned. The comma-separated fields that are listed in the request parameter(s) are returned as separate values. For example, consider once more the request that was made by the SourceForge RP above:

```
openid.sreg.optional=nickname,email,fullname,country,language,timezone
```

This resulted in a response containing, amongst others, the following values:

```
openid.sreg.nickname=jogrimst
openid.sreg.email=jogrimst@stud.ntnu.no
openid.sreg.fullname=Jo Grimstad
etc...
```

In addition to these values, the response also contains two fields called `openid.signed` and `openid.sig`. For a description of these parameters, please take a look at Appendix A.1.

## 3.6 Multi-Factor Authentication

Several OpenID Providers support authentication mechanisms that are considered stronger than a basic username/password combination. There are many different methods to choose from, some of which will be presented in the following. As mentioned in Section 2.4.1, *multi-factor* authentication consists of combining two or more authentication factors. The OPs that will be used as examples utilize a password as one of the factors when authenticating their users. To achieve multiple factors, the following authentication methods are examples of what can be used (in addition to a password):

### 3.6.1 One-Time Password (OTP)

A *One-Time Password* (OTP) is often used in addition to a main password in order to keep malicious users out of the system even if they have been able to obtain a user's password. An OTP functions as a second barrier, which reduces the severity of a compromised main password. Obviously, it is important not to keep the main password anywhere near the device that is being used for OTP generation. Then the point of using OTPs would be pretty much useless.

As the name implies, OTPs have to change in some way, each time a user wants to obtain one. There are a lot of different ways to generate OTPs. Still, the techniques can be roughly categorized depending on the manual involvement of the user. Basically, OTPs can be generated in one of two ways:

1. In specified *time intervals* (e.g. a new one every minute)
2. By a user-initiated action (e.g. by the press of a button on some device), i.e. *not* in specified time intervals.

The following shows some examples of both of these techniques:

**OTP Generation Using Mobile Phone** The VeriSign PIP OP, which has been used as an example earlier, also offers a way to obtain OTPs using a mobile phone. VeriSign has called it *VeriSign Identity Protection* (VIP). And this is something that users of VeriSign PIP can choose to activate in order to further protect their account. Figure 18 shows how the sign-on process works with PIP activated. As before, the user is first asked to provide a username and a password. At the next page, the user is prompted for the OTP; in this case a six-digit security code. To obtain this OTP, the user can install an application on her mobile phone which generates OTPs for her corresponding VeriSign PIP account.<sup>18</sup> Each time the user signs on, she has to start the application and type the OTP into the input field of the VeriSign Web site. A new OTP is generated automatically every 30 seconds.

### Sign In

Enter your username and password, then click the **Sign In** button below.

Sign In



### Sign In with Your VeriSign Identity Protection (VIP) Credential

Your account is protected by VeriSign Identity Protection (VIP). Enter a new six-digit security code from your VIP credential and click **Sign In**.

[Click here if you don't have your VIP credential.](#)

Sign In

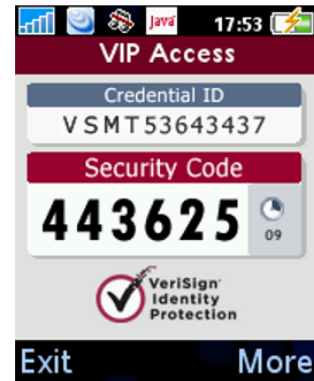


Figure 18: VeriSign VIP on Sony Ericsson mobile phone

There are many ways a mobile phone can be used to obtain OTPs. As an example, another way to do it is sending OTPs as text messages to the mobile phone belonging to the user currently being authenticated.

<sup>18</sup>The application can be downloaded from <https://vipmobile.verisign.com/home.v>

**OTP Generation Using Hardware Token** Another way to generate OTPs is through the usage of *hardware tokens*, like USB devices. These are normally small electronic devices that can be carried with your key chain. An example of such a device is the *YubiKey*,<sup>19</sup> a USB-dongle manufactured in Sweden. The YubiKey is illustrated in Figure 19. YubiKey is a *unique* USB-




Figure 19: YubiKey USB-key connected to key chain

key which can be used for strong authentication, i.e. it can generate OTPs. The YubiKey can be used to generate OTPs that consist of 44 characters, of which the first 12 characters are fixed. Each Yubikey is identified by the first 12 characters of the OTPs it generates. Every YubiKey that is produced gets its own, unique, 12 characters long identifier, which is prepended to a cryptographically secured OTP consisting of 32 characters. This means that it is impossible for two different YubiKeys to ever generate identical OTPs.

Usage of the YubiKey is not widely spread amongst OpenID Providers as of today, but this might change in the future. Some of the OPs that have implemented support for YubiKey is `nettid.no` and `clavid.com`. In the following, there will be given a brief introduction to how the YubiKey can be used when signing on using the `clavid.com` server as OP. First, the user behaves as in the examples given earlier, e.g. typing the OpenID URL into some Web site in order to sign on. Let us once more use the OpenID-enabled Web site `sourceforge.net` as an example. Figure 20 shows the Web page at `clavid.com` that is presented to the user as part of the sign-on process. In this case, the sign-on form consists of an additional field, namely one reserved for a YubiKey OTP. At this stage, the user types in the password. Next, he has to utilize the YubiKey to generate an OTP. This is not a complicated procedure; the user simply inserts the YubiKey into an available USB-port and presses the button.



<sup>19</sup><http://www.yubico.com/products/yubikey>



The screenshot shows the login interface for clavid.com. At the top, the clavid logo is displayed with the tagline "one key, all access". Navigation links for "homepage" and "de" are visible. A green header bar contains the text "Login" and "clavid - one key, all access". The main content area prompts the user to sign in to authenticate to a specific URL: [https://sourceforge.net/account/openid\\_verify.php](https://sourceforge.net/account/openid_verify.php). The login form includes fields for Username (jogrimst.clavid.com), Authentication type (YubiKey + Password), Password (masked with dots), and YubiKey (masked with dots and a YubiKey icon). A "Login" button is present, along with links for "I forgot my login information" and "Back". The Yubico logo and tagline "trust the net" are at the bottom right. A copyright notice "copyright © 2007-2010 clavid ag, all rights reserved." is at the bottom left.

Figure 20: Using the YubiKey at clavid.com

In fact, the YubiKey is configured to identify itself to computers as a *USB keyboard*.<sup>20</sup> When the user presses the button, an OTP is generated and automatically inserted into the input field currently in focus. At this point, a string of 44 characters is typed into the YubiKey input field, just as if it was done manually using a regular keyboard, completed with the **Enter** button.

Next, when the user has been authenticated using the YubiKey, the sign-on process continues as normal, through information sharing with the `sourceforge.net` Web site. As demonstrated, obtaining multi-factor authentication is not necessarily a procedure that has to be complex and exhausting. The YubiKey uses techniques that makes the OTP generation fast and easy. Many users (my self included) might find usage of the YubiKey easier and more elegant than other OTP solutions, like VeriSign's VIP application for mobile phones described previously.

A large difference between the YubiKey and many other hardware tokens for OTP generation is the fact that the user is relieved from the burden of manually reading and typing in the OTP. The YubiKey comes with many other advantages:

- No client software to install (it works on all computers, platforms, and browsers).
- Portable (small and easy to carry), weighs only 2.5 grams.
- Battery-free (it obtains the needed electricity from the USB-port).
- Low-cost (\$25 per piece, or \$15 if bought in bulk).
- Supports both the OpenID and the Security Assertion Markup Language (SAML) standard.

There are of course *disadvantages* with such a solution as well:

- A computer with a USB-port is needed. Many people would probably miss having a simple way to use the UbiKey with mobile phones lacking USB-port.
- The YubiKey has to be ordered and mailed to the user. This process is slow, in comparison to the situation in which users can utilize something they already possess for immediate OTP generation, e.g. their mobile phone.

---

<sup>20</sup><http://www.yubico.com/products/description>

- If a user already owns hardware tokens from several other providers, she might eventually end up carrying too many tokens at once, hence being forced to remove some of them.

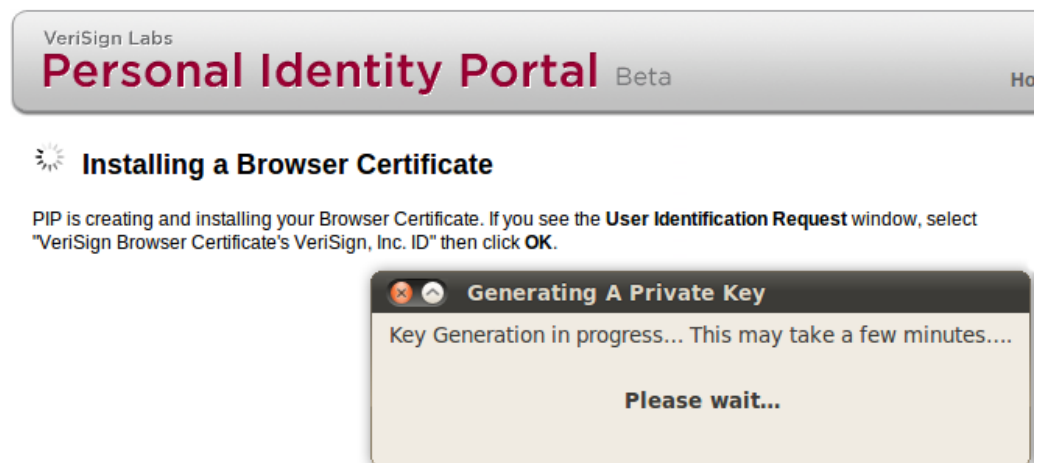
Do note that the YubiKey is not restricted to be used in conjunction with OpenID only; even though it has been used in OpenID examples here, it can be used with many other systems as well, e.g. *TrueCrypt*<sup>21</sup> and *LastPass*.<sup>22</sup> For technical details of how the YubiKey works, see Appendix C.

### 3.6.2 Digital Certificates

An alternative to OTPs as a way to achieve multi-factor authentication is the usage of *digital certificates*. Instead of using OTPs, a user can present a digital certificate installed in the browser. This requires that the user is logged into a computer with the certificate installed; each time the user changes the computer (or browser), the certificate needs to be reinstalled.

As an example, the VeriSign OP offers its users to use a browser certificate as one of the additional account protection mechanisms. As with OTPs, a malicious user would not be able to gain unauthorized access to a user's account due to a compromised username/password combination unless he is also in possession of the digital certificate associated with the account.

VeriSign PIP is one of the OPs that offer strong authentication using digital certificates. The user simply logs into her account and enables the feature. As part of the process, a certificate is generated on-the-fly and installed in the browser that is currently being used. Having enabled this



<sup>21</sup><http://www.truecrypt.org>

<sup>22</sup><http://lastpass.com>

protection, the subsequent logins performed by the user will include a check to make sure that the correct certificate is installed in the browser. Figure 21 shows how the login process looks like at the VeriSign PIP OP when browser certificate authentication has been activated. If the user wishes to sign in with a different computer or with a browser that does not have the certificate installed, VeriSign PIP offers a temporary way of signing in. In this case a PIN code can be sent to the e-mail address or mobile phone associated with the account. Once logged in, the user can install the certificate.

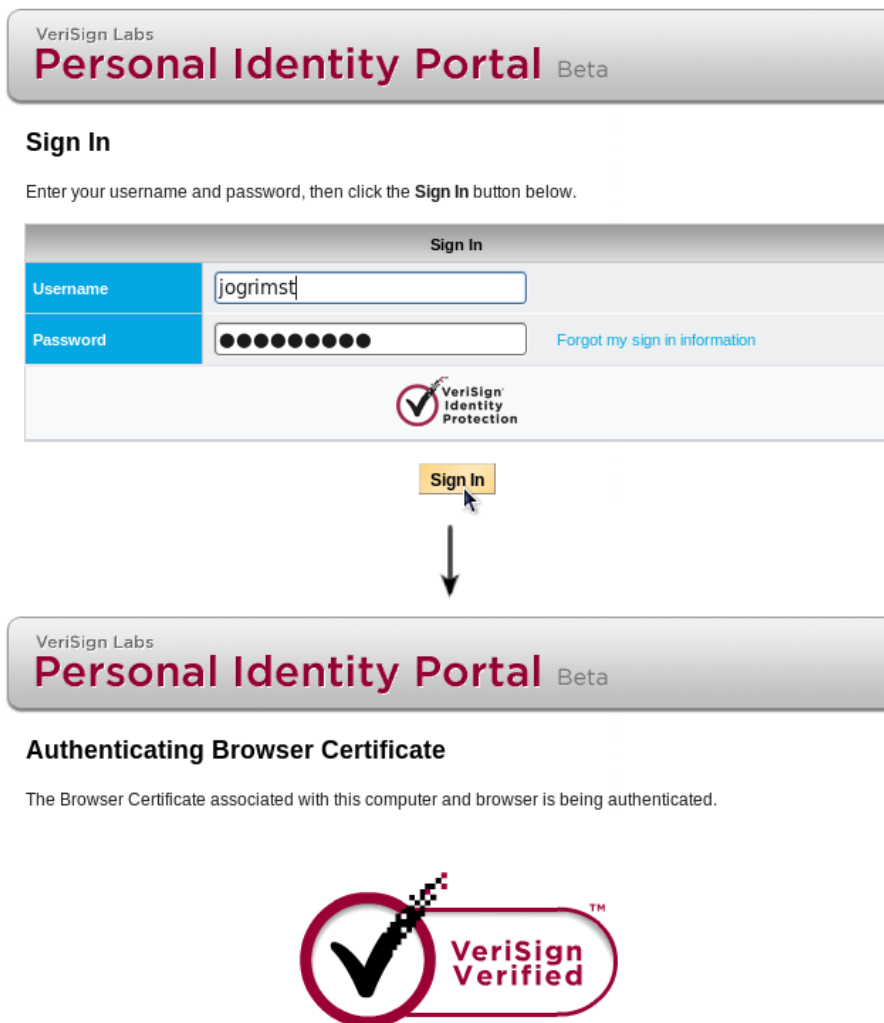


Figure 21: Multi-factor authentication using a digital certificate



### 3.6.3 Other Methods

There are also other methods for multi-factor authentication; e.g. something called *CallVerifID*.<sup>23</sup> This is a security measure offered to users of the myOpenID OP. When signing into myOpenID, CallVerifID can be used to make the user instantly receive a call. Then, to authenticate, the user simply answers the call, and presses #. To enable this type of two-factor authentication, the user first needs to sign on at her myOpenID account, go to “Authentication Settings” (see Figure 22), and then add the phone number

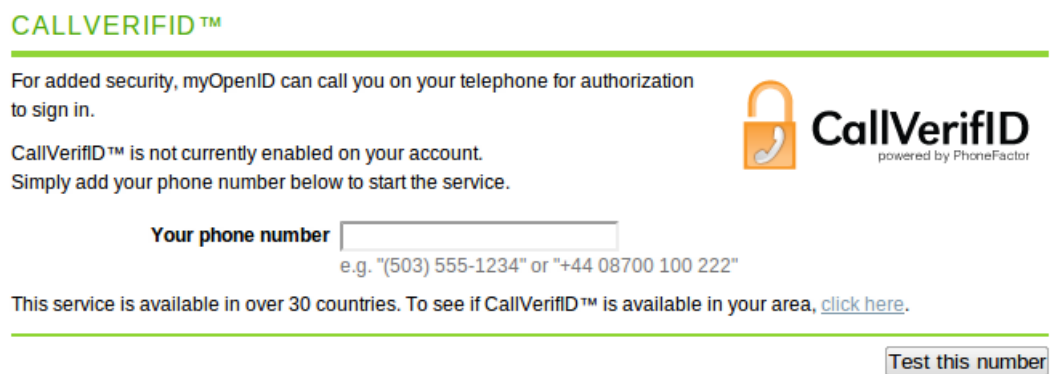


Figure 22: Multi-factor authentication with CallVerifID

that shall be used. There are, however, only a handful of countries where this solution is available to use with mobile phones; only Canada, China, Hong Kong, Puerto Rico, Singapore, USA, and the Vatican City.<sup>24</sup> It should be mentioned that CallVerifID is available through land line telephones in most countries (Norway included), but as less and less people continue using land line phones, this alternative is a relatively impractical one.

## 3.7 Provider Authentication Policy Extension (PAPE)

The Provider Authentication Policy Extension (PAPE) is an *optional* extension to the OpenID authentication protocol, but the OpenID community strongly recommends it [8]. The purpose of the PAPE extension is to allow RPs to request that certain authentication policies should be applied by the OP during authentication of an End User [8]. Also, the extension allows OPs to inform RPs which policies that were used during a certain authentication procedure. For example, the RP can request that the End User authenticates using a phishing-resistant method or a multi-factor authentication method.

<sup>23</sup>CallVerifID: [https://www.myopenid.com/about\\_callverifid](https://www.myopenid.com/about_callverifid)

<sup>24</sup>[https://www.myopenid.com/callverifid\\_availability](https://www.myopenid.com/callverifid_availability)

First of all, an *authentication policy* can be defined as the following [8]:

“An Authentication Policy is a plain-text description of requirements that dictate which Authentication Methods can be used by an End User when authenticating to their OpenID Provider. An Authentication Policy is defined by a URI which must be previously agreed upon by one or more OPs and RPs.”

A policy description can be added by OPs to an End User’s eXtensible Resource Descriptor Sequence (XRDS) document. This, however, is an optional part of the extension. These are the steps that follow:

- When sending an authentication request, the RP includes parameters that describe its preferences regarding the authentication policy.
- The OP processes the received request, and prompts the End User to fulfill the requested policies during the authentication process.
- In the response returned to the RP, the OP includes PAPE information related to the End User’s authentication.
- The RP processes the OP’s response, and determines if the End User should be allowed to log in or not.

Because the OpenID protocol allows anyone to create OPs, it is up to the RPs to decide whether to trust the returned policy claims or not. I.e., the RP itself is responsible for deciding which OPs are to be considered trustworthy.

### 3.7.1 Defined Authentication Policies

As mentioned above, authentication policies describe how End Users may authenticate to an OP. The following are defined policies (and corresponding policy identifiers) in the OpenID protocol [8]:

- *Phishing-resistant* authentication  
<http://schemas.openid.net/pape/policies/2007/06/phishing-resistant>  
An authentication mechanism where the End User does not provide credentials to a party that might possibly be under the control of the RP.
- *Multi-factor* authentication  
<http://schemas.openid.net/pape/policies/2007/06/multi-factor>  
An authentication mechanism where the End User provides more than one authentication factor (see Section 2.4.1).

- *Physical multi-factor* authentication  
<http://schemas.openid.net/pape/policies/2007/06/multi-factor-physical>  
 An authentication mechanism where the End User provides more than one authentication factor (see Section 2.4.1), and where at least one of these factors is a physical factor.

Additional policies can be specified; the ones mentioned are just designed to be a starting point. As an example, CallVerifID (mentioned in Section 3.6.3) can be added as authentication policy (by defining the policy identifier <http://janrain.com/pape/callverifid.html>).

If an RP wants to find out which authentication policies a particular OP supports, it is possible to do so via the use of Yadis<sup>25</sup> within OpenID. As an example, an OP supporting the phishing-resistant authentication policy, would have an XRDS document looking like this:

Listing 2: XRDS document example

---

```

1 <xrd>
2   <Service>
3     <Type>http://specs.openid.net/auth/2.0/signon</Type>
4     <Type>
5       http://schemas.openid.net/pape/policies/2007/06/phishing-resistant
6     </Type>
7     <URI>https://example.com/server</URI>
8   </Service>
9 </xrd>
```

---

As we see on line 5, the policy identifier for phishing-resistance has been added as the value of a `Type` element. For a detailed real-life example of an XRDS document describing the supported authentication policies at the VeriSign PIP OP, see Appendix B.1.

If an RP wants to make an OP aware that the PAPE extension is used, the following parameter must be included in an authentication request.

```
openid.ns.pape=http://specs.openid.net/extensions/pape/1.0
```

If an RP wants the user to go through with certain authentication policies during authentication at the OP, the RP needs to include this information as part of the request, here with `phishing-resistant` and `multi-factor` used as example:

```
openid.pape.preferred_auth_policies=
  http://schemas.openid.net/pape/policies/2007/06/phishing-resistant
```

---

<sup>25</sup> *Yadis* is an XML-based simple protocol which is able to discover services at a particular URL [1].

`http://schemas.openid.net/pape/policies/2007/06/multi-factor`

Another (optional) PAPE parameter is `openid.pape.max_auth_age`. It makes it possible for an RP to require an End User to re-authenticate with the OP if she has not actively authenticated within the specified number of seconds. For example, including the following in a request would require an End User to authenticate to the OP at least every half hour (using the requested authentication policies):

```
openid.pape.max_auth_age=1800
```

When an OP returns an authentication response to the RP, it includes the `openid.pape.preferred_auth_policies` parameter, and the value of this parameter informs which authentication policies the OP satisfied when authenticating the End User. Also, the OP can return a parameter named `openid.pape.auth_time`. This is the most recent time when the End User authenticated to the OP using a way conforming to the authentication policies requested by the RP. The authentication response example provided in Section 3.5 contained this parameter, with the following value:

```
openid.pape.auth_time=2010-04-21T14:02:47Z
```

Optionally, a custom assurance level can be used as well, i.e. assurance levels defined by country-specific or industry-specific standards bodies. As an example, a parameter named `openid.pape.auth_level.ns.nist` indicates usage of a set of assurance levels defined by the *National Institute of Standards and Technology* (NIST):<sup>26</sup>

Assurance level	Factor count
1	1
2	1
3	2
4	2

Table 2: NIST authentication mechanism levels

The right column specifies the minimum number of authentication factors required at each level. So, when an OP has satisfied a certain level, it can return its value to the RP, e.g:

```
openid.pape.auth_level.nist=2
```

<sup>26</sup>NIST Electronic Authentication Guideline: [http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1\\_0\\_2.pdf](http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1_0_2.pdf)

Note that this is just a simple string value in the range 1-4, and there is no problem at all for a malicious OP to simply change this value to a higher one even though the level has not been accomplished. Once again; these values rely on the RP's *trust* in the OP and in the parameter values it returns. From a security perspective, PAPE is a very interesting extension to the OpenID protocol. However, it relies on the RPs to decide which OPs to trust, so there is a need for some kind of public whitelisting of approved OPs. Section 3.9 will discuss this possibility.

As stated previously, the PAPE extension contains a defined policy extension named *phishing-resistant*. So obviously, phishing is one of the attacks that can be mitigated using the extension. When an RP requires a phishing-resistant authentication policy to be followed, an OP could authenticate End Users by using one of the following methods [8]:

- PIN and digital certificate via HTTPS.
- PIN and hardware cryptography token via HTTPS.
- Information card<sup>27</sup> via HTTPS.

For a complete list of the possible request and response parameters in the PAPE extension, please refer to the specification [8].

### 3.8 Phishing Protection

As mentioned in Section 3.7, the PAPE extension can be used for anti-phishing (and other purposes) [1]. To reduce the risk of phishing, there are several other protection mechanisms available as well. First, consider the main phishing scenario threatening OpenID:

1. An End User visits a malicious RP.
2. The RP forwards the End User to a bogus OP, where the End User is asked to provide her credentials.
3. The malicious OP (controlled by the owner of the malicious RP) obtains the End User's login credentials for her OpenID account.

In this case, an anti-phishing solution is for the OP to require that the user authenticates with it **before** authenticating with the RP. OPs that mandate this solution do not allow users to authenticate if they have been redirected

---

<sup>27</sup>A "card" used for login [1]. As an example, the VeriSign PIP OP offers creation and usage of information cards.

from another Web site, i.e. a possibly malicious RP. This protection mechanism requires more manual work by the user, but it is effective against phishing. WordPress<sup>28</sup> is an example of an OP which mandates users to pre-authenticate.

Another way to reduce the risk of phishing is to use a *sign-on seal*. This method is commonly used at OPs, e.g. at myOpenID, Yahoo! ID, and nettId.no, to mention a few. For instance, at the Yahoo! OP, the sign-on seal is either a secret message or a photo that will only be displayed on the user's computer.<sup>29</sup> An example is shown in Figure 23. When a user has

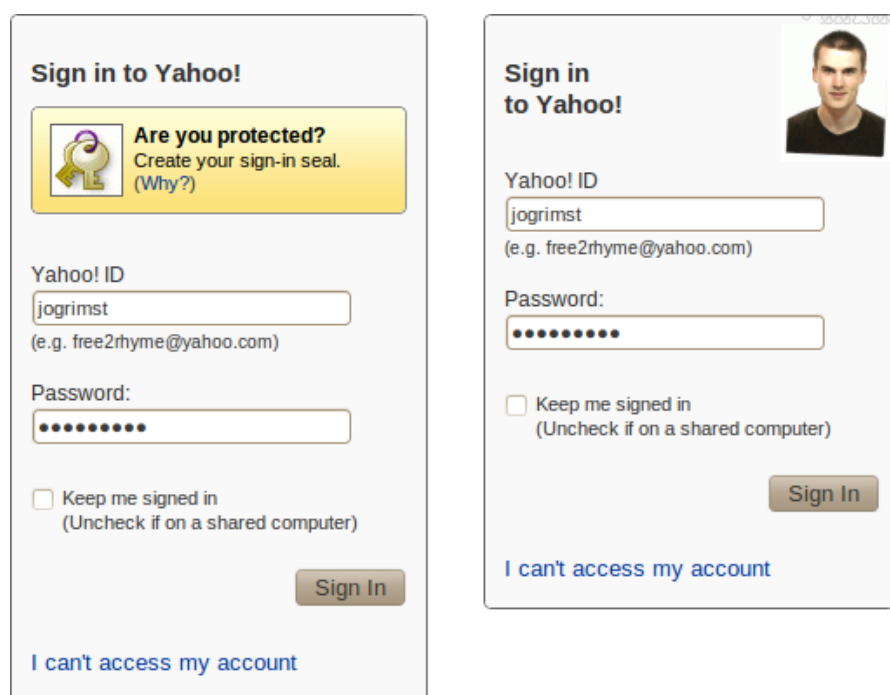


Figure 23: Yahoo! OP with and without sign-on seal

activated the sign-on seal, e.g. using a personal icon, he knows that he must look for it every time he signs on with the Yahoo! ID. If the seal is missing, or if it looks different, then the user knows that he might have landed on a phishing site. A sign-on seal is a secret between the browser where it is activated and the OP; it is not associated with the OpenID itself (this way, attackers cannot discover your sign-on seal even if they know your OpenID Identifier). For this reason, if other browsers or computers are used, the seal must be activated there as well.

<sup>28</sup><http://wordpress.com>

<sup>29</sup><https://protect.login.yahoo.com>

Also, the OpenID specification states that RPs only should redirect End Users to OPs in a top-level browser window where all controls are visible [6]. Obviously, with the address bar being visible, it would be easier for users to reveal an ongoing phishing attack. The specification also encourages RPs to educate their End Users about the potential for OpenID phishing attacks, accompanied by tools and techniques for defeating such attacks.

Note that today, *blacklisting* is the primary mode of phishing defense [9]; many browsers come shipped with lists of Web sites that cannot be trusted, i.e. known phishing sites. The sites in such lists are automatically blacklisted, so that users will receive warnings about them if accessed. To keep the list current, the browsers receive updates continuously. Also, many browsers allow their users to rate sites for trust and reputation, so that the browser can display alerts when potential phishing scam Web sites are being visited.

### 3.9 Certification of OPs

Section 3.7 (*Provider Authentication Policy Extension*) explained how RPs themselves are responsible for identifying trustworthy OPs since there is no trust model specified by the OpenID protocol [8]. It was also indicated that this situation introduces a need for some type of whitelisting, or *certification*, of OPs that over time have proven to be trustworthy. This way, RPs would feel more secure that their users are not tricked into interacting with OP look-a-likes during authentication. A whitelisting solution, however, would require a continuous maintenance of the list of trustworthy OPs.

Note that IdP whitelisting is not directly relevant to OpenID alone; SSO systems in general also suffer from attacks like phishing, so some kind of certification of trusted IdPs would also be needed in other SSO systems where malicious IdPs are a problem. With OpenID, the PAPE extension makes it possible for RPs to look up supported authentication policies of OPs, thus aiding RPs in choosing between multiple listed OPs depending on their authentication requirements [8]. This helps RPs to create their own whitelist, but still, a public list might be a better alternative. One suggestion is to use so-called *reputation brokers*, which will publish lists of prestigious OPs on demand from an RP [12]. An RP would also be able to ask a reputation broker about which PAPE authentication policies a certain OP has implemented.

## 3.10 Summary

This part has given an introduction to *OpenID*, a decentralized authentication protocol that can be used by SSO Web applications. The basic elements of the protocol were explained in Section 3.3; the *End User*, the *Relying Party* (RP), and the *OpenID Provider* (OP). Additional terms like *Identifier* and *Endpoint URL* were described as well.

The section that followed (*3.4 Creating an OpenID*) gave an explanation of how to proceed in order to obtain an OpenID. Some popular OPs were outlined, and the VeriSign PIP OP was used as an example for how to create and use an OpenID. Real-life scenarios of how to use the OpenID with various RPs were described. Also, the request and response parameters exchanged between an RP and OP were explained.

Next, some useful extensions to the OpenID protocol were described; the *SREG* extension (Section 3.5.1), an extension for easy exchange of user profile information, and the *PAPE* extension (Section 3.7), an extension for managing authentication policies.

Also, a section described usage of *multi-factor authentication* in OpenID (Section 3.6). Here, the concept of *One-Time Passwords* (OTPs) was explained, including various examples of how it can be applied. One example that was demonstrated is OTP generation using a hardware token named *YubiKey*. Usage of digital certificates and other methods were also mentioned.

The part ended with an explanation of various anti-phishing mechanisms (Section *3.8 Phishing Protection*), and a discussion of ways to whitelist OPs (Section 3.9) as a solution to the problem with OPs that cannot be trusted.



## 4 Security Assessment

### 4.1 Goal

The goal of this part of the thesis is to perform security assessment of Web applications utilizing SSO. As described in the following section, an application will be developed for the purpose of the assessment, and this application will utilize the OpenID protocol (which was described in Section 3). As stated in the introduction, the development will result in a running RP, and this will be signed into using various existing OPs, e.g. the Google OP. The objective of this part is basically to perform experimenting in order to reveal security vulnerabilities related to the usage of RPs and OPs in an SSO system.

The part starts by giving a description of the Web application, i.e. its functionality and the various technologies that it uses. Next, performing exploitation using some of the attacks described in Section 2 (*Theoretical Background*) will be attempted. First, CSRF will be executed against the RP. And then, Clickjacking attacks will be attempted against various OPs. Brief countermeasures to each of these attacks will be mentioned as well, and details can be found in Appendix D.1 and E.2. Note that theory explaining these attacks were given in the following sections: *2.5.1 Cross-Site Request Forgery (CSRF)* and *2.5.2 Clickjacking*.

### 4.2 Application Development

As part of the security assessment in this thesis, a Web application utilizing SSO will be used for demonstrating various security threats. What is important to note about this application is that it is an RP, allowing users to sign on using their OpenID Identifier. Additional development of an OP is out of scope for this work. Instead, existing OPs on the Internet (e.g. VeriSign PIP, Yahoo! ID, MyOpenID, and Google) will be used during the course of the assessment.

The Java programming language is used for the development, as well as together with the following technologies related to Java Web development:

- Apache Maven (project management tool)  
<http://maven.apache.org>
- Spring Framework (open source Java Web application framework)  
<http://www.springsource.org/about>
- Spring Security (open source framework providing authentication and access-control services). Note that Spring Security does not require

usage of the Spring Framework.

<http://static.springsource.org/spring-security>

- openid4java (library for OpenID-enabling of Java Web applications)  
<http://code.google.com/p/openid4java>

In addition, *Trac*<sup>30</sup> was used for handling wiki and issue tracking, as well as milestones during the development.

#### 4.2.1 Java Library: openid4java

In order to make it possible for users to sign on to the Web application using their OpenID, it is necessary to somehow make the application able to communicate with the OpenID protocol. For this purpose, *openid4java* is being used. The *openid4java* package is an open source library which allows to OpenID-enable Web applications written in Java. It also offers support for implementing OP servers.



The *openid4java* package supports specifications like OpenID Authentication 2.0, and the ones described in the previous section (3. *OpenID*), e.g. SREG and PAPE. The library helps developers exchange messages between RPs and OPs, but they still need to do the following themselves:

- Obtain the OpenID (URL) Identifier from the users of the Web site.
- Create authentication request for the identifier, and redirect the user to the OP using this request.
- Receive and verify the authentication response returned from the OP.

---

<sup>30</sup>Issue tracking system for software development projects: <http://trac.edgewall.org>

### 4.2.2 Spring Security

These steps listed above can be simplified further using a framework. *Spring Security* is one such framework, which uses openid4java and supports authentication of users logging in using an OpenID URL instead of a username/password combination. The framework offers several configuration options. As an example, Listing 3 shows how it is possible to define which attributes a user should be requested to provide when authenticating with an OP.

Listing 3: Spring Security configuration example

```
1 <openid-login>
2   <attribute-exchange>
3     <openid-attribute name="email" required="true"
4       type="http://axschema.org/contact/email" />
5     <openid-attribute name="fullname"
6       type="http://axschema.org/namePerson" />
7     <openid-attribute name="nickname"
8       type="http://axschema.org/namePerson/friendly" />
9     <openid-attribute name="dob"
10      type="http://axschema.org/birthDate" />
11     <openid-attribute name="gender"
12      type="http://axschema.org/person/gender" />
13   </attribute-exchange>
14 </openid-login>
```

As described in Section 3.5.1 (*Simple Registration Extension*), it is possible for the developer of the Consumer Web site (i.e. the RP) to mark which attributes that should be compulsory.

When using Spring Security for this purpose, one of two outcomes are possible. Either the user refuses to share personal information marked as required by the Web site, thus leaving the user unauthenticated and not signed in to the Web site in question. Or, the user successfully authenticates with the OP, hence causing a positive authentication response to be returned to the RP. In the latter case, Spring Security verifies the OP response and provides the user with a fresh session identifier. At this point, the RP would have access to the OpenID attributes returned from the OP, i.e. the e-mail address and, if provided, full name, nickname, date of birth, and gender.

### 4.2.3 Functionality

In the following, there will be given a brief introduction to the functionality of the Web application developed for this thesis. First of all, when the users want to sign on, they are presented with a page requesting an OpenID Identifier, as shown in Figure 24. Next, the user types in an OpenID URL and

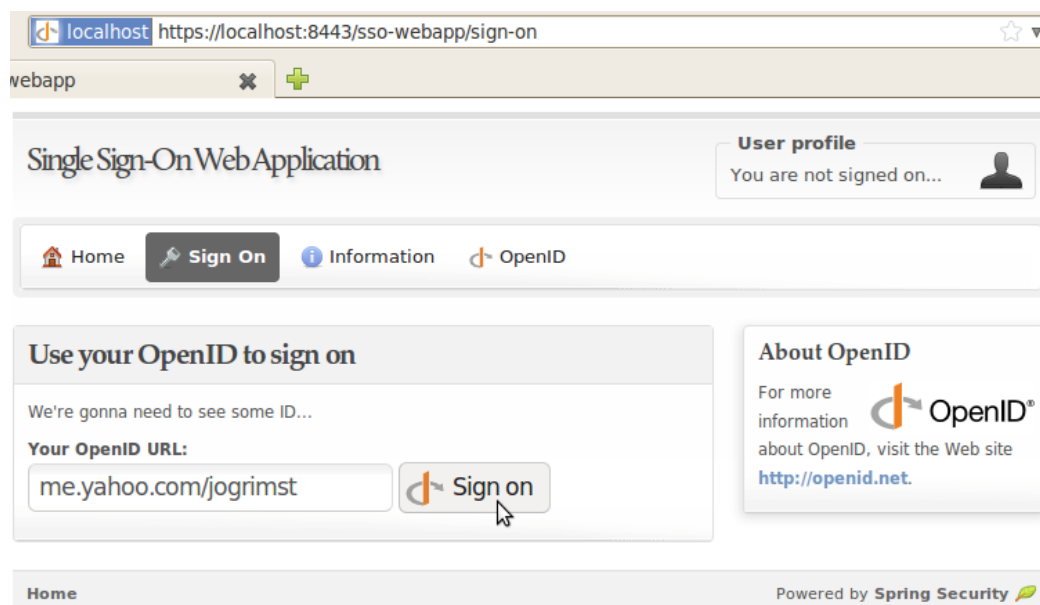


Figure 24: Web application sign-on page

is then redirected to the OP associated with the given OpenID. This example uses Yahoo! as OP, as seen in Figure 25. The user types in the credentials, and that leads directly to the point where the user has to agree to share information with the RP.<sup>31</sup> When clicking “Agree”, the user is redirected back to the Web application together with the authentication response. After having been successfully authenticated, the user is presented with his account details at the Web site (the RP), as shown in Figure 26. At this point, the RP has access to the user’s information, i.e. the attributes that the user agreed to share in Figure 25.

To see some of the log messages that were produced during this sign-on example, please take a look at Appendix F.1. Further explanation of the Web application will also be given in the section that follows (4.3 *Security Assessment*).

<sup>31</sup>Notice how the authentication steps with the Yahoo! OP are basically the same as those performed earlier using the VeriSign OP (Section 3.5).

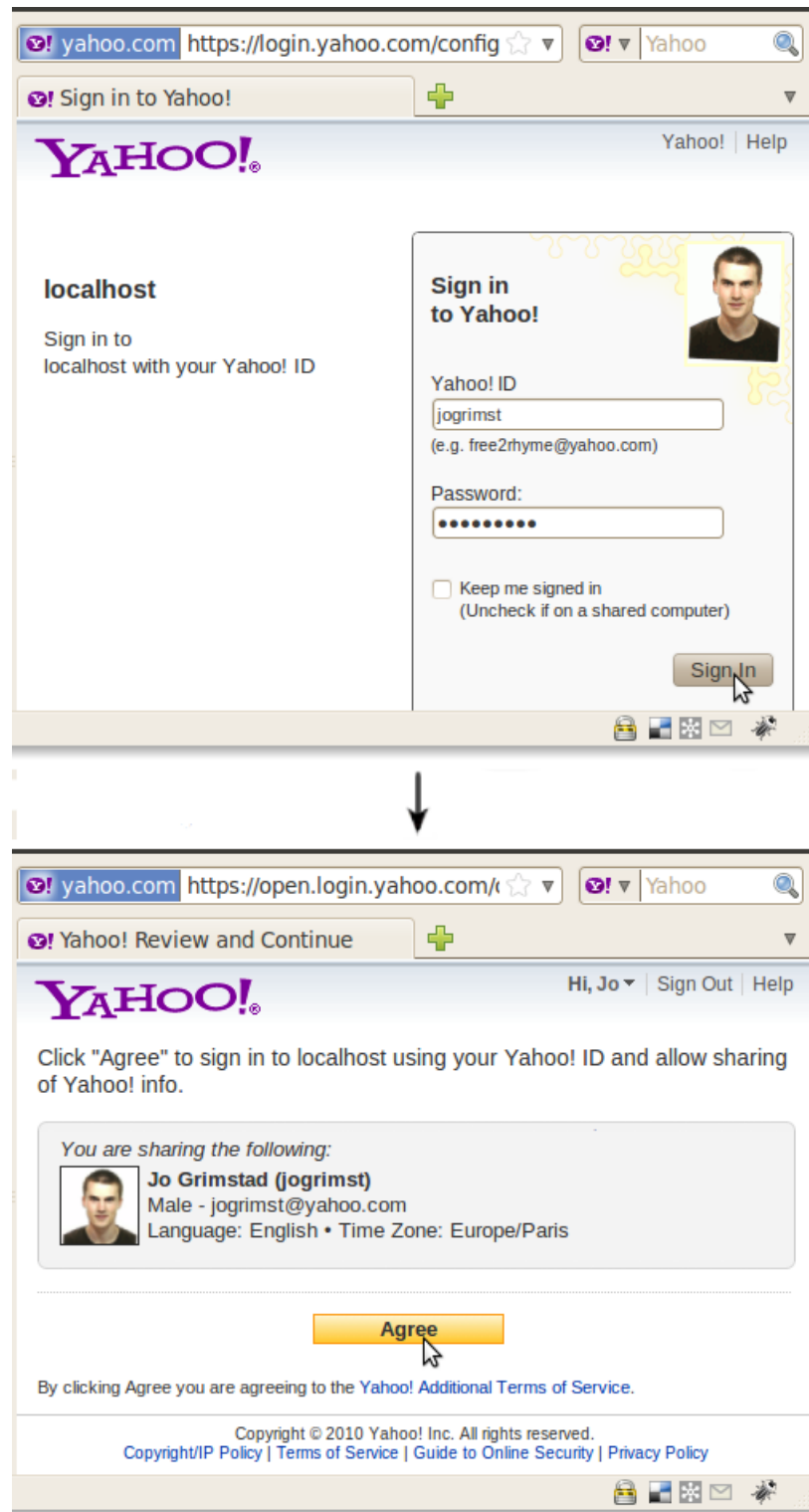


Figure 25: Agreement to share information at OP

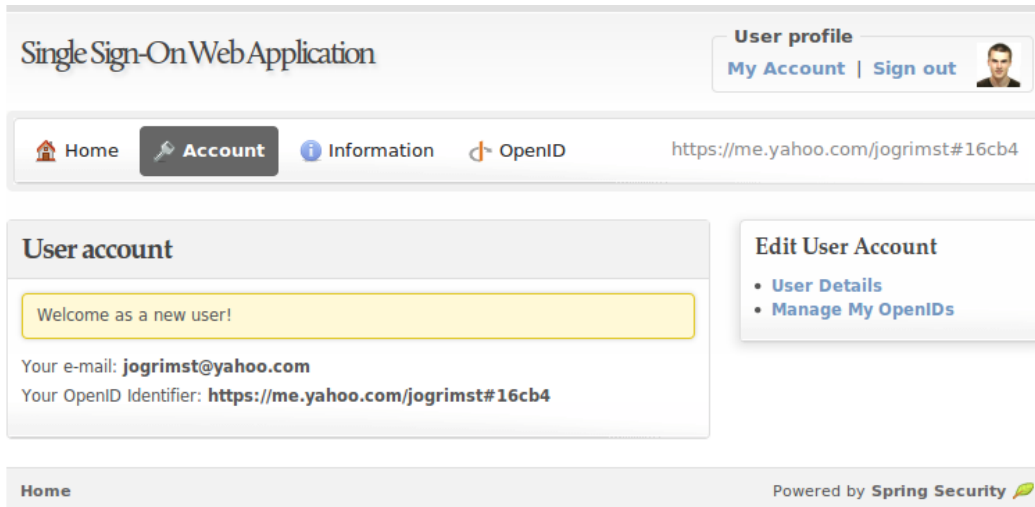


Figure 26: RP account details

## 4.3 Security Assessment

Having shown the core functionality of the Web application, let us begin with the assessment of its security. As indicated in the introductory part (Section 1.4), there are several different scenarios that can be considered when performing an investigation of security vulnerabilities in an SSO system.

A possible scenario is one where a malicious Web site is visited by a user who is signed on at an RP. If the RP, i.e. the service-providing Web site, has not implemented sufficient protection mechanisms, it might be susceptible to CSRF attacks. Another scenario is one where a user has authenticated at an OP, and then visits a malicious Web site which intends to trick the user into sharing information from her OP account without knowing about it (using Clickjacking).

Each attacker might have a different motivation for exploiting Web sites; some might have financial motivations, and some might be attempting to sabotage the work of a competitor, while others might be doing it to gain respect within a certain community, etc. In this part of the thesis, the assumption is that an attacker is interested in gaining access to a user's RP account or OP account. The following will describe scenarios where CSRF and Clickjacking can be utilized achieve such goals.

### 4.3.1 Attacking an RP Using CSRF

A typical scenario with OpenID-enabled Web sites is the fact that a user often will be logged in at many different places at once. This is inherent in

the nature of OpenID, and unless the users are so security-aware that they sign out each time they have been using a Web site, it is very likely that a user at some point in time will have multiple active sessions against different domains.

This situation facilitates *CSRF* attacks. As explained in Section 2.5.1, a *CSRF* attack is exploiting the situation where users are signed on at the page under attack, while visiting a malicious page. According to the *Same Origin Policy* (SOP) described in Table 3, Web sites with different protocols, hosts,

URL	Access allowed?	Reason
http://www.domain.com/	Yes	Triplet fulfilled
http://www.domain.com/dir/	Yes	Triplet fulfilled
https://www.domain.com/	No	Different protocol
http:// <b>sub</b> .domain.com/	No	Different host
http://domain.com/	No	Different host
http://www.domain.com: <b>8080</b> /	No	Different port

Table 3: The Same Origin Policy

or ports cannot access or manipulate each other's data using script languages [9]. So while JavaScript (as an example) cannot be used to access data from a different origin, we can still use *CSRF* to forge requests against it.

Several OpenID-enabled Web sites allow users to associate multiple OpenID Identifiers with a user account. One such example is [SourceForge.net](http://SourceForge.net), which was referred to in Section 3 *OpenID*. Another solution is to allow two OpenIDs; one primary and one alternative OpenID, like the way it is done at [stackoverflow.com](http://stackoverflow.com) (a Web site for questions and answers about programming). Others only allow to associate *one* OpenID identifier with each user account, but offer their users a way to change this to another one if needed. And there are also OpenID-enabled Web sites that only allows one, fixed, OpenID Identifier associated with each user account, without any way of changing it to another one.

The OpenID protocol does not dictate how RPs should handle the way OpenID Identifiers are to be associated with the Web site user accounts, so it is up to themselves how to implement this functionality. However, RPs should keep in mind that there are situations in which users might need to change or use another Identifier. Some people might use different OpenIDs for different purposes (e.g. work and personal), while others might have changed to an OP they felt was a better choice than the current one. Additionally, there might be occasions where OPs decide to end their service. For this

reason, users left without access to their accounts, should be provided with some way to recover the access which does not rely on an obsolete OpenID Identifier.

Figure 27 shows how such functionality has been added to the Web application developed for this thesis. By clicking “Manage My OpenIDs”, and

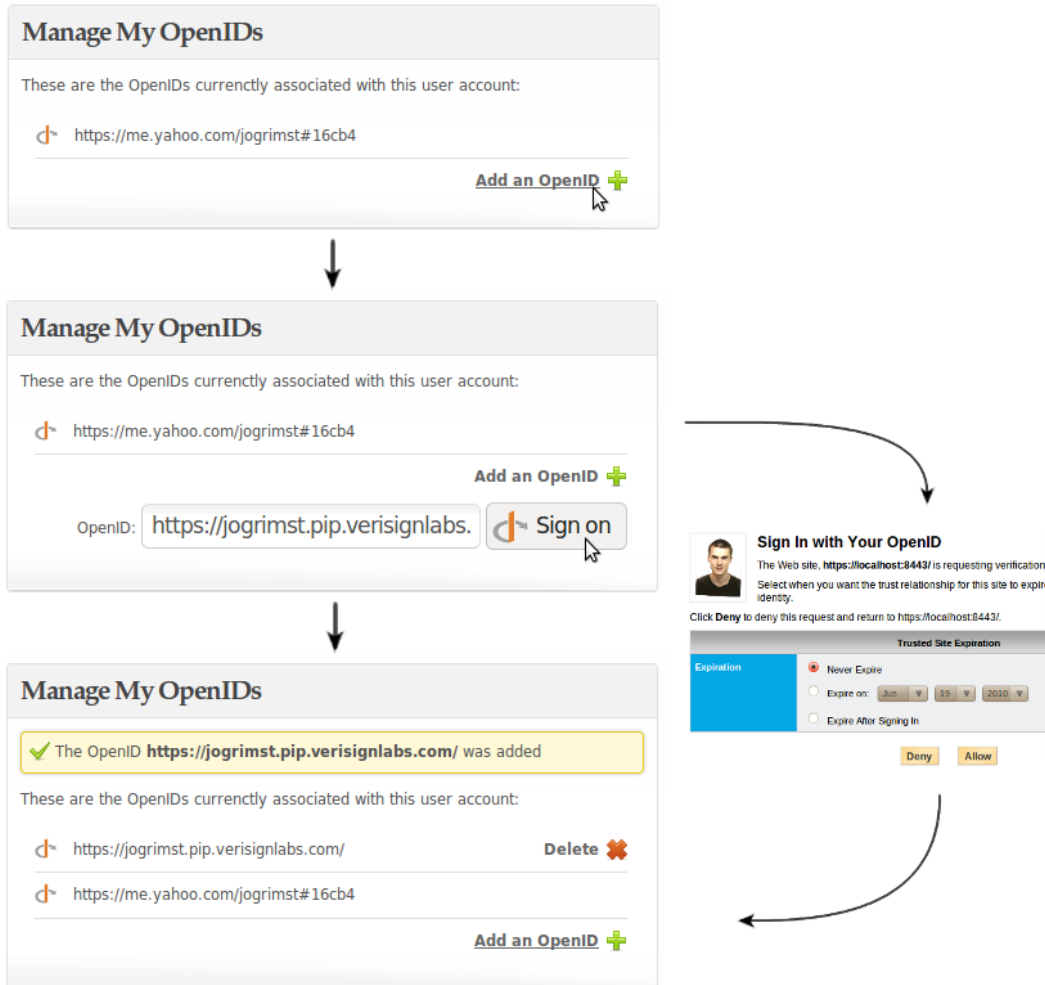


Figure 27: “Manage My OpenIDs”

then “Add an OpenID”, the user is able to associate additional OpenID URLs to the user account that is currently signed on. The Figure shows how the OpenID `jogrimst.pip.verisignlabs.com` is being associated with the account. Now, assume that an attacker tries to exploit this functionality. Depending on the degree of security implemented at the RP, the attacker might be able to perform a CSRF.



The RP Web application is located at `https://localhost:8443`, so we will use another origin (protocol, host, port), `http://127.0.0.1:8080`, when running the malicious Web site. Assume that the latter URL is owned by the attacker. Before being able to conduct a CSRF against the OpenID management site of the RP, the attacker needs to take a look at the form that is being filled by users when they are adding new OpenIDs. She could sign on, herself, and see that the HTML code looks like this:

---

Listing 4: Form used for associating additional OpenIDs

---

```

1 <form id="newOpenId"
2     action="/sso-webapp/user/manage-openids/add-an-openid"
3     method="GET">
4     <input type="text" name="openid_identifier" />
5 </form>

```

---

So far, she knows that submitting the form would produce an HTTP GET request made to the following URL:

```

https://localhost:8443/sso-webapp/user/manage-openids/
add-an-openid?openid_identifier=<OpenID>

```

This means that the attacker could set whatever she likes as value for the `identifier` parameter, and thus initiate an OpenID authentication request against the OP belonging to the given OpenID Identifier.

**HTTP GET Example** Assume that the attacker has created a Web site within another origin, `http://127.0.0.1:8080/sso-webapp/csrf/http-get`, and that this site contains the following code:

---

Listing 5: CSRF using HTTP GET

---

```

1 

```

---

Now, the goal of the attacker is to make the user visit this page. This can be achieved in several different ways, e.g. through social engineering, placing redirect scripts at other Web sites, etc. If she somehow succeeds in tricking the user into visiting the page, the user would just be presented with an empty white page, and would probably navigate back to the previous page or do something else. However, as shown in Figure 28, an HTTP GET request was made in the background against the following URL:

```

https://localhost:8443/sso-webapp/user/manage-openids/add-
an-openid?openid_identifier=http://trudy.evilsite.com/

```

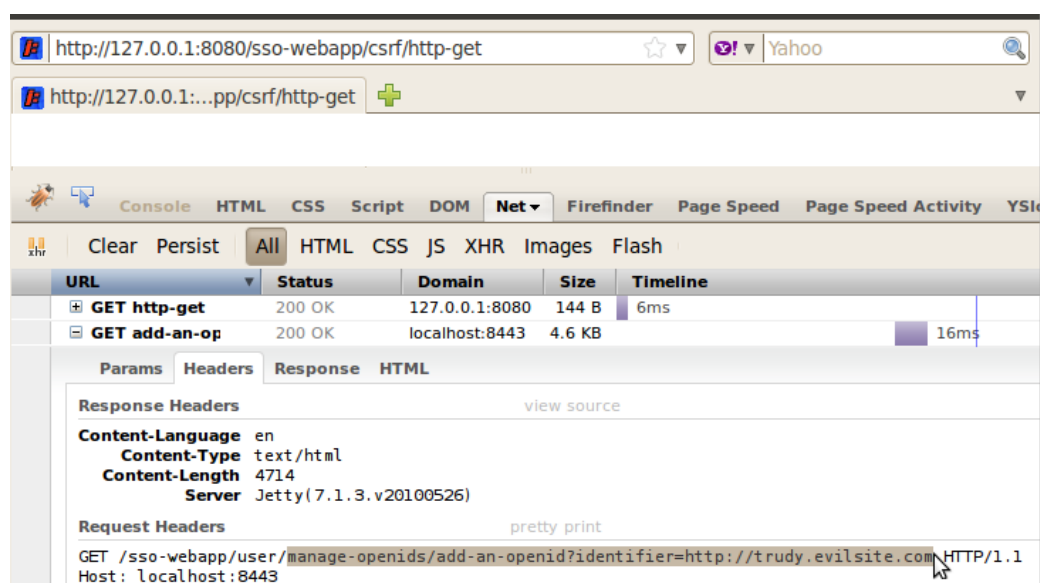


Figure 28: CSRF attack

And if the user at this point were to navigate back to the OpenID association overview, he would be presented with a list, as demonstrated in Figure 29. The figure shows that the malicious OpenID URL has been added to the collection of OpenIDs associated with the victim's account. Thus, the attacker is able to sign on with her OpenID to gain access to the compromised user account.

Note that `http://trudy.evilsite.com/` is not an existing OpenID; a functioning OP server has not been developed at `evilsite.com` for the purpose of this demonstration. But assume, in this case, that the attacker could have created an OP for the sole purpose of handling authentication requests for *her own* OpenID Identifier. In such a scenario, it is up to the attacker how to perform the authentication at the OP. What this means, is that the attacker could configure the malicious OP to simply return a positive authentication response when an RP requests authentication of `http://trudy.evilsite.com/`. This is possible because RPs has no control of what OPs does during authentication on the server side. As long as the malicious OP manages to communicate with RPs using the OpenID protocol, it can fake positive authentications.

Obviously, this type of attack would require a lot of work, including relatively advanced programming skills possessed by the attacker. However, if the attacker discovers an RP allowing users to have multiple OpenIDs with their accounts and that Web site contains valuable information, it might be

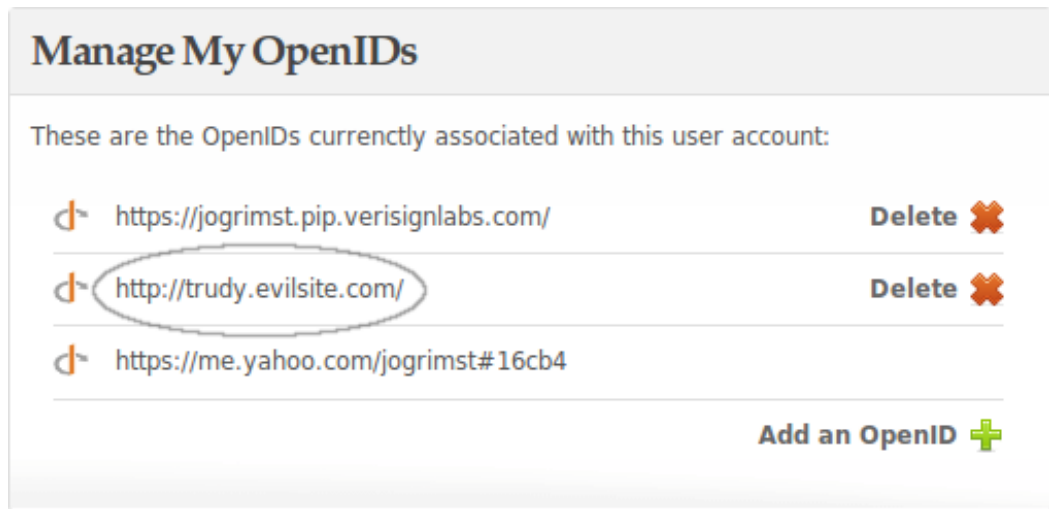


Figure 29: Results of CSRF attack using HTTP GET

worth it to construct a fake OP. As in any other case, the attacker will need to compare the possible profit up against the time required to develop it.

Also note that a variant of this attack might succeed even if the user is not already authenticated! This, however, presupposes that the RP under attack utilizes a technique for remembering the original requests performed by unauthenticated users. As an example, the Spring Security holds this capability. So theoretically, the CSRF attack might succeed in a slightly different order as well: 1) the user is tricked into visiting the malicious site; 2) the malicious site forges a request against the RP; and 3) the user signs on at the RP. . In such a scenario, the CSRF does not actually succeed at the exact moment when the request is first performed. Instead, if the user decides to sign on with the RP within its session expiration time, the RP will *retry* the original request that was previously forbidden. This variant, however, is a lot easier for the user to discover because she will be redirected to the “Manage My OpenIDs” page and see a message saying that the `http://trudy.evilsite.com/` Identifier was successfully added.

**HTTP POST Example** The example above has demonstrated a relatively simple way to perform CSRF, where the RP had not added any kind of protection against this form of attack. As a basic form of defense against CSRF attacks where `GET` requests are forged using the `src` (or `href`) attribute of elements like images, iframes, scripts, and stylesheets, a Web site can require that the form must be submitted using `POST` as request method [2]. This way, accessing the URL manually or including it as source of some element would not work because the server would be expecting another method. For instance, trying to access the URL used in the example above after having modified the server-side code to require `HTTP POST`, would result in an error message looking something like this:

```
org.springframework.web.HttpRequestMethodNotSupportedException:
Request method "GET" not supported
```

At first sight, introduction of `HTTP` method restrictions might work as a reasonable protection mechanism. Unfortunately, it is also possible to forge `POST` requests [2]. For instance, JavaScript can be used to circumvent the restriction. The attacker can simply copy the code of the original form into her own malicious site, pre-fill it with the fake OpenID Identifier, and then point it to its destination under attack, and submit it using JavaScript:

Listing 6: CSRF using `HTTP POST`

---

```

1 <form id="newOpenId" method="POST" style="display: none"
2     action="https://localhost:8443/sso-webapp/user/
3     manage-openids/add-an-openid"
4     <input type="text" name="openid.identifier"
5         value="http://trudy.evilsite.com/" />
6 </form>
7 <script>
8     document.getElementById("newOpenId").submit();
9 </script>
```

---

This code would actually submit the form as if done by the user. That would redirect the user to the “Manage My OpenIDs” page, where she would probably notice that a new Identifier has been added (see Figure 30). In this case, it seems more probable that the user would reveal the attack, and thus perform some action to restore the damage that was done. So, to summarize, CSRF is still possible to perform even though the page under attack only accepts `HTTP POST` requests, but pages that can be invoked with `HTTP GET` are often easier to exploit.

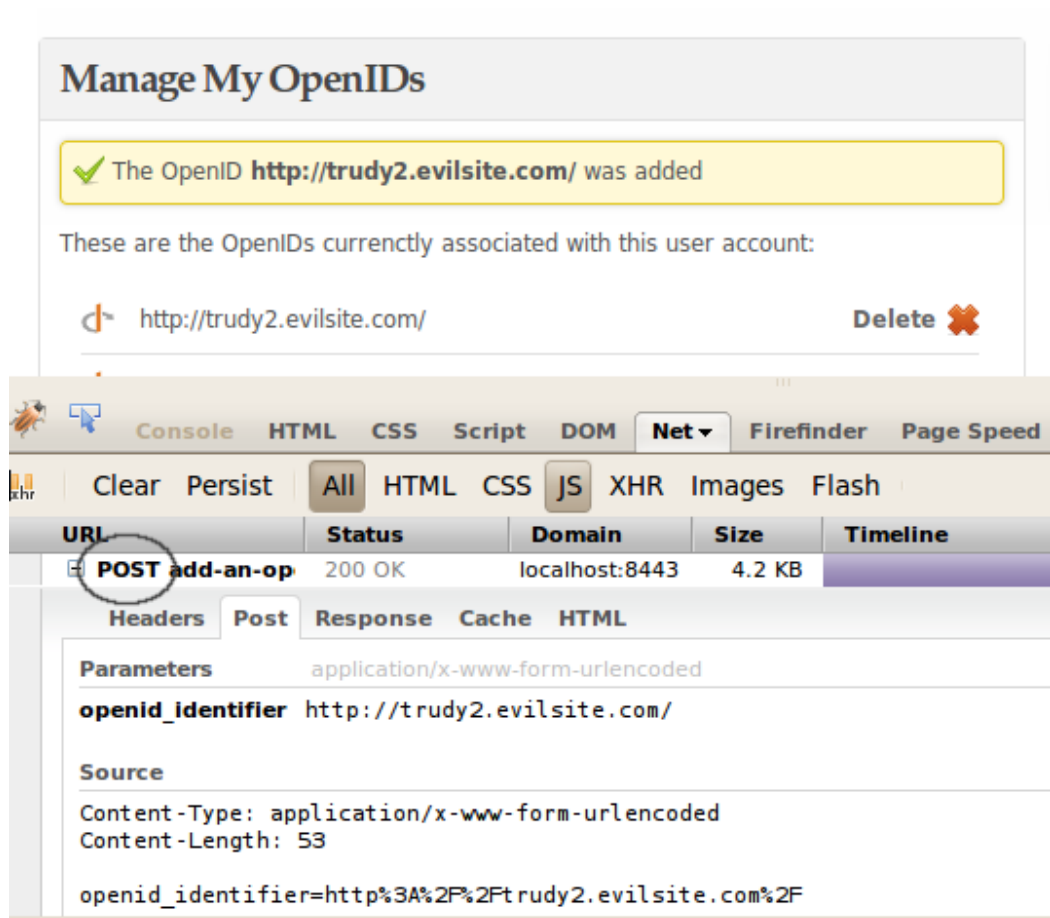


Figure 30: Results of CSRF attack using HTTP POST

### 4.3.2 Countermeasures Against CSRF

There are many protection mechanisms that can be employed in order to prevent, or at least, reduce the risk of CSRF attacks. Section 4.3.1 (Attacking an RP Using CSRF) showed how usage of `POST` as the HTTP method would make it a little more difficult for an attacker to perform CSRF, while certainly not impossible.

Another common way for Web developers to prevent CSRFs is to check the HTTP `Referer`<sup>32</sup> header. This header specifies where a user comes from. If a user clicked on a link, or was redirected from another Web site, then that site's URL will be the value of the header. So, many Web sites simply perform a check whether the value of the `Referer` header is a URL located at the same domain, and not some external one. However, as this header is user-defined input (like cookies or form values), its value might easily be tampered with [9].

Another, and more secure, solution would be to force users to re-authenticate each time an important request is made to the Web site. E.g. a request for changing user information could require the user to type her password to confirm the change. This might be experienced as obtrusive, so to improve user-friendliness, a better solution (depending on the situation) might be to use unpredictable *tokens* included in each request. Appendix D.1 explains this protection mechanism in detail.

### 4.3.3 Attacking an OP Using Clickjacking

Section 2.5.2 explained the theoretical background for Clickjacking. In this part, there will be a demonstration of how it was possible to perform an actual Clickjacking attack against an OP (Google). The Web application developed as part of the thesis will be used to provoke the attack.

As with CSRF, Clickjacking also exploits Web sites where users already have authenticated. First of all, let us assume that the malicious Web site in this example knows the OpenID URL of the user,<sup>33</sup> and that the URL ends with `google.com`. To be able to successfully execute the attack, the malicious Web site first needs to take a look at the authentication form of the Google OP.

In order to be redirected to the Google OP authentication form, a user could simply type his OpenID Identifier into the input field on the sign-on

---

<sup>32</sup>The name of the header is not spelled correctly; it uses one r instead of two. The W3C standard itself misspelled the word [9].

<sup>33</sup>Note that OpenID URLs can be easily collected many places. As an example, they are often found publicly available as part of user comments at various Web sites.

page (shown in Figure 24). This would redirect the user to the page shown in Figure 31. However, if the malicious Web application already knows the

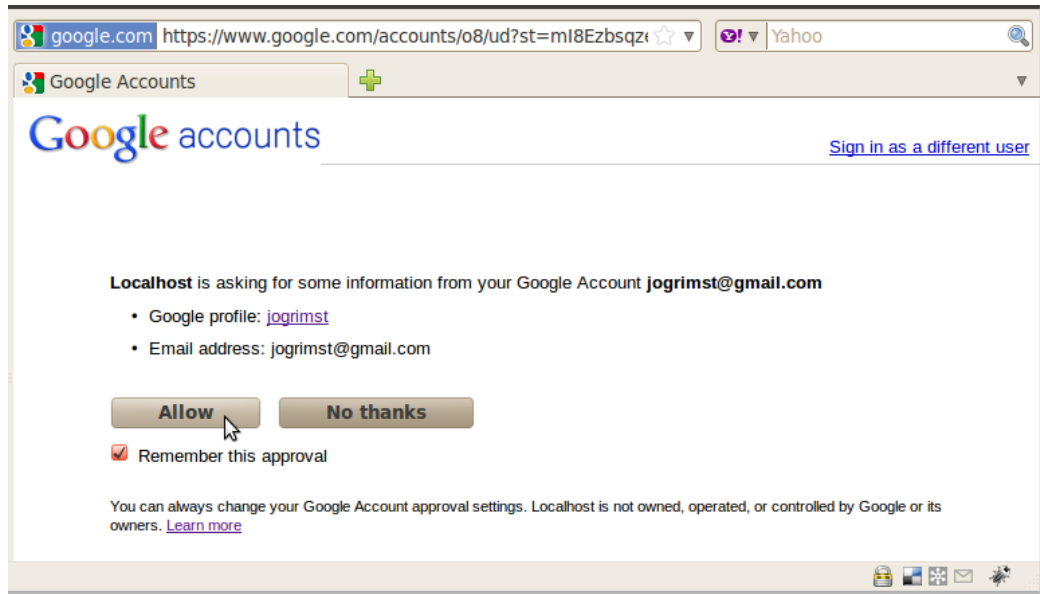


Figure 31: Authentication with Google OP

OpenID Identifier of the user, the same authentication page could be reached simply by sending the user to the following URL:

```
https://localhost:8443/sso-webapp/j_spring_openid_security_check
?openid_identifier=https://www.google.com/profiles/jogrimst
```

Requesting this URL would entail the OpenID authentication process to be handled as before, equivalent to the scenario where the user types in his OpenID Identifier himself. This also means that the Web site could actually include the Google OP authentication form in an `iframe` like this:

Listing 7: Inclusion of Google OP in an `iframe`

```
1 <iframe src="https://localhost:8443/sso-webapp/j_spring_openid_security_check
2     ?openid_identifier=https://www.google.com/profiles/jogrimst">
3 </iframe>
```

This way, the external OP authentication form (located at `https://www.google.com/accounts/o8/[...]`) would be presented to the user who is still located on the original insidious domain, i.e. `localhost`. Figure 32 shows how the Google authentication form is presented to the user in an `iframe` at the `localhost` domain.

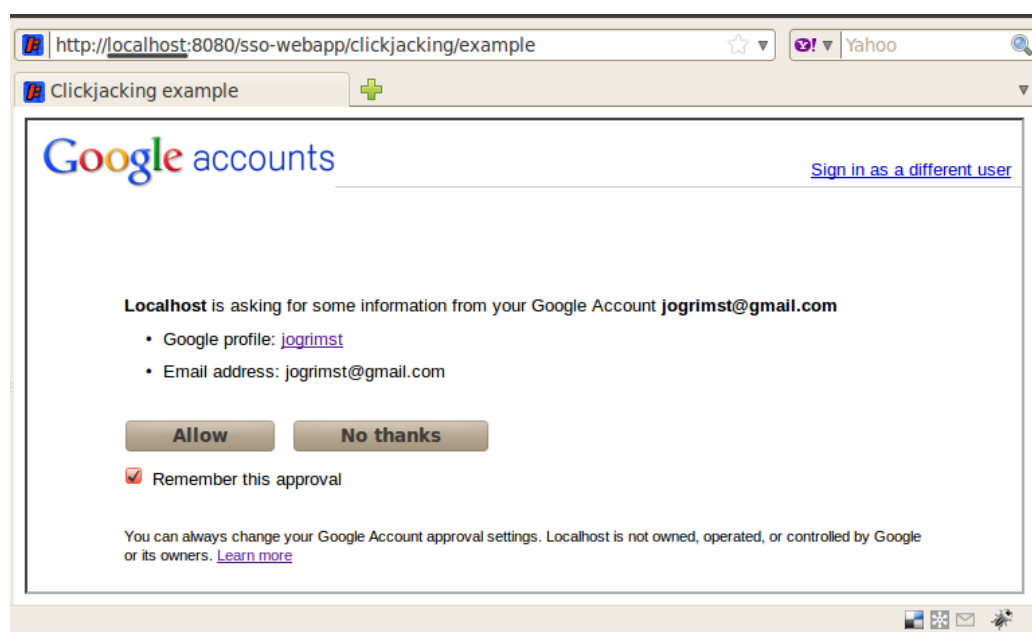


Figure 32: Authentication with Google OP located inside an `iframe`

Next, assume that an attacker has created some Web page that is meant to work as a diversionary maneuver, like the one shown in Figure 33. This is a Web page that tries to present some fake content to the user. To view the rest of the text, the user is encouraged to click the button at the bottom of the page. This element does not necessarily have to be a button; it could easily be substituted by a link, an image, or a checkbox, etc. The goal of the attack is to make the user perform a click at a given location. There are many different ways to achieve this. To make a user click something, it is important to make the page seem trustworthy. Basically, the attacker needs to use social engineering to fool the user into doing the following:

1. Visit the cloaked page.
2. Click some element on the page.

With the `iframe` (Figure 32) and the fake page (Figure 33) in hand, the attacker can assemble the two into a mixture that can be used to mislead the user. By default, `iframes` and their contents are opaque, but by using CSS (as described in Section 2.5.2), the attacker can make them appear invisible to the user. Figure 34 explains the concept. First of all, there are **two** layers of content; a lower layer, and an upper layer (the one closest to the user). In this example, the `iframe` containing the Google OP authentication form



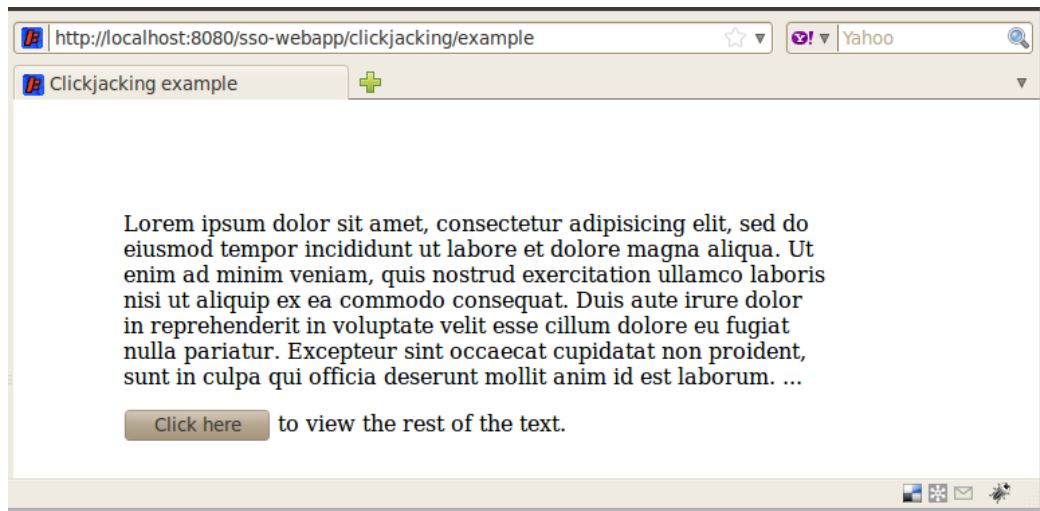


Figure 33: Fake page presented to the victim

is the layer located at the top, while the fake page is located at the bottom. The upper part of Figure 34 shows a scenario where the opacity of the `iframe` is set to 100% (default). The lower part of the figure shows a scenario where the opacity has been reduced, thus making the `iframe` partially invisible.

Using CSS's `z-index` property, the `iframe` layer has been placed directly above the fake page layer (for code details of the attack, please see Listing 12 in Appendix E.1). Now, if the opacity of the `iframe` is set all the way to zero (also with the help of CSS), a user would think that he is clicking on the "Click here" button that he sees on the screen, while actually it is the form in the background that is being interacted with! In this particular example, it is the "Allow" button for allowing user information to be shared with the malicious RP that is pressed. At this point, given that the `localhost` domain is controlled by an attacker, the user actually authenticates the malicious RP with his account without knowing about it (since it all happens in the background).

The main consequence of such an attack is that the malicious RP can steal user information (e.g. e-mail, full name, date of birth, etc) from the victim's OpenID account. This happens completely hidden from the user, so unless the victim at some later point takes a look at the OP's account history showing all of the user's visited sites, there is little probability that the attack will be discovered afterwards.

To demonstrate the success of the Clickjacking attack, take a look at Figure 35. This shows how a click on the "Allow" button redirected the victim to the, in this case malicious, RP. Note that the opacity of the `iframe`



Figure 34: Reducing opacity of OP authentication form layer

in the figure has been increased for the purpose of the demonstration. With a real attack, however, this would be set to zero, thus presenting the user with the fake page (given in Figure 33) even after the success of the attack. This means that she will notice that nothing happens when she tries to click the button located at the fake page, since it is not possible to interact with it directly. At this point, she might become suspicious, or she might give up and just navigate back or somewhere else.

In this particular example, the RP developed for this thesis was used for playing the role as a malicious Web site controlled by an attacker, while my Google OpenID account (<https://www.google.com/profiles/jogrimst>) was the one under attack. Note that an attacker is free to choose which Web application to use, but there are two main requirements that should be fulfilled:

1. It must support the OpenID protocol; e.g. successfully handle authentication requests according to the protocol.
2. It must exist in a location controlled by the attacker, in such a way that the attacker is able to obtain the stolen user information.

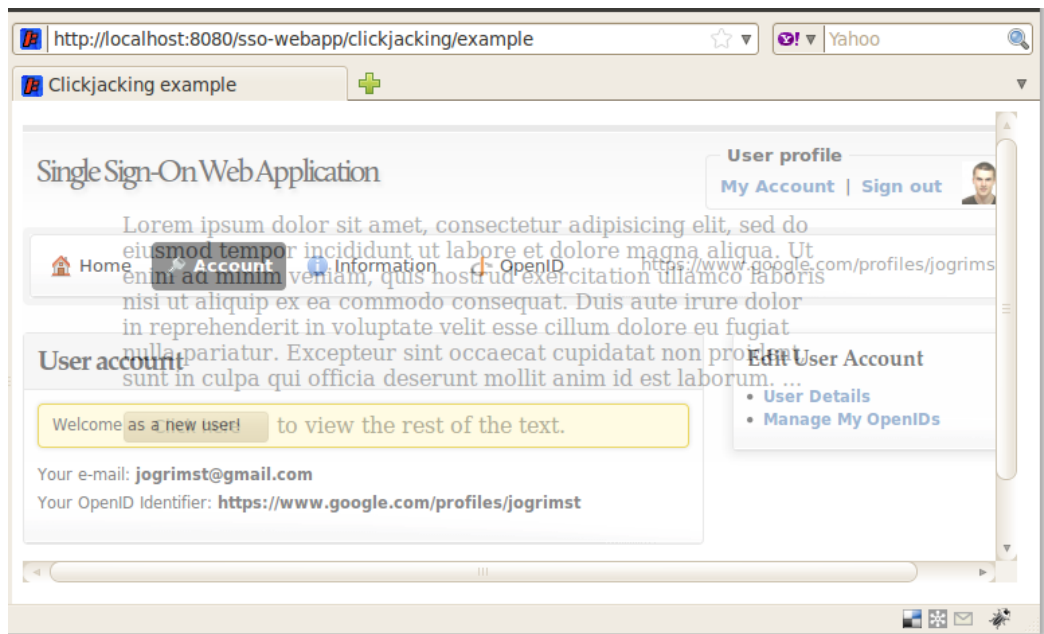


Figure 35: Successful Clickjacking attack

**Gaining Access to User Account at the clavid OP** The scenario above showed how an attacker can manage to obtain access to a user account at an RP. In the following, we will consider an even more lucrative attack, namely gaining access to the OP account of an End User. If this can be done, the attacker would not only have access to one RP Web sites, but *all* RPs associated with that particular OP account.

First of all, let us use the clavid OP as an example, and look at the page where an authenticated user can edit their account settings. Figure 36 shows the page where a user can change his password. The user simply types in

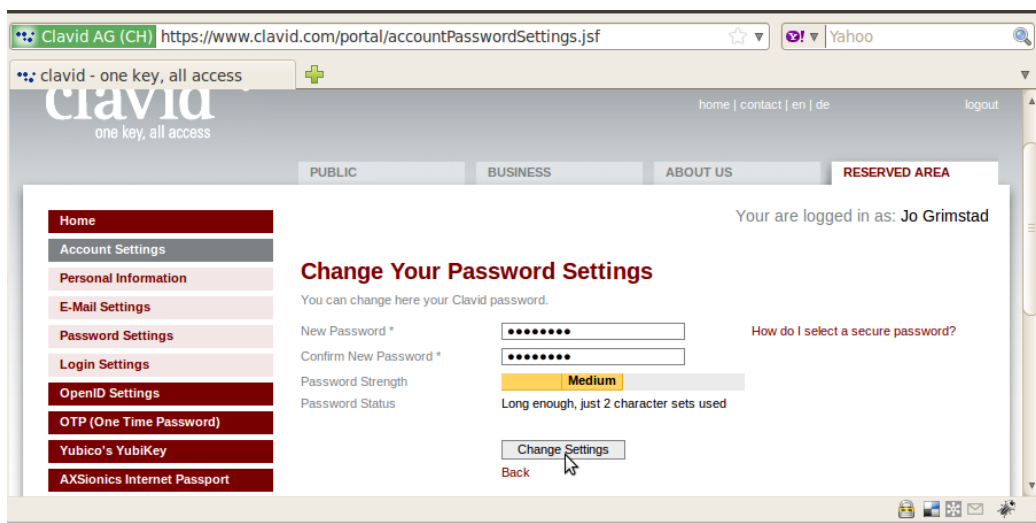
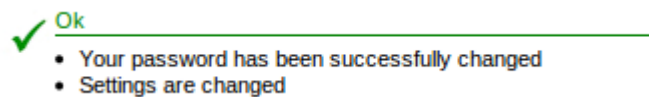


Figure 36: Password settings at clavid.com

the new password, then once more in the confirmation field, and clicks the “Change Settings” button. If the given password matches the requirements set by clavid, a confirmation message is shown, stating that the settings were successfully changed:



As it turns out, this page is vulnerable to a Clickjacking attack. In this case, the goal of an attacker would be to trick an already authenticated user into typing in a known text string into the input fields (using an invisible `iframe`) and click the button.

An example scenario could be that the victim user is presented with a form for commenting on some article, etc. Then, when submitting the

comment, the spoof Web site would ask her to type in a *CAPTCHA*<sup>34</sup> code twice in order for the comment to be submitted, as shown in Figure 37. Most

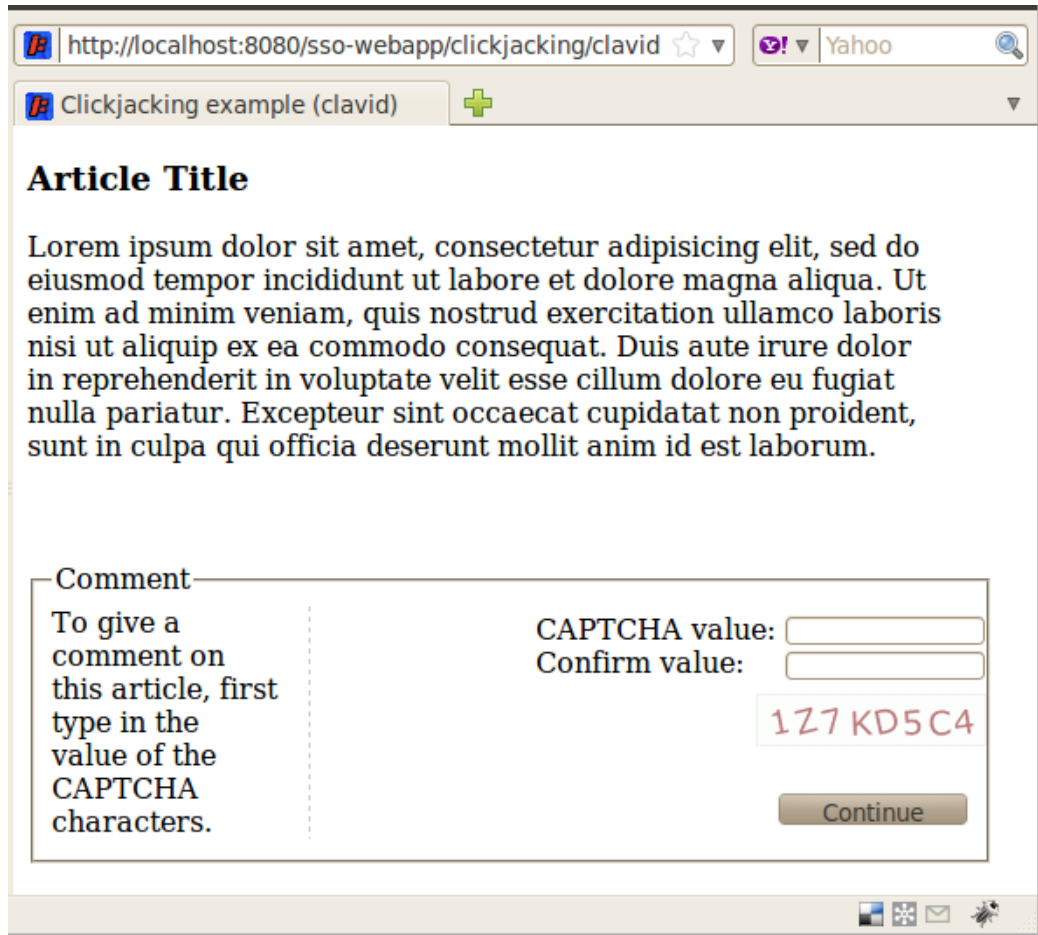


Figure 37: A fake page with a commenting form

people are familiar with the fact that CAPTCHA often is used on order for Web sites to avoid problems with spam robots. This way, it is reasonable to believe that it would be possible to make a user type the characters into the two fields and hit the button (given that the content of the article is of interest to the user).

Normally, a new CAPTCHA image is generated each time the page is loaded. However, the attacker needs to know the value of the password that as written into the clavid form. For this reason, the CAPTCHA image on this

<sup>34</sup>A CAPTCHA code is an image with jumbled characters. It stands for *Completely Automated Public Turing test to tell Computers and Humans Apart* [1]

spoof page would *never* change. I.e., the attacker knows that if an End User authenticated with clavid is tricked into filling this form, the new password of that user's clavid account would be 1Z7KD5C4.

To demonstrate the concept, Figure 38 gives a visualization of what the page would have looked like for a user if the opacity of the `iframe` had been adjusted to about 30%. It illustrates how the pair of input fields and the button are located directly above the clavid Web page.

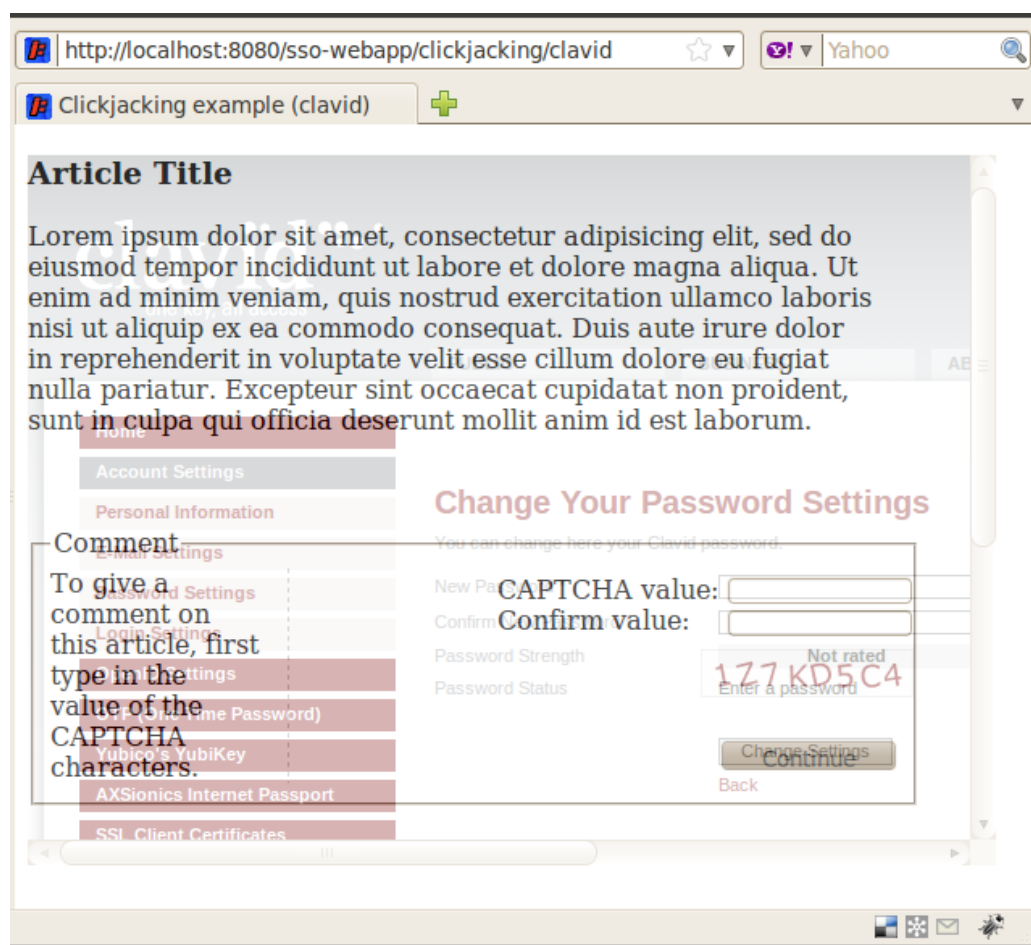


Figure 38: Partially opaque `iframe` containing clavid page

#### 4.3.4 Countermeasures Against Clickjacking

The level of technical skills needed to perform a Clickjacking attack is relatively low; only basic HTML and CSS skills are needed in order to place

an `iframe` above another page and make it transparent. This makes implementation of protection mechanisms even more important. However, a higher level of skills might be needed to create the page to be included in the `iframe`, especially if it is supposed to support authentication mechanisms of the OpenID protocol, like in the example above. This will, of course, depend on the specifics of each attack.

The most common defense for Clickjacking is called *frame busting* [15]. When frame busting defense has been added to a Web page, it will not work inside an `iframe`. This means that if any site tries to include such a page in a sub-frame, the included page will refuse to work, and bust out of the frame. Obviously, without a functioning `iframe`, a Clickjacking attack would no longer be possible.

There are many different ways to implement frame busting. The following method can be used to break page framing [11]:

Listing 8: Frame busting code

---

```
1 if (parent.frames.length > 0) {  
2     top.location.replace(document.location);  
3 }
```

---

This script is contained within the Web site that wants to avoid being included in `iframes`. It checks if there are any parent frames, and if this is the case, it tells the browser to redirect to the URL of the Web site (`document.location`). This way, the Web site is presented directly in the browser instead of being included inside a frame at another page. Actually, this countermeasure (and others) can be circumvented [15], but details about that is out of scope for this thesis. However, it is important to note that using frame busting is a lot better than using no defense mechanisms against Clickjacking at all. Another possibility is to use an HTTP header named `X-FRAME-OPTIONS` in order to restrict the usage of `iframes`. Details about the functionality of this header can be found in Appendix E.2.

As expected, there are differences between OPs with regards to protection against Clickjacking attacks. As an example, in Section 4.3.3 the Google OP was demonstrated to be vulnerable to the attack, while experimenting showed that VeriSign PIP and others were not vulnerable to the same attack. Table 4 shows the varying degree of Clickjacking defense for a group of popular OPs. The experimentation has looked at the ability of each OP to break out of `iframes` when using various browser types, and shows that some OPs lack such a defense mechanism.

Context	Browser	Attack possible?
Google	Firefox 3.6	Yes
	IE 7	Yes
	IE 8	No
	Chrome 5	No
	Opera 10.6	No
VeriSign PIP	Firefox 3.6	No
	IE 7	No
	IE 8	No
	Chrome 5	No
	Opera 10.6	No
clavid	Firefox 3.6	Yes
	IE 7	Yes
	IE 8	No
	Chrome 5	Yes
	Opera 10.6	Yes
Yahoo! ID	Firefox 3.6	No
	IE 7	No
	IE 8	No
	Chrome 5	No
	Opera 10.6	No
myOpenID	Firefox 3.6	No
	IE 7	No
	IE 8	No
	Chrome 5	No
	Opera 10.6	No

Table 4: OP comparison of Clickjacking vulnerabilities



## 4.4 Summary

This part has shown how a security assessment of SSO Web applications was performed. The part started by giving a description of the Web application that was developed as part of the thesis (4.2). This application is in practice an RP, where End Users can use their OpenID in order to sign on. It is up to the user which OP to use; the application allows for any OP to be used (as long as it has implemented support for the OpenID protocol).

Next, in Section 4.3, there was performed experimenting with Web security threats like *CSRF* and *Phishing*. It was shown how an RP could be attacked using CSRF in order for an attacker to gain access to a user's account. The difference between the effect of **GET** and **POST** requests was discussed, as well as various countermeasures to CSRF.

Section 4.3.3 showed how Clickjacking could be used to attack an OP. By including the authentication form of OPs in sub-frames, it was demonstrated how it is possible to trick users into authenticating malicious RPs using their active OP account session. The same section also demonstrated a more complex form of Clickjacking, which made it possible to actually gain access to a user's OP account. An overview of OPs vulnerable to Clickjacking was made for various browser versions. And possible countermeasures against the attack were explained at the end.

## 4.5 Discussion

The results of the security assessment shows that out of a group of 5 chosen OPs, two of them suffered from security vulnerabilities (Google and clavid). The vulnerabilities that were discovered made it possible for an attacker to do the following:

- Take over RP account of victim.
- Take over OP account of victim.

Obviously, gaining access to a user's OP account is a lot more attractive for an attacker, since this would make it possible to access all the RPs associated with that OP account.

The success of the attacks explained in Section 4.3.1 (*Attacking an RP Using CSRF*) and Section 4.3.3 (*Attacking an OP Using Clickjacking*) relied on certain assumptions. First of all, it was assumed that the attacker somehow knew (or guessed) that the user was already authenticated at the sites under attack. Also, when demonstrating OP attacks (Section 4.3.3), it was assumed that the attacker knew the OpenID Identifier of the victim user.

It is also worth pointing out that OpenID Identifiers in most cases are not considered sensitive information, so it is assumed that these can be obtained easily, e.g. by using robots to scan Web sites.

When the security vulnerabilities were discovered at Google and clavid, the companies were contacted by e-mail and notified about the possible attack (using an extract from the thesis). A security issue report was filed in Google's reporting system, and a few days later Google responded and acknowledged the vulnerability. They informed that they had considered Clickjacking protection via use of the `X-FRAME-OPTIONS` header (described in Appendix E.2). As demonstrated in this thesis, users of browsers like Firefox 3.6 and IE 7.0 (or earlier versions) are still vulnerable to the attack, and as of today this is a significant number of users. I have shared my thoughts on how to protect against Clickjacking for the users of these browser version, and Google informed me that they would use an internally developed framebusting script for the particular vulnerability discovered in this thesis.

The possible Clickjacking scenarios discovered at the clavid OP have also been reported. The OP was notified about the attacks explained in Section 4.3.3, where it was shown how it is possible to use Clickjacking in order to gain access to a user's RP account or OP account by including pages at `clavid.com` in an `iframe`. After some e-mail correspondance in order to clarify the concept for clavid, they have informed me that they will implement a fix for the security hole in a future release of the `clavid.com` OP.

## 5 Conclusion and Further Work

### 5.1 Evaluation and Conclusion

This Master's thesis has explained the concept of SSO, and it has made an assessment of the information security in Web applications that are utilizing SSO for user management. The results of the thesis show how it is possible for an attacker to trick users into performing requests without actually being aware of it. As an example, it was demonstrated that CSRF could be used to make a victim unknowingly perform harmful requests against a Web site where she is already authenticated. Also, usage of Clickjacking demonstrated how it was possible to successfully hijack SP and IdP accounts by placing an invisible content layer between the user and a fake Web page.

Obviously, the success of these attacks relies on the fact that the victim users have authenticated with the sites under attack *before* they are executed. The goal of an attacker utilizing CSRF and Clickjacking is to somehow perform an interaction with a Web site where a user has already signed on. If the user has not signed on anywhere, the attacker cannot use the mentioned techniques in order to hijack a user's Web site accounts.

**Security Responsibilities** The success of the attacks shows that all of the participants in a SSO system have a *security responsibility*, i.e. the End User, SP, and IdP. All of them should abide to certain best practices regarding information security. As mentioned earlier, the End User is responsible for picking a suitable IdP. The users should be aware that as the number of SPs associated with an IdP account increases, it seems more attractive for attackers to gain access to it. In order to avoid their IdP account to be compromised, users should ensure that the password they pick is strong, and they should consider using multi-factor authentication as an additional mean of protection. Not all IdPs support multi-factor authentication, so this is also related to the correct choice of IdP. Also, End Users should be aware of phishing when using SSO Web applications, and they might protect themselves with techniques like sign-on seals or manually typing in the URL of their IdP when authenticating. In addition, users should manually place `https://` in front of their Identifiers to avoid DNS poisoning attacks against their SP. Also, users should **never** share their password with sites other than their IdP. As mentioned in Section 2.3, SLO is implemented in some SSO systems, but not in all (e.g. OpenID). So if using a SSO system where SLO is lacking, a user should be aware that it is also necessary to log out of the IdP after logging out of an SP (especially if using a public computer).

SPs also have a security responsibility when it comes to protecting their

users. First of all, they need to be aware of the danger of CSRF. If an End User signs up at a large number of SPs using her IdP account, there is a possibility that some of those sites might be malicious ones. State-changing user operations (e.g. update of e-mail address or password) that might enable an attacker to gain access to the account **must** be protected against CSRF. A solution could be to force users to re-authenticate when performing sensitive operations like changing password, etc. Usage of unpredictable tokens is another protection mechanism RPs can use (as mentioned in Section 4.3.2 and described in Appendix D.1).

Needless to say, IdPs have a huge security responsibility. First of all, they are responsible for protecting data stored about all of their users. Additionally, IdPs depend on a good reputation amongst SPs in order to be included at the SPs as one of the alternative IdPs for users to choose between during sign-on. As with SPs, IdPs are also encouraged to educate their users about phishing, i.e. to teach them how to best separate the genuine IdP login site from phishing attempts. Also, IdPs should use HTTPS as the preferred protocol for their login page. Since the user accounts protected by the IdPs are entrance points to multiple associated SP accounts, user credentials should always be over HTTPS. The thesis has also demonstrated that Clickjacking is a severe threat against IdPs. For this reason, IdPs should not allow for their login pages to be included in sub-frames of SPs (or any other external Web site). Also, a practice of including IdP login pages in frames at external Web sites is contradictory to the training of users to avoid phishing; to separate genuine IdP login pages with fakes ones, End Users should in general be presented with the URL bar of the current login page.

**Multi-Factor Authentication** Section 2.4.1 gave a theoretical introduction to the concept of multi-factor authentication, and Section 3.6 explained practical examples used in the OpenID protocol, e.g. the physical YubiKey OTP dongle. When using multi-factor authentication, it becomes a lot harder for attackers to sign on at a victim's account, even though he has knowledge of the password. However, in SSO systems, a very common scenario is that users are already authenticated with an IdP which is used to sign on at multiple SPs. In such a situation, the thesis has shown that an attacker can interact with IdP Web pages without actually stealing the user's login credentials (e.g. password) beforehand. As shown in Section 4.3.3 (*Attacking an OP Using Clickjacking*), multi-factor authentication does not help at all if an IdP is vulnerable to CSRF or Clickjacking. Multi-factor authentication is functioning as an additional security factor during the authentication with the IdP, but after the actual sign-on has been performed,

it is absolutely necessary to implement protection mechanisms against the mentioned attacks. As an example, the clavid offers an excellent way for their users to sign on using the YubiKey for extra security, but once the user is authenticated, multi-factor authentication cannot prevent attacks like CSRF and clickjacking. This underlines the importance of implementing counter-measures against such attacks.

## 5.2 Further Work

Further work for this Master's thesis might be to consider the possibility of implementing SLO functionality in the OpenID protocol. It is problematic that certain SSO systems lack this functionality, so a support for SLO would make it easier for users of the protocol. Such a work might consist in investigating ongoing efforts for OpenID SLO, and maybe a commitment to working with the creation of a specification. A first step could be to add it as an optional extension (like PAPE and SREG) to the OpenID protocol.

Another problem with the OpenID protocol is the fact that usage of HTTPS is not required during authentication. Here, a practical topic for further work could be to consider ways to check for usage of HTTPS by utilizing the PAPE extension. As mentioned in Section 3.7.1 (*Defined Authentication Policies*), the PAPE extension has pre-defined authentication policies for phishing-resistancy and multi-factor authentication. So, a topic might be to look for existing policies addressing HTTPS requirements, and implementation of an addition to the PAPE extension.

Also, if the student is familiar with Java Web development, then another possible future work could be to make an enhancement to the security framework that was used for the development in this thesis. The Spring Security framework supports usage of the OpenID protocol, but it does not support usage of the PAPE extension. Adding this is actually planned by the Spring Security community,<sup>35</sup> as can be seen at <https://jira.springsource.org/browse/SEC-1332>. Spring Security is an open source project, so anyone with interest is able to contribute with development. For access to the project, the first step would be to contact the project lead (Luke Taylor).

---

<sup>35</sup><https://jira.springsource.org/browse/SEC>



## References

- [1] Rafeeq Rehman, “*Get Ready for OpenID*”, Conformix Technologies Inc., 2008
- [2] Shreeraj Shah, “*Web 2.0 Security: Defending Ajax, RIA, and SOA*”, Course Technology, 2007, pp. 137-158 (“Cross-Site Request Forgery with Web 2.0 Applications”)
- [3] Elisa Bertino, Lorenzo D. Martino, Federica Paci, and Anna C. Squicciarini, “*Security for Web Services and Service-Oriented Architectures*”, Springer, 2010, pp. 80-82 (“Overview of Digital Identity Management”)
- [4] Jan De Clercq, “*Single Sign-On Architectures*”, Infrastructure Security (InfraSec) International Conference, Bristol, UK, 2002 pp. 40-58
- [5] Andreas Pashalidis and Chris J. Mitchell, “*A Taxonomy of Single Sign-On Systems*”, Royal Holloway, University of London, UK, 2003, <http://www.isg.rhul.ac.uk/~xrtec/cv/ssotax.pdf>
- [6] The OpenID Community, Specification: “*OpenID Authentication 2.0*” (final), OpenID Foundation, 2007, [http://openid.net/specs/openid-authentication-2\\_0.html](http://openid.net/specs/openid-authentication-2_0.html)
- [7] The OpenID Community, Specification: “*OpenID Simple Registration Extension 1.0*” (final), OpenID Foundation, 2006, [http://openid.net/specs/openid-simple-registration-extension-1\\_0.html](http://openid.net/specs/openid-simple-registration-extension-1_0.html)
- [8] The OpenID Community, Specification: “*OpenID Provider Authentication Policy Extension 1.0*” (final), OpenID Foundation, 2008, [http://openid.net/specs/openid-provider-authentication-policy-extension-1\\_0.html](http://openid.net/specs/openid-provider-authentication-policy-extension-1_0.html)
- [9] Billy Hoffman and Bryan Sullivan, “*Ajax Security*”, Addison-Wesley, 2007, pp. 75-77 (“Cross-Site Request Forgery (CSRF)” and “Phishing”)
- [10] Jeremiah Grossman, Robert Hansen, Petko Petkov, and Anton Rager, “*XSS Attacks: Cross Site Scripting Exploits and Defense*”, Syngress, 2007, pp. 93-97 (“CSRF”), pp. 238-248 (“CSFR Proof of Concepts”)
- [11] Mike Shema, “*Seven Deadliest Web Application Attacks*”, Syngress, 2010, ch. 2 (“Cross-Site Request Forgery”)

- 
- [12] Per Rynning, “*Trusler mot Single Sign-On mekanismer*” (eng: *Threats Against Single Sign-On Mechanisms*), University of Bergen, 2008, <https://bora.uib.no/handle/1956/3007>, ch. A5 (“Cross-Site Request Forgery (CSRF)”)
- [13] The OWASP Community, “*OWASP Top 10 2010 - The Ten Most Critical Web Application Security Risks*”, OWASP Foundation, 2010, <http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf>
- [14] Simon Josefsson, “*Security Evaluation of Yubico Authentication Devices*”, Datakonsult, 2007, [http://www.yubico.com/files/YubiKey\\_Security\\_Review.pdf](http://www.yubico.com/files/YubiKey_Security_Review.pdf)
- [15] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson, “*Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites*”, Stanford University, Carnegie Mellon University, 2010, <http://seclab.stanford.edu/websec/framebusting>
- [16] Alexander Lindholm, “*Security Evaluation of the OpenID Protocol*”, Royal Institute of Technology, CSC (Stockholm), 2009, [http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlister/2009/rapporter09/lindholm\\_alexander\\_09076.pdf](http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlister/2009/rapporter09/lindholm_alexander_09076.pdf)



## Appendices

### A OpenID

#### A.1 Signature Calculation in OP Response

In an authentication response returned from an OP (see Section 3.5.1 for an example), there are (amongst others) two fields called `openid.signed` and `openid.sig`. The value of `openid.signed` is a comma-separated list of signed fields [6]. The value of `openid.sig` is the result of a signature calculation of the fields defined in `openid.signed`:

```
openid.signed=ns.sreg,sreg.nickname,sreg.email,sreg.fullname,[...]
openid.sig=QA1pZ+Y9GNmr+mrwvO2wlsGTJ5eNepHEQFlibWGs1e0=
```

The value of the `openid.sig` parameter is a Hash-based Message Authentication Code (HMAC), which is encoded using Base64:

```
base64(HMAC(secret(assoc_handle), token_contents))
```

Here, `token_contents` is a string in key-value format containing all of the keys defined in the `openid.signed` parameter, listed in the same order. The parameter `assoc_handle` is the one named `openid.assoc_handle` in the authentication requests and responses. It is being used to find the HMAC key for the signature [6]. In the example shown earlier, the value looked like this:

```
openid.assoc_handle=a8843e90-4d6c-11df-a273-d702551e809e
```

This value remains the same for both the authentication request and the authentication response.

### B eXtensible Resource Descriptor Sequence

The eXtensible Resource Descriptor Sequence (XRDS) format is an XML format that can be used for discovery of metadata about a resource, in particular discovery of services offered by IdPs [1]. When IdPs receives discovery requests from SPs, XRDS documents are returned, containing the necessary information.

#### B.1 Supported Authentication Policies at VeriSign

The following shows the contents of the XRDS document at the VeriSign PIP OP for the user `jogrimst` (located at <https://pip.verisignlabs.com/user/jogrimst/yadisxrds>):

Listing 9: The contents of VeriSign PIP’s XRDS file

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xrds:XRDS xmlns:xrds="xri://$xrds"
3     xmlns:openid="http://openid.net/xmlns/1.0" xmlns="xri://$xrd*(($v*2.0))">
4   <XRD>
5     <Service priority="0">
6       <Type>http://specs.openid.net/auth/2.0/signon</Type>
7       <Type>http://openid.net/sreg/1.0</Type>
8       <Type>http://openid.net/extensions/sreg/1.1</Type>
9       <Type>http://schemas.openid.net/pape/policies/2007/06/phishing-resistant</Type>
10      <Type>http://schemas.openid.net/pape/policies/2007/06/multi-factor</Type>
11      <Type>http://schemas.openid.net/pape/policies/2007/06/multi-factor-physical</Type>
12      <URL>https://pip.verisignlabs.com/server</URL>
13      <LocalID>https://jogrimst.pip.verisignlabs.com/</LocalID>
14    </Service>
15
16    <Service priority="1">
17      <Type>http://openid.net/signon/1.1</Type>
18      <Type>http://openid.net/sreg/1.0</Type>
19      <Type>http://openid.net/extensions/sreg/1.1</Type>
20      <URL>https://pip.verisignlabs.com/server</URL>
21      <openid:Delegate>https://jogrimst.pip.verisignlabs.com/</openid:Delegate>
22    </Service>
23  </XRD>
24 </xrds:XRDS>
```

---

## C YubiKey

As explained in Section 3.6.1 *One-Time Password (OTP)*, the YubiKey<sup>36</sup> is a hardware device that can be used to obtain multi-factor authentication (described in Section 2.4.1). It looks like a small USB memory stick, but it is actually a keyboard, i.e. it can send a code as if it was typed in from a keyboard. The device can be used for various authentication purposes, but the focus in this thesis is on authentication in SSO systems, i.e. at IdPs.

Each YubiKey has its unique per-device key (also called *Device ID*). This key is the first 12 characters of the OTPs that the device generates. The following shows 10 sample OTPs generated by the YubiKey that was used during the writing of this thesis:

ccccccblteicguebdcgbdvgedihvudteiujjihfbhku

---

<sup>36</sup>The YubiKey is created by the company Yubico (<http://www.yubico.com>)

```
ccccccblteihhledenelcjjdkvtcdggfhukfnchridth  
ccccccblteirjkuatdknunkurvvfjgdhlcifgdervekc  
ccccccblteilcnejbehggngccilrjfidfhtihfdnen  
ccccccblteiigvnnbguufrcchluillnvivgfkndrnt  
ccccccblteiiuknrgucnlkvhiutcgnktrcdkiucbhj  
ccccccblteiuujffuunijvngfbgkenjgtihhihrkfcf  
ccccccblteieggjulgrtbbtecihnedldneundtvcbtt  
ccccccblteigerkjhkcdrtnvjdnkuhrduelrdbngfg  
ccccccblteiljkftcvlltjcuuvlidetethbdnggrvnt
```

In this case, we see that the unique key for this particular YubiKey device is `ccccccbltei`.

The authentication process is initiated by the user connecting the YubiKey to her computer using an available USB port. When the button is pressed, the following happens [14]:



1. A plaintext authentication token is generated.
2. The device then uses the unique key described above and encrypts the token using the Advanced Encryption Standard (AES) algorithm.
3. Next, it encodes the ciphertext with *ModHex*, which is a variant of base16 encoding.
4. The textual token is then transmitted to the computer by emulating a USB keyboard which types in the characters of the encoded token.

Figure 39 shows the control flow of these steps. The reason why ModHex (base16) encoding is used instead of regular Hex (base64) encoding is to make the YubiKey device independent of language settings in operating systems [14]. In some languages, keyboard layouts switch some keys, e.g. QWERTY on a US computer and QWERTZ on a German computer. The YubiKey uses the alphabet “cbdefghijklnrtuv” because these characters were found to be unswitched on **all** keyboards.

The plaintext token that is generated is 16 bytes (128 bits) long, i.e. the same size as one AES block. During encryption, the YubiKey uses a 128-bit

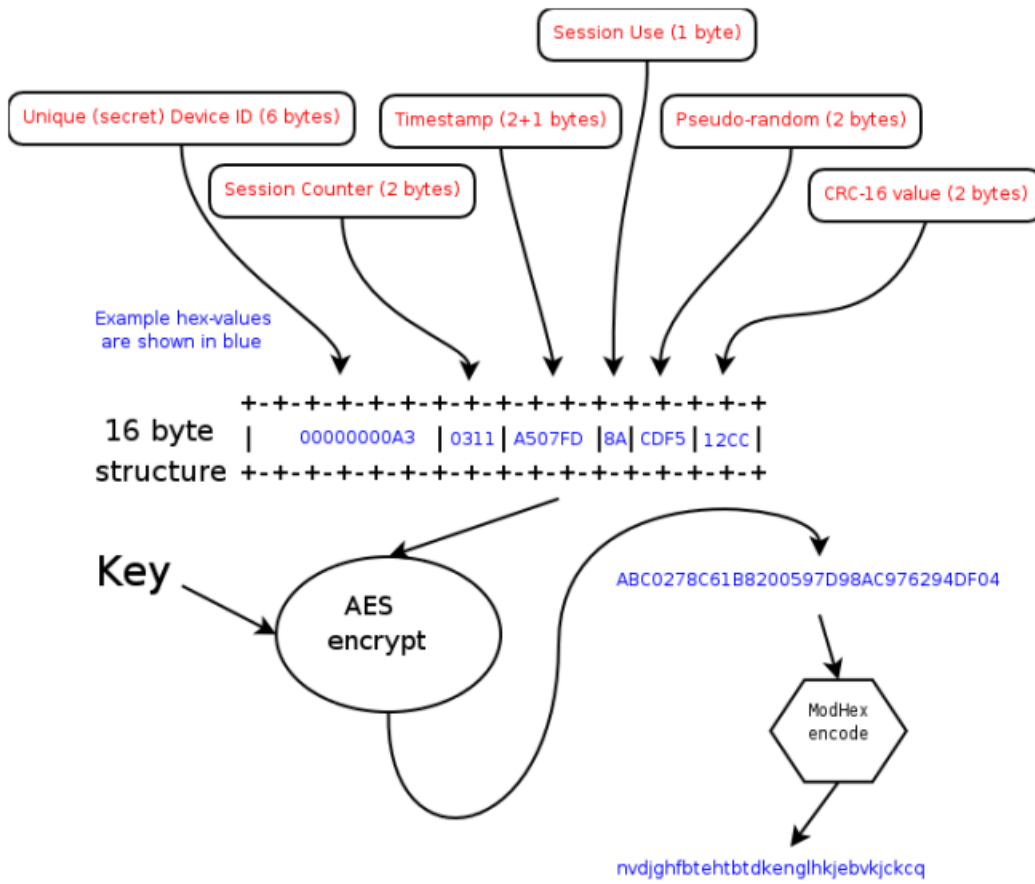


Figure 39: YubiKey control flow [14]

block length AES cipher with 128-bit keys in Electronic Codebook (ECB) mode. This gives a key space of  $2^{128}$ . The generated string is sent to a Web server or or host application for verification. This includes:

1. Convert back to a byte string.
2. Decrypt the byte string using the same (symmetric) 128-bit AES key.
3. If the checksum of the string is verified, the OTP is considered valid. If not, the OTP is rejected.

## D Cross-Site Request Forgery (CSRF)

As explained in Section 2.5.1, CSRF (also known as *XSRF* or *hostile linking*) is an attack that can happen behind the scenes of a Web browser, hidden from what is presented to the user. This can be achieved by including HTML elements like `img`, `link`, `iframe`, `script`, etc. The following shows some examples that all performs the same HTTP GET request:

Listing 10: CSRF examples

---

```
1 
3
4 <link rel="stylesheet" type="text/css" href="http://bank.com/account.php
5     ?transferTo=1234.12.1234&amount=100" />
6
7 <iframe src="http://bank.com/account.php
8     ?transferTo=1234.12.1234&amount=100"></iframe>
9
10 <script type="text/javascript" src="http://bank.com/account.php
11     ?transferTo=1234.12.1234&amount=100"></script>
```

---

### D.1 Countermeasures

As details of protection mechanisms is not the focus of this thesis, it is placed here in this appendix instead of in the main text.

To prevent CSRF, an unpredictable *token* can be used (also called *nonce*). This token can be included in the body of the HTTP request, or in the header. For the highest level of security, this token should be unique per request, but it can also be unique per user session [13]. The preferred way to include a token is to use a hidden form field, and send it as part of the body of an

HTTP POST request. This way, the value of the token does not become a part of the URL, and the probability of disclosure is smaller. Listing 11 shows an example of how a secret token value can be included in a form in order to avoid CSRF attacks:

Listing 11: Usage of unpredictable token to avoid CSRF

---

```
1 <form action="/user/change-password" method="POST">
2   <input name="token" type="hidden"
3     value="692c730d783a50127faee4543b8bcd74" />
4   New password: <input type="text" name="password1" /><br />
5   Confirm new password: <input type="text" name="password2" />
6   <input type="submit" value="Save" />
7 </form>
```

---

Now, every time this form is submitted, the server would verify that a parameter named `token` exists, and that its value is correct. This way, it is possible to differentiate between legal requests and forged requests. So, if an attacker tries to request the URL without the expected token, no harm can be made. Several ready-to-use libraries for creation of such tokens exist, e.g. the *CSRFGuard Project*<sup>37</sup> initiated by the *Open Web Application Security Project* (OWASP). This is a filter (available in Java, .NET, and PHP) that can be used to append unique request tokens to each form and link in a Web application.

Note, however, that Clickjacking can be used to get around CSRF nonce protections (see Section 2.5.2 *Clickjacking*).

## E Clickjacking

### E.1 Clickjacking Attack Against Google OP

Listing 12: Code used during Clickjacking attack against Google OP

---

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Clickjacking example (Google)</title>
5     <meta http-equiv="Content-Type"
6       content="text/html; charset=UTF-8">
7     <style>
8       iframe {
```

---

<sup>37</sup>OWASP CSRFGuard Project: <http://www.owasp.org/index.php/CSRFGuard>

```
9         position: absolute;
10        width: 740px;
11        height: 350px;
12        border: 0;
13        opacity: .5; /* 50% opacity */
14        z-index: 1; /* The stack level */
15        /* The fake page has z-index: 0 (default) */
16    }
17
18    div {
19        padding-top: 75px;
20        padding-left: 74px;
21
22    }
23    div#text {
24        width: 525px;
25    }
26    div:not(#text) {
27        /* The element containing the button is
28         * positioned directly above Google's button */
29        position: absolute;
30        top: 155px;
31    }
32    div:not(#text) input {
33        /* Match the width of the button with the
34         * width of Google's button */
35        width: 110px;
36    }
37    </style>
38 </head>
39 <body>
40     <iframe src="https://localhost:8443/sso-webapp/
41         j_spring_openid_security_check?openid_identifier=
42         https://www.google.com/profiles/jogrimst">
43     </iframe>
44     <div id="text">
45         Lorem ipsum dolor sit amet, consectetur adipisicing
46         elit, sed do eiusmod tempor incididunt ut labore et
47         dolore magna aliqua. Ut enim ad minim veniam, quis
48         nostrud exercitation ullamco laboris nisi ut aliquip
49         ex ea commodo consequat. Duis aute irure dolor in
50         reprehenderit in voluptate velit esse cillum dolore
51         eu fugiat nulla pariatur. Excepteur sint occaecat
```

```
52         cupidatat non proident, sunt in culpa qui officia
53         deserunt mollit anim id est laborum. ...
54     </div>
55     <div>
56         <input type="submit" value="Click here" />
57         to view the rest of the text.
58     </div>
59 </body>
60 </html>
```

---

## E.2 Protection Mechanisms

As indicated in Section 4.3.4 *Countermeasures Against Clickjacking*, it is possible to use an HTTP header named `X-FRAME-OPTIONS` in order to indicate to browsers that certain pages are not allowed to be displayed inside `iframes`. Figure 40 shows an example where the Google OP login page is requested. When its value is set to `SAMEORIGIN`, it is not possible for Web sites to include

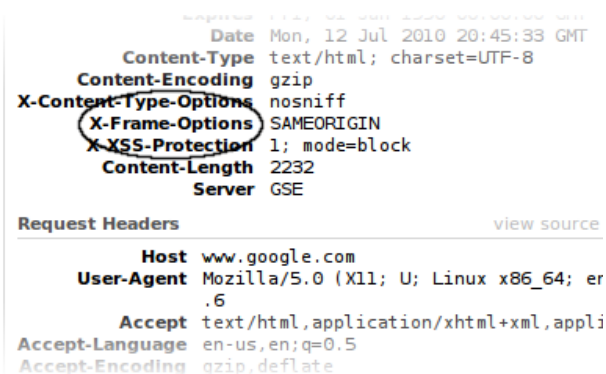


Figure 40: The HTTP `X-FRAME-OPTIONS` header

this page in a frame unless the Web site is at the same origin as the included page. An alternative value of the header is `DENY`, which would never allow the content to be rendered within a frame, even not at the same domain [11].

The `X-FRAME-OPTIONS` header was introduced by Microsoft in version 8 of Internet Explorer [11, 15]. The header is supported by today's versions of the browsers Safari, Chrome, Opera, and IE. Firefox does not support it as of today, but the future version (Firefox 4) will support it [15].



## F Web Application

### F.1 Log Messages During Sign-On

Log messages produced during the sign-on example described in Section 4.2.3:

Listing 13: Logging produced during sign-on

---

```
1 [INFO,Discovery] Starting discovery on URL
2   identifier: http://me.yahoo.com/jogrimst
3 [INFO,YadisResolver] Yadis discovered 1 endpoints from:
4   http://me.yahoo.com/jogrimst
5 [INFO,Discovery] Discovered 1 OpenID endpoints.
6 [INFO,ConsumerManager] Trying to associate with
7   https://open.login.yahooapis.com/openid/op/auth
8   attempts left: 4
9 [WARN\HttpMethodBase] Going to buffer response body of
10  large or unknown size. Using getResponseBodyAsStream
11  instead is recommended.
12 [INFO,ConsumerManager] Associated with
13   https://open.login.yahooapis.com/openid/op/auth handle:
14   Ng.45DcOvbL23FITFHpFL9v51cZy4buHOqj1EGmf0Zt6IHE7j [...]
15 [INFO,ConsumerManager] Creating authentication request for OP-
16  endpoint: https://open.login.yahooapis.com/openid/op/auth
17  claimedID: https://me.yahoo.com/jogrimst
18  OP-specific ID: https://me.yahoo.com/jogrimst
19 [INFO,RealmVerifier] Return URL:
20   https://localhost:8443/sso-webapp/j_spring_openid_security_check
21   matches realm: https://localhost:8443/
22 [INFO,ConsumerManager] Verifying authentication response...
23 [INFO,ConsumerManager] Received positive auth response.
24 [INFO,ConsumerManager] Found association: Ng.45DcOvbL2
25   3FITFHpFL9v51cZy4buHOqj1EGmf0Zt [...]
26   verifying signature locally...
27 [INFO,ConsumerManager] Verification succeeded for:
28   https://me.yahoo.com/jogrimst#16cb4
```

---

### F.2 Attached Source Code

The source code of the SSO Web application that was developed as part of this thesis has been attached in a ZIP file. If desired, it is possible to run this application locally on a computer. This, however, requires that certain software is installed:

- Java
- Maven

The ZIP file contains a folder named `sso-webapp`. In order to run the application, unzip the file, and navigate into this folder. The folder contains a file named `README.txt`. This file contains the commands that need to be executed in order to start the application. Commands for both Windows and Linux are provided.