CHALMERS

Developing a Single Sign-On System

A Java-based authentication platform aimed at the web.

Master of Science Thesis in Software Engineering

HENRIK JERNEVAD

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Göteborg, Sweden, March 2009

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Developing a Single Sign-On System
A Java-based authentication platform aimed at the web.

HENRIK JERNEVAD

© Henrik Jernevad, March 2009.

Examiner: Andrei Sabelfeld

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2009

Abstract

A typical computer user today, spends a lot of her time on the Web. As a part of this, she often needs to type her username and password at a dozen different sites or more every day. To cope with this, users typically choose simple passwords or reuse a few ones. This lowers the security of the system and increases the risk of an attacker being able to compromise the user's account(s).

The goal of this thesis is to build a so called single sign-on system which solves these problems. The result is NaviBase, a system based on the Java technology stack, which uses the Security Assertion Markup Language to provide single sign-on services to applications and users.

The system consists of two primary components; NaviBase, the server component which holds all information and processes requests; and SamlLib, a slimmed-down implementation of the SAML protocol.

In retrospect, a focus on sound development principles and using well known design patterns proved successful and preliminary security auditing suggest the system is sufficiently secure. On the flip side, much time was spent on unplanned activities and the system is somewhat hurt by a lack of focus on usability.

Sammanfattning

En typisk datoranvändare idag tillbringar mycket av sin tid på webben. Som en del av detta behöver hon ofta skriva in sitt användarnamn och lösenord på dussintals sidor varje dag. För att orka med detta använder användare vanligtvis enkla lösenord eller återanvänder ett fåtal. Detta sänker säkerheten i systemet och ökar risken för en attack som äventyrar användarens konton.

Målet med detta arbete är att bygga ett så kallat ”single sign-on”-system som löser dessa problem. Resultatet är NaviBase, ett system baserat på Java-teknikplattformen, som använder Security Assertion Markup Language för att tillhandahålla ”single sign-on”-tjänster till applikationer och användare.

Systemet består av två huvudsakliga komponenter; NaviBase, en serverkomponent som håller all information och bearbetar förfrågningar; samt SamlLib, en nedbantad implementation av SAML-protokollet.

I efterhand visade sig en fokus på sunda utvecklingsprinciper och välkända design-mönster vara framgångrik och en preliminär säkerhetsgranskning antyder att systemet är tillräckligt säkert. Dock spenderades mycket tid på oplanerade aktiviteter och systemet är något lidande av en avsaknad av fokus på användbarhet.

Preface

This report was performed as a Master of Science Thesis at Chalmers University of Technology. It covers the development of a software system dealing with authentication and security. My examiner at Chalmers was Andrei Sabelfeld, Associate Professor at the Department of Computer Science and Engineering.

The system was developed for MindValue AB, a company which specializes in developing software for interaction, knowledge management and business. This includes software such as communities and Content Management Systems.

The system as well as any documentation is available in the form of demonstration by the author.

Table of Contents

1 Introduction

The introduction describes the problem discussed in this report and motivates why it is important. It further describes the specific objectives for the solution described in this report. Finally, the limitations on the project and report as well as earlier attempts at solving the problem are discussed.

1.1 The Problem

A typical computer user today, spends a lot of her time on the Web. Anything from entertainment such as games , and videos and picture sharing, to private economy with banking and income-tax return forms are available online, or social web pages such as communities and chats. Then of course, there are email clients, and even more traditional productivity software such as word processors and spreadsheet programs seems to be moving online. Ever more data is stored in "the cloud" rather than on the user's own computer. You can access anything from anywhere.

Most of these different sites, services, and applications require some kind of membership. You need to prove who you are, so the system knows what resources you have access to. Typically today, you gain access to a site by the means of a username and password. For the user, this means that during a typical day, she might have to type in her username and password at a dozen different sites or more. This problem is referred to as *password fatigue* (Wikipedia 2007).

To achieve good security, a unique password should be chosen for every site the user becomes a member at. However, because a typical user finds it very hard to remember a lot of random sequences of characters, users tend to reuse the same password at many different sites. This means that the load on the user's memory becomes lower. However, security also gets lower. A malicious site owner, or an attacker breaking in to a site, could get hold of the user's password. If they did, the security of all of the user's other sites would have been compromised too.

Thus, to summarize, we really have two problems. One is the fact that today, a user typically has to authenticate a lot of times per day. Secondly, because of how often the user is required to authenticate, the user often chooses passwords which are cryptographically weak.

One possible solution to these problems is for the user to have a unique password for every site, and let her computer store the passwords in a special password store. This password store could then be protected by a single cryptographically strong password. The user would get unique and strong passwords, but only have to remember one and specify it only every once in a while.

An attacker breaking into one of the sites in question only can gain access to the user's account on that site. In order to get access to the user's accounts on other sites, the attacker needs to break into the password store. That in turn requires the user to first gain access to the user's computer, and then also break into the password store. If the attacker can do this, there are probably even worse things she can do.

The downside of this approach is that since the password store is now located on a specific computer, the user might not be able to access a site from another computer. This is a major drawback with this solution. The solution to this problem is to move the password store away from

the user's computer to a special server accessible from anywhere the user might want to access it. Such a special server is called a *single sign-on* system. (Wikipedia 2009)

1.2 Objectives

The goal of the work described in this report is to solve the two problems mentioned above. That is, the objective is to build a sign-on system which saves the user from repeating her authentication credentials many times during a short period of time, and increases the security by stopping an attacker which breaks in to one site from being able to access a user's account on other sites.

Thus, we have two primary goals for the system. It should be easy to use and at the same time be secure. As these two goals are often conflicting with each other, prioritizing is needed among different possible solutions.

1.3 Limitations

While the system has two goals, usability and security, the primary focus of this report is on security. Description and discussion of analysis and results is from a technological and security viewpoint. A number of aspects of the system such as visual presentation are not given as much space unless they directly affect security (which they sometimes do).

2 Theory

This section describes the “technological landscape” for the work and the background theory needed to understand the report. It covers diverse topics ranging from development best practices and design patterns to digital signatures and hashing algorithms. Things that the reader is supposed to be familiar with (e.g. basic data structures and algorithms) or where detailed knowledge isn’t required (e.g. RSA encryption) are not mentioned, or mentioned very briefly.

2.1 General Software Security

This section describes theory regarding general security concepts covered by this thesis. Basic knowledge about some concepts including encryption, digital signatures, and hashing algorithms, is assumed. Thus, these topics are only covered if further details are required for understanding this work.

2.1.1 Authentication vs. Authorization

Authorization and authentication, while very similar in name and often used in combination, are really two quite different things.

Authentication, commonly abbreviated “authn”, is about identifying who someone is. This can be done in many different ways, depending on the context. In “real life”, you typically authenticate by showing your photo ID, or perhaps writing your signature. On the web, the without doubt most common way of doing this is through a username and password. However, many other options are available, such as various kinds of two-factor authentication.

As a second step, after authentication, comes authorization (often abbreviated “authz”). *Authorization* is the process of deciding whether a specific user has rights to access a specific resource. Thus, you can be authenticated but still not authorized.

2.1.2 Password Security

The purpose of a password is to be a secret combination of characters known to only one or a few people authorized to access a resource. Thus, it should not be a sequence easily guessed by others, such as a name, phone number, birth date or other information with some connection to real world entities. Instead a more or less randomly chosen sequence is typically used.

Excluding attacks such as tricking someone to reveal a password, eavesdropping, or other types of “social” techniques, the typical way of cracking a password resolves to some form of brute force search. That is, an attacker repeatedly tries possible combinations until the right one is found.

In order to make it as difficult as possible for the attacker to figure the password out, we want to force the attacker to search through an as large set of possible passwords as possible. Increasing the password space can be done in two ways; having a long password, and include characters from an as large set of possible characters as possible. For example, a password with 5 characters is easier to crack than one with 10, and a password with only letters is easier than one with both letters and numbers.

Table 1 displays a few examples of how long time it would take to search through the full set of possible passwords, given the parameters password length, and number of possible characters. We assume that a computer can try 1,5 million passwords per second. (PlayStation a hacker's dream 2007) Of course, if we had access to a 1000 computers, the times could be divided by roughly as much.

	4 characters	**8 characters**	**16 characters**
Numbers only (10)	<1 second	1 minute	210 years
Letters only (50)	4 seconds	10 months	3,2*10^13 years
Letters and numbers (60)	9 seconds	3,5 years	6,0*10^14 years
Letters, numbers, and special characters (75)	21 seconds	21 years	2,1*10^16 years

TABLE 1: TIME TO CRACK PASSWORDS

Password storage

Again, excluding attacks where the attacker gets hold of the password unencrypted, a typical brute attack is performed against the password storage. Obviously, an attacker shouldn't be allowed to access it, but it can still happen. In fact, early UNIX implementations allowed anyone to read the encrypted password store assuming it would be safe. This assumption was made based on the vast computing power required to crack a password. However, this assumption no longer holds given modern computer equipment.

Because of the vulnerability of short passwords, various techniques are used to improve their security when stored. Using a salt is the most typical solution. A randomly generated string of characters is appended to the password before encryption. This also helps in that two identical passwords are hashed into two different values, because while the passwords are the same their salts are not.

2.1.3 Two-Factor Authentication

The primary way of authenticating on the Internet today is using a username and password. For most uses this is a solution which provides adequate security, especially given a cryptographically strong password.

However, no matter how secure the password is, if an attacker gets hold of it, he can access the resource it was protecting. For high-security applications, this is not acceptable. There is thus a need for a higher level of security. One way to achieve this is through so called *two-factory authentication* where the user needs to provide not only one security credential (e.g. a password), but two. This raises the bar as the attacker needs to get hold of both these security credentials.

For two-factor authentication to be truly effective, two different kinds of credentials should be used. A password is something you *know*. Thus, the second credential should be something else, for example something you *have* or something you *are*. (Viega and McGraw 2002)

Also using something you *have* for authentication makes it much harder for an attacker to compromise the security of the system. Not only does the attacker need to get hold of the user's password, but they also need to physically get hold of something the user is in possession of.

There are many kinds of credentials which are based on something you *have*. The most common is perhaps a smart card, a pocket-sized card with embedded integrated circuits which in combination with a card reader can store and process a digital certificate used to authenticate the user carrying it. Another common type of 'something you have' is a mobile phone. Much like a smart card, it can carry a digital certificate. A third common type is *one time password* token which generates a pseudo-random number that change at pre-determined intervals.

What provides an arguably even higher level of security is requiring authentication based on something you *are*. This could be a fingerprint scan, a retina scan, or any other kind of biometric. In order to break this kind of authentication, the attacker needs to either get hold of an actual part of the user's body(!), force the user to authenticate, or somehow be able to fool the biometric scanner. While the first and second alternatives can obviously be done, it comes with a much greater risk for the attacker. Thus, the most likely type of attack is fooling the sensor. How hard this is depends on what biometric is used, and the quality of the sensor. Especially some fingerprint scanners have been proven to be quite easy to fool, while other scanners have proved to be very reliable.

2.1.4 Attack Trees

An *attack tree* is a conceptual graph for representing threats and possible attacks to a computer system, suggested by (Schneier 1999). They are derived from ”fault trees” in software safety. An attack tree is structured to correspond to of the decision-making process of an attacker.

Structure of an Attack Tree

Potential goals that an attacker wants to achieve acts as root nodes for one or more trees. First level nodes under the root nodes correspond to high-level ways in which a goal could be achieved. The leaves of the trees represent the details of the different ways of achieving these goals. The lower in the tree you go, the more specific the attacks get.

Given a complete attack tree, one can make it more useful by annotating its nodes with values representing the perceived risk of that attack. This includes estimating how feasible the attack is in terms of time (effort), cost, and risk to the attacker.

Building an attack tree

The first step of constructing an attack tree is to identify the data and resources of a system that may be targeted in an attack. These targets become the root nodes in the attack tree. After that, all components, communication channels between the components, and all the types of users of the system, are considered. Together, these tend to include the most likely failure points (Viega and McGraw 2002).

Furthermore, not only the software developed in-house is included, but also any components used by the software developed elsewhere. Also included in the analysis are the computers on which the software runs, the network which they use, etcetera.

2.2 Single Sign-On

To recap from the introduction, a single sign-on system builds on the notion that one special server holds the responsibility to authenticate users to a number of different sites or services. To put it from the user's perspective, a user can authenticate once to one single server, and then gain access to multiple sites.

In the terms of SAML (Security Assertion Markup Language, see details below), a *service provider* is a site which provides some functionality or service to a user. This could be a webmail client, a newspaper, an online banking system, or just about any site on the Web. The special server responsible for authenticating the user to these service providers is called an *identity provider*. (OASIS 2006)

To provide a typical scenario of how single sign-on systems usually work, we look at a user who wants to access a certain web page (service provider). In this case, the service provider requires the user to authenticate in order to function properly. The service provider asks the identity provider to authenticate the user in question through a request which is (at least in theory) completely transparent to the user.

It is then up to the identity provider to perform the actual authentication. If the identity provider does not currently know who the user is – i.e. there is no session established between them – it is forced to ask the user to provide some suitable security credentials (typically username and password). Given correct security credentials, a session can be established, and the identity provider can return the identity of the user to the service provider. The service provider can then continue serving the user the requested resource.

However, if another service provider recently has asked about the identity of the user, the identity provider already has a session established for that user. In that case, there is no reason to ask the user for security credentials, and the identity provider can just return the identity of the user to the service provider without bothering the user at all. This is the main benefit from single sign-on from a usability perspective.

2.3 Security Assertion Markup Language (SAML)

SAML stands for *Security Assertion Markup Language*. In essence, it is an XML-based framework for asking questions and making assertions about the authentication and authorization of users between security domains. The standard is developed by the Security Services Technical Committee of the Organization for the Advancement of Structured Information Standards, abbreviated OASIS. (OASIS 2005)

SAML consists primarily of four main components; the Core, Protocols, Profiles, and Bindings specifications. Each of these is described in further detail in the following chapters. While there are a few more components in SAML, they are out of scope for this thesis and are thus not covered.

2.3.1 History

Since its first version, 1.0, SAML has gone through one minor and one major update. These are versions 1.1 and 2.0, respectively. The following sections provide an overview of all three versions of SAML.

SAML v1.0

SAML was put together as an effort to "to define an XML framework for exchanging authentication and authorization information." (Maler 2001) The results of a number of earlier related projects were contributed, and after almost two years, in November 2002, SAML version 1.0 was adopted as an OASIS standard. (OASIS 2007)

It aims to define a data format for authentication assertions, as well as authorization attributes in a secure fashion. The following are the main scenarios for which it is developed.

- Single Sign-On over the Web – a web user after authenticating with a web site can access secured resources at another web site, without directly authenticating to that web site.
- Authorization Service – one business entity can ask another entity to make authorization decision on its behalf.
- User Session – two applications can share a common user session.

SAML v1.1

Work continued, and version 1.1 of SAML was ratified as an OASIS standard in September 2003. It is a minor update to 1.0 and contains only smaller reorganizations, improvements and a few deprecations. It is today widely implemented and deployed.

SAML v2.0

A major overhaul of the standard came in March 2005, when SAML version 2.0 was standardized. It represents a significant upgrade in terms of features, compared to version 1.1. The enhancements came from not only normal feature request, but also from the Liberty Alliance Identity Federation Framework (ID-FF) V1.2 specification that was contributed to the standards committee in 2003, and features in Internet2's Shibboleth architecture. The new version breaks backwards compatibility with version 1.0. (trscavo@idp.protectnetwork.org 2007)

Many new features are introduced in SAML v2.0. Following is a list of the new features introduced which are of interest for this thesis. (OASIS 2005)

- Pseudonyms –pseudo-random identifiers with no discernible correspondence with any meaningful identifiers such as username or email.
- Identifier management –two providers can establish and manage the pseudonyms for the principals for whom they are operating.
- Session management – a protocol by which all sessions provided by a particular session authority can be near-simultaneously terminated.

2.3.2 Core Specifications

This and the subsequent sections specifically describe SAML v2.0, but the basic structure is the same in earlier version as well although specific things may have been renamed or restructured.

At the very heart of SAML v2.0 lays the definitions of the messages that can be sent between various entities. These are divided into two sub categories, the *assertions* which contain actual security assertions that we want to communicate, and the *protocols* which contain the messages needed to carry the assertions. (OASIS 2006)

Assertions

The assertions part of SAML is the very essence of what SAML is about. They describe one entities assertion to another about a user. We could call them the nouns of the SAML language.

SAML defines three types of assertions: *Authentication* dealing with who a subject is, *Attribute* which is about specifying information about subjects, and *Authorization Decision* handling the question of what the subject is allowed to do. Shared among these assertion message types is a set of elements which formalize concepts such as ids, names, subjects, conditions, encryption, and advice.

Figure 1 contains an example of a typical authentication assertion. To go through it from the top to the bottom, the assertion was issued January 31st 2008 by `navibase.com`. It provides an assertion about a user with e-mail address `h.jernevad@example.com` and is valid within a span of 24 hours. The assertion also tells that the user has a global session with index 67775277772. This information can be used later in order to achieve single sign-out. Finally, the assertion tells us that the user was authenticated through the means of a password sent over a protected transport, such as TSL.

```
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    Version="2.0"
    IssueInstant="2008-01-31T12:00:00Z">
  <saml:Issuer>
    navibase.com
  </saml:Issuer>
  <saml:Subject>
    <saml:NameID
        Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
      h.jernevad@example.com
    </saml:NameID>
  </saml:Subject>
  <saml:Conditions
      NotBefore="2008-01-31T12:00:00Z"
      NotOnOrAfter="2008-02-1T12:00:00Z">
  </saml:Conditions>
  <saml:AuthnStatement
      AuthnInstant="2008-01-31T12:00:00Z"
      SessionIndex="67775277772">
    <saml:AuthnContext>
      <saml:AuthnContextClassRef>
urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
      </saml:AuthnContextClassRef>
    </saml:AuthnContext>
  </saml:AuthnStatement>
</saml:Assertion>
```

FIGURE 1: EXAMPLE OF SAML ASSERTION

Protocols

The other part of the SAML core is the *protocols.* These are the verbs of the SAML language, so to speak. Although there are a number of different messages, this section covers only the most basic ones, which are needed in order to make a working SSO system. The messages are divided into two broad categories, *requests* and *responses*.

The most typical request is the `AuthnRequest` by which a Service Provider (SP) can ask an Identity Provider (IdP) to issue assertions about a specific user. The Identity Provider then returns a `Response` message, containing either the requested assertions or a failure response. Such a response can also be sent unsolicited by an Identity Provider. Thus, the process can be initiated by either the SP or the IdP.

Another common request and response pair is `ManageNameIDRequest` and `ManageNameIDResponse`. A Service Provider can use these to request that the Identity Provider uses a specific id (number, name, e-mail, etc) to identify a subject. This id is then only used for this SP. It can also tell the Identity Provider to terminate a given identifier, thus "deleting" that user.

Yet other requests can be used to achieve near-simultaneous logout of a collection of related sessions ("single logout") or request a name identifier mapping between multiple service providers.

2.3.3 Bindings

Where the assertions and protocols specify *what* to send, the bindings tell *how* to. They map SAML request-response message exchanges onto standard messaging or communication protocols. The bindings specify exactly how the messages should be encoded, how to respond in various error cases, and more. The SAML standard specifies a set of common bindings. These include bindings for e.g. SOAP, HTTP Redirect, and HTTP POST.

All SAML protocols allow the requester to attach a so called *relay state* to a sent request. This is an arbitrary string which the responder must include unaltered in the response. It can therefore be used by the requester to keep state over the message exchange even if it has to give up control over the user agent.

HTTP Redirect and HTTP POST

For this thesis, HTTP Redirect and HTTP POST are of special interest. They are similar in the regard that they both operate over the HTTP protocol, and both are intended for cases in which the SAML requester and responder need to communicate using an HTTP user agent as an intermediary. This is typically needed when the responder (Identity Provider) needs to interact with the user in order to authenticate her.

In HTTP Redirect a party sends a message to another party by returning a HTTP redirect to the user agent, directing it to the other party's "consumer" url which includes the actual request message and attached relay state as parameters. HTTP POST works in the same way, but an auto-submitting (through JavaScript) HTTP form is used instead of HTTP redirect. This allows for larger messages to be transferred.

2.3.4 Profiles

The last part of the equation is *profiles* which define different uses of the SAML assertions, messages and protocols in order to achieve some specific goal. Examples include the *Web Browser SSO* profile which a users whishes to request a protected resource over the internet, or the *Single Logout* profile where a authenticated user whishes to not only log out from the identity provider, but also from all service providers to which she may be authenticated.

In this thesis, the Web Browser SSO profile is of primary interest.

2.3.5 Implementations

At the time when this thesis work was started, there only existed a single major implementation of the SAML specification, namely *OpenSAML* produced by the *Shibboleth* team. OpenSAML is a set of open source libraries, implemented in both C++ and Java, which implements the SAML specification. (OpenSAML 2008)

The first version, available when this thesis work started, was OpenSAML 1 which provides support for SAML 1.0 and 1.1. OpenSAML 2, a re-rewrite of OpenSAML 1 which also supports 2.0 was under work but not yet finished.

2.4 Software Design and Methodology

This section describes a few concepts in software design and methodology which are used or discussed in later sections.

2.4.1 Model View Controller (MVC)

The major design pattern underlying many, if not most, web applications is the *Model View Controller* pattern (Reenskaug 1979). It suggests that a system is divided into three main blocks of code; the model, the view, and the controller.

The *model* is the core of the system which holds state and all business related functionality. It knows everything about the business domain such as what entities and concepts there are and what operations are available. It holds all the data and is responsible for its integrity.

A *view* is responsible for displaying a subset of the model to a user of the system. This could be anything from displaying a graphical user interface to a human user, an application programming interface to another application, or any other type of interface.

The *controller* is the glue of the system, so to speak. It is responsible for putting the model and the view together into an actual application. It contains all the wiring for the application and knows how to map user requests into functionality. If model and views know “how” to do something, the controller knows “what”, “why”, and “when”.

2.4.2 Don't repeat yourself (DRY)

Don't Repeat Yourself, or *DRY*, is a design principle which is both very basic and simple but at the same time very powerful and applicable almost everywhere. To quote Hunt and Thomas from their book The Pragmatic Programmer, it is the notion that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system". Put otherwise, it's a way of saying "remove duplication". (Hunt och Thomas 1999)

The primary benefit of this principle is that code gets easier to read, understand, and maintain. It helps us avoid problems which come from copy 'n paste-programming where you copy a section of code, find a bug in it, correct it but forget to also correct it where you copied the code from. That is great in itself, but it also gives a number of secondary benefits. For example, consider security. Not only is code that is easier to understand also easier to keep free of security vulnerabilities, but applying the DRY principle, you often get security related code in fewer places and thus better opportunities to use patterns such as Barricade and Choke Point (discussed below).

2.4.3 Barricade and Choke Point

A useful defensive programming technique described by McConnell in Code Complete is called *barricade*. It is the notion that you define some parts of the software that work with dirty data and some that work with clean data. The latter parts, which typically are a majority, can be relieved of the responsibility for checking for bad data. Instead, you only check data crossing the boundary (the barricade) for validity. Inside its safe, but outside all bets are off. (McConnell 2004)

Choke point is a term which is derived from military strategy, where it is a geographical feature where an opposite force is forced to pass, typically on a narrow front. It could conceptually be described as a funnel. In software development, it refers to a piece of code through which every call or bit of data of a certain type has to pass. This could be e.g. a method through which all user data has to go in order to be validated, or all SQL queries in order to be properly escaped. A choke point has much higher value if it is not possible, or at least very hard, to pass through it in an unwanted way. (Wikipedia 2009)

These two techniques can be used together in a very natural way. A barricade keeps dirty data out of the clean parts of the system, and is only let in through a choke point at which it can be cleaned appropriately.

2.4.4 Test coverage

Test coverage is a measurement of how thorough a set of test cases actually verifies that some code works as intended. It is typically measured in per cent, and describes how many lines of code are exercised by at least one test case or how many of the possible program states or code branches that are tested.

There are a number of theoretical ways of discussing this metric, and ways of deciding how many (and which) test cases are needed in order to fully exercise the code under test. For example, *Cyclomatic complexity* is a method of measuring complexity in code, developed by Tom McCabe. It directly measures the number of linearly independent paths through a program's source code. While it is often used as a number on how complex some piece of code is, it also is an upper bound for the number of test cases that are necessary to achieve complete branch coverage. (McCabe 1976)

2.4.5 Simplicity

To some degree, programming is just mechanical work. We know what we want the computer to do, and we just need to write down the instructions. However, although we normally know what we want the computer to do, we don't necessarily know the best way to do it. Figuring out a good way to do this is what programming is all about.

Unfortunately, the simplest solution is not necessarily the first one that comes into our minds. The saying goes that "less is more", and this is very true when it comes to software design. By keeping our system as simple as we can we gain one very important quality; it makes our system easier to understand. This can all be summed up very nicely by a quote by Hoare. (Hoare 1981)

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*

While this is less tangible than many other principles or guidelines, it is nevertheless very important.

2.5 Frameworks and Libraries

A number of existing frameworks are available which help to speed up development and let developers focus on the actual issues of their particular solution rather than the infrastructure.

2.5.1 Apache Tomcat

Apache Tomcat is a platform for developing and deploying web applications and web services which implements the Java Servlet and Java Server Pages technologies. It is one of many available implementations of these standards, and also one several available under an open source license. (Apache Software Foundation 1999)

It provides an object model for basic HTTP-based communication which web applications can use as a stable and scalable foundation. Primary concepts here are requests and responses, which correspond to the concepts in HTTP traffic with the same names. Among many other concepts, one noteworthy is that of filters. A *filter* dynamically intercepts requests and responses going in and out of the application in order to transform or use the information they contain. These are typically used for "auxiliary" functions such as authorization and logging.

2.5.2 Apache Struts

Apache Struts is an open source framework for building Java web applications based on the Model-View-Controller (MVC) design paradigm. It runs on top of a Java Servlet-based web application server. (Apache Software Foundation 2000)

The Struts web framework rests on the concept of "actions" and "forwards". An *action* is a piece of code designated to handle a single HTTP request from the user such as "get page A" or "post data to page B". Based on pre-defined "wiring", an appropriate action is selected and run for each incoming request. An action processes the incoming request and its data, modifies the model if necessary, and generates a so called forward. A *forward* is simply a "link" to what view or other action the user should be redirected based on as a response to the request.

Apache Tiles

To make the creation views easier, Struts incorporates library called *Tiles* which provides us with the two related concepts of layouts and tiles. A *layout* is a template page, which holds the parts which are common to more than one page. For example, all web pages in the same user interface can typically use one layout, while rendering XML messages requires a completely different layout. A *tile* is an interface component which may be used more than one time and is thus broken out into a separate unit. Tiles are a clear way of incorporating the DRY principle in view building. Wiring various tiles and layouts together is made through an external declarative configuration file. (Apache Software Foundation 2001)

2.5.3 Hibernate

Hibernate is an open source object-relational mapping (ORM) framework for Java. It allows developers to work with the (object) model and takes care of the details of storing and retrieving this model to and from a relational database. (Red Hat Middleware 2006)

3 Method

This section describes the division of work for this thesis. It describes how the problem is analyzed, data gathered, choice of methodology, and so on. Assumptions regarding choice of model and method are explained and motivated.

Work is divided into two major parts, planning and execution. Planning handles the analysis work performed before the system is created. Execution covers the actual implementation work.

3.1 Planning

The work performed during the planning phase is performed at the beginning, aimed at creating a solid foundation for the following execution. Many questions need answers, especially regarding "what" and "why". The results of this planning are presented in the Analysis chapter.

3.1.1 Use cases and requirements

The very first step needed in order to figure out what system to build is collecting use cases and requirements. For this thesis, two primary methods will be used, interviews and attack trees. These two also correspond to the two primary goals of the work, achieving both great usability and security.

Interviews collect the user-centered requirements such as requested functionality, usability features, and much more. Also, interviews provide many non-functional requirements regarding things such as performance and reliability.

Attack trees are a complement which provides a more security-based angle. Major areas of concern regarding security are identified by looking at potential ways an attacker could compromise the system as well as what measures of protection could be used to guard against them.

3.1.2 An iterative process

Secondly, with a huge number of use cases and requirements, much work is required in choosing what and in which order to build it. Here, a highly iterative process will be used. At regular intervals and in concert with stakeholders of the system, a batch of the currently most interesting features is selected for implementation. When this batch is finished, the process is repeated. This procedure results in a very flexible way of working which is highly adaptable to change in requirements and needs of the stakeholders. The features which receive the lower priority are naturally done last or even left out of the work in order to limit its scope.

While the design of the system naturally evolves as the system grows, decisions on architecture, platform and more are decided in advance. The reason for this is to achieve a solid foundation for the system to be built upon.

3.2 Execution

This section describes how everything after the planning is done, the actual detail design and implementation of the system. Whereas the planning is more about "what" and "why", this section is about "how". The results of the execution are presented in the Result chapter.

As described briefly in the previous section, all execution work will be performed iteratively. That means that all phases of development, including design, coding, testing and so on, are performed over and over again, in small chunks. These cycles, called iterations, typically last for two weeks. That means that first two weeks worth of development work are chosen and performed. Then another two weeks of development follows, and so on.

3.2.1 Domain Modeling

During development, much time is spent on designing the system. This work can be divided into two major parts; domain modeling and system design. Domain modeling includes deciding what concepts or entities exist in the system, how they relate to each other, and finally how they are represented in code. This domain model can also easily be translated into a database schema. The domain model will be based on the concepts identified through stakeholder interviews.

The other part of the design work is system design. This is focused on how the various blocks of code needed to make the system run should be structured. That includes objects related to domain, encryption, network communication, web controllers and much more. These objects and their classes are divided into various packages, grouping them together logically. System design for this work will rely on the guiding principles mentioned in the Theory section.

3.2.2 Test-Driven Development

To ensure, or at least increase the likelihood of, working code a technique called test-driven development (abbreviated TDD) will be used. This practice means that an automated test displaying the lack of the wanted functionality is always written before the functionality itself. As mentioned in the Theory section, it has two noticeable features in that it forces using an interface before designing it and it increases the test coverage.

Most TDD advocates claim it is not about tests, but about design. This is because of the first noticeable feature, forcing use before design. While this may sound strange, it is in fact a very powerful practice. Because a test should be written before the functionality is implemented, that test in fact has to be written without a fixed interface for the new functionality. This means that while writing the test, you are in fact also designing that interface. However, you do it with a focus of *using* the interface, rather than what might be typical, that you design the interface based the technology used to implement the feature. This helps getting a usable and practical interface.

The second feature of TDD is that it naturally creates many test cases. While many more tests cases might need to be created in order to reach full test coverage, it creates a solid base for testing. It also helps keeping testing in focus during development. Given the high-security nature of the system, having a large set of automated tests covering as much of the system as possible is very important. But arguably more important is that not only are the tests verifying the system, they do so *automatically*, that is, without requiring human interaction.

4 Analysis

The analysis chapter describes the problem in further detail, what decisions were made and on what grounds these decisions were made.

As noted in the Method section, most of the information available comes through interviews with various stakeholders. Interview results give us three major parts; the concepts of the system, the functional requirements, and the non-functional requirements. The results of the interviews are then complemented through the use of attack trees. Each of these results are discussed below.

4.1 Concepts

The first result of the stakeholder interviews is a description of the concepts in the system, or the *domain model* of the system. This includes users, applications, bookmarks, and much more, as described below. This model also becomes the base for both the object model which is used internally in the system as well as for the relational model used in the database.

First, a graphical representation of the concepts, or the domain, is provided (please refer to Figure 2: An overview of the concepts). It is followed by a description of each concept and it's relation to other concepts.

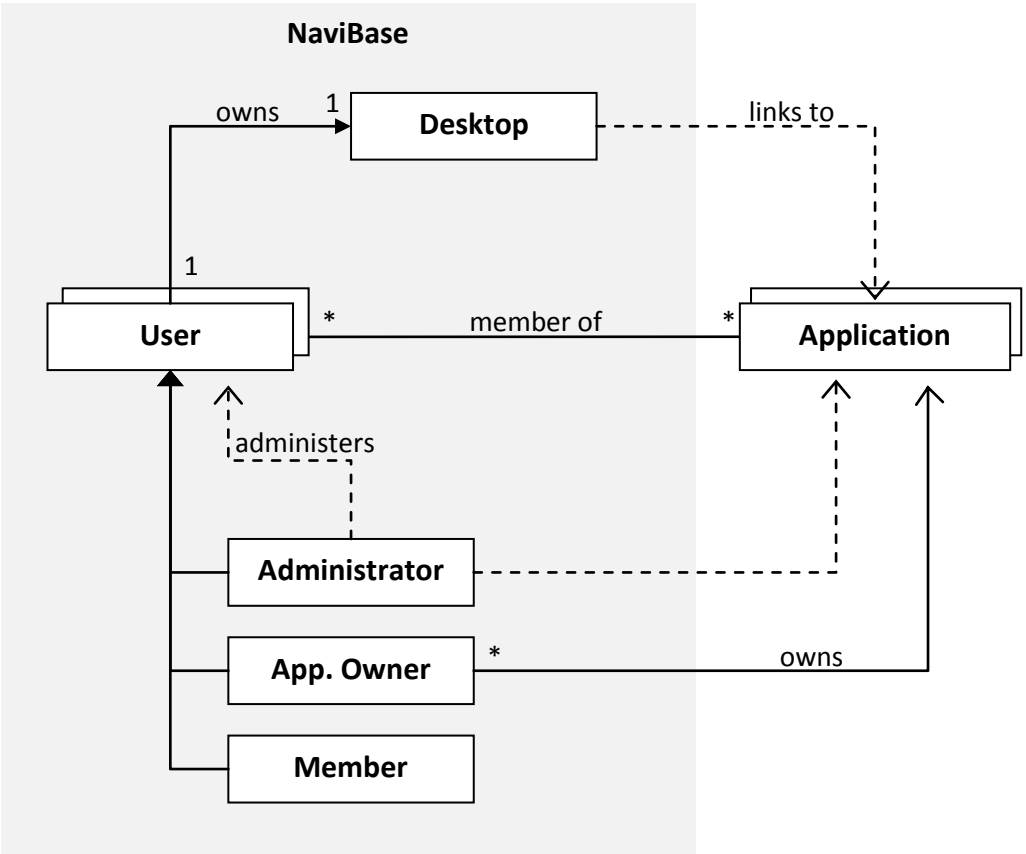

FIGURE 2: AN OVERVIEW OF THE CONCEPTS

NaviBase: Working name for the single sign-on system produced within the frame of this thesis.

User: A human being who utilizes the system. Can have different authorization levels. Has access to a personally customizable desktop.

Member: A user of NaviBase who has no administrative rights. Can be a member of and log in to one or more applications.

Administrator: A user who is authorized to add, edit, and remove users, applications and other entities in the system.

Application owner: A user who manages and administers one or more applications.

Desktop: A start page which is customized for every user. Can be further personalized by the user. Used to access applications, change settings, and perform other related tasks.

Application: An existing (third party) web site which uses NaviBase as single sign-on provider. Can also use further services provided by NaviBase if available. No matter if the user goes directly to this application or through the NaviBase user interface, the user will be automatically logged in.

4.2 Functional Requirements

Through interviews with project stakeholders, various functional requirements were gathered. These requirements are organized into use cases, where each use case describes a typical scenario for what a user might want to achieve. These use cases and requirements are summarized below.

It is worth to point out that these represent the original use cases captured during interviews, and not necessarily the exact functionality of the implemented system.

4.2.1 Generic Use Cases

Use cases under this category can be performed by any user, no matter their access level.

1) Log in through an application

By logging in to any of the applications connected to NaviBase, the user also becomes logged into the NaviBase itself, and thus indirectly all other connected applications.

2) Log in through NaviBase

A user can also log in to NaviBase through the NaviBase website (the desktop).

In this use case and the previous, special consideration was given to what means of authentication NaviBase would rely on. Except from the standard username/password model, various forms of two-factor authentication were examined. However, it was concluded that for most applications, basic username/password authentication was secure enough. Some form of two-factor authentication could be added later when applications requiring a higher degree of security were added to the system.

3) Register new user

A user which is not yet a member of the site can register for membership through an easy process. The user needs to provide a valid email address, a password, and some basic user information to complete this process.

4) **Go to user's desktop**

A user can view her desktop, which lists connected applications where the user is currently a member.

5) **Add/remove existing applications**

At their desktop, the user can add and remove applications, that is, choose to become or stop being a member at the applications connected to NaviBase.

6) **Change password**

Through the desktop, the user can change the password which she uses to get access to NaviBase. In order to do this, the user needs to provide her old password.

7) **Reset forgotten password**

If the user cannot remember her password, she can set a new by re-verifying her email address, that is by responding to an email which is sent to the email address she used to sign up.

8) **Go directly to application**

After having established a session with NaviBase, a user can go directly to an application through any means (e.g. by typing the URL in their web browser) and become authenticated without having to enter any information.

9) **Go to application through desktop**

A user can from her desktop choose to visit any site where she is, or wants to become, member by clicking a simple link.

10) **View user's basic information**

A user can from her desktop choose to view the basic information entered upon registration.

11) **Modify user's basic information**

A user can from her desktop choose to update or otherwise modify the basic information she filled out when registering.

12) **Grant/revoke access to user's contact information for application**

A user can from her desktop grant or revoke different applications the right to read her basic information. By accessing this information, an application can in most cases simplify their registration process by requiring the user to enter less information.

13) **Log out from NaviBase through the desktop**

The user can from her desktop choose to end her NaviBase session. This also ends all sessions the user currently have active on various applications.

4.2.2 Application Related Use Cases

These use cases can be performed by administrators or application owners, but not regular members. An administrator does automatically have owner privileges over all applications, whereas an application owner only over certain applications, as allotted by administrators. For all other applications, they are just regular members.

14) List applications

From her desktop, an administrator can view a list of all applications connected to the system.

15) List owned applications

From her desktop, an application owner can view a list of application which she owns, i.e. have administrative rights over. For users who are not application owners, this alternative does not exist at all.

16) View application details

From above mentioned lists, an application owner or administrator can view a page with detailed information about an application (name, url, owner, statistics, and more) as well as various administrative tools for the application.

17) Edit application

From the page in case 16), an application owner or administrator can edit the applications details (name, URL …). If the user is administrator, she can also change application owner.

18) Add new application

From her desktop, an administrator can choose to add a new application to the system.

19) Remove application

From her desktop, an administrator can choose to remove an existing application. This also effectively terminates membership for all users, as NaviBase will no longer accept authentication requests from this application.

20) List users registered through application

An administrator can through her desktop view a list of all users registered through a certain application.

4.2.3 User Related Use Cases

Use cases related to handling of users.

21) List users

From her desktop, an administrator can view a list of all users in the system.

22) Search for user

From her desktop, an administrator can search for users in the system, based on multiple criteria such as name, application membership, and other information available in the system.

23) View details about user

From the lists mentioned above, the user can view a page with information about a user including basic information, and any application memberships or ownerships. Also, a number of administrative tools are available.

24) Add new user

From her desktop, an administrator can add new users to the system.

25) Force password change

From the user detail page mentioned in case 23), an administrator can mark a user as having to change password after the next successful login.

26) Delete user

From her desktop, an administrator is able to delete a user from the system, including any information about that user and any memberships or ownerships.

4.2.4 System Use Cases

In these use cases, the main actor isn't a human user, but NaviBase or a connected application.

1) Request authentication of user

An application can request that NaviBase authenticates a given user. If NaviBase already has established a session with that user, a response can be issued directly; otherwise NaviBase will first authenticate the user.

2) Request basi c information about user

An application can request basic information (name, address, and more) about a given user in order to simplify registration or other processes where such information is needed. The user will have to explicitly accept every such request.

3) Merge two users

If for some reason, a single human being ends up having multiple NaviBase accounts, these can be merged into one.

4) Log out inactive user

After a certain amount of time during which a user with an established session has not performed any action in NaviBase, she will be automatically logged out. This log out includes logging out in any application to which she is recently authenticated.

5) Log system errors

If an unhandled error or exception occurs in the system, it should be logged to a special error log file.

4.3 Non-Functional Requirements

Except for the functional requirements of the system, there is another important set of requirements. These requirements are criteria that can be used to judge the operation of a system, rather than specific behaviors. Below is a summary of the non-functional requirements.

4.3.1 User experience

User experience requirements are criteria established regarding the user experience of the system. A few examples are provided below, but in general, these are left out of the thesis as the thesis rather focuses on the system from a technology and security viewpoint.

- The graphical design of the system shall be consistent and comprehensible and intuitive for experienced and inexperienced users alike.
- Navigation shall be effective and few clicks needed to reach frequently used functionality.
- The user interface shall adhere to Web Content Accessibility Guidelines, Priority 1, in order to be accessible for people with various disabilities.
- The system shall support user agents compatible with Internet Explorer 5.5 or later, and Mozilla Firefox 1.0 or later.
- The system shall be designed so that the risk of user errors becomes as low as possible while keeping usability high.

4.3.2 Technical Platform

Except for the user experience requirements, there are a number of requirements about the platform on which the system is built. These are used to make decisions in the Architecture section.

User and System Interface

- The graphical user interface shall be based on XHTML 1.0 and CSS 2.1 or newer.
- All communication between NaviBase and client applications shall be based on open and well tried standards.
- All communication between NaviBase and client applications shall use SSL 3.0 / TSL 1.0 or a more secure channel.

Documentation

- All application programming interfaces (APIs) shall be fully documented.
- A tutorial-style piece of documentation shall be produced which describes how an application developer can adapt their software to use NaviBase for authentication.
- An example application which demonstrates the above shall also be created.
- All source code elements shall be documented.
- The database schema shall be documented.

Quality Assurance

1. Automatic tests with 100% statement coverage shall exist, that is, that every line of code is exercised by at least one automatic test.

4.4 Security Threats

By creating attack trees, additional requirements can be found. These are naturally more security related, focusing on what a potential attacker might want to achieve and then the different ways of achieving that. Three main targets of the system are impersonation of a user identity, (ab)using a client application or service, and getting hold of sensitive information about the system. These are discussed in order below.

For each target, the various ways of achieving that goal are described. For each goal more details are provided, as well as some notes on how dangerous that risk actually is and what can be done to remedy the threat.

4.4.1 Impersonation and Identity Theft

An attacker may want to gain control over a user's (online) identity. This can be done for a number of reasons, such as using another's identity to obtain goods and services, or posing as another when apprehended for a crime. Here, three major ways are described.

1) Getting hold of the user's credentials, such as username *and* password.

a) By guessing credentials through the normal user interface.

So time-consuming that it is virtually impossible.

b) By tricking the user to provide it, through a so called phishing attack.

A serious risk, which is impossible to remedy through a technical solution only. Educating the user and ensuring that it is easy to handle credentials in the system safely.

c) By getting access to the NaviBase database.

Hopefully hard to get access. Even if the database is compromised, passwords are hashed together with a salt using a strong cryptographical hash. This makes this type of attack more time consuming, but not impossible.

2) Using an already established session between the user and the server.

a) Get access to a logged-in session while the user leaves (temporarily) is away from the computer.

Except from encouraging the user to log out after using the system, the most common way of handling this problem is using a session timeout so that the session ends automatically after a certain time of inactivity. However, the problem still exists if the attacker can access the computer within that window. Another (or complementary) solution is to require password re-entry even with an active session in order to perform certain, more sensitive, operations.

b) High-jack an logged-in session through the network.

Again, there exists not one single solution to this problem, but a range of partial solutions may be used. For example, the server might store the IP address of the computer which performed the authentication in order to ensure that no other computer replaces the first one.

3) Trick the NaviBase SSO service into believing that a valid request was performed.

a) By using a security vulnerability in the open protocols on which the system is based.

We minimize this risk by relying on well known, published and reviewed, security algorithms and protocols.

b) By using a security vulnerability in the implementation of the system.

Try to ensure, through extensive testing, that the implementation matches the specification.

4.4.2 (Ab)using a client application or service

An attacker might want to access an application or service which relies on NaviBase for authentication. The reason for doing to do so might be to use a service for free or use a services to which the attacker otherwise could not gain access. On the other hand, the goal can also be to perform a denial of service (DOS) attack to hinder other users from using the service. Here, two major ways of achieving this goal are described.

1) Trick the application by impersonating NaviBase as SSO service.

a) Performing a man-in-the-middle attack somewhere between the real SSO service and the application.

Given that the communication protocols used are secure and used properly, this should be impossible because messages should be encrypted and/or digitally signed.

b) Cracking NaviBase in order to be able to modify the messages sent by the server.

Again, try to en sure a bug free implementation through extensive testing.

c) Getting hold of NaviBase's private key, with which the attacker can generate messages which the application cannot tell apart from valid messages.

Apart from ensuring a correct implementation, we also need to ensure a secure computing environment for the server as private encryption keys will exist in computer memory at most times. Also, any backup locations (e.g. disc or tape based backups, possibly offsite) must be secure in order to ensure that private keys are not compromised.

2) Crack the application to allow the unauthorized access.

a) Gain access to the client application through any of a variety of different means.

This is probably (hopefully!) a more likely scenario. While we certainly do not hope that any application will or can be cracked, individual application developers most likely can or will not perform as extensive testing and auditing for security related problems as would be needed for a SSO solution such as NaviBase. In the end, these problems are out of scope for this work.

4.4.3 Getting hold of sensitive information

Finally, an attacker might want to, not gain control over any entity, but simply to get some confidential or sensitive information about some part of the system, its users or connected applications. This information could either be used directly by the attacker, perhaps to create a competing service, or be sold by the attacker to a third party. Two major ways to reach this goal are described.

1) **Get the information directly from the server which stores it.**
   a) Cracking NaviBase in order to be able to read (or modify) the information.

      As been mentioned previously, this type of problem is remedied by trying to en sure a bug free implementation and a secure computing environment through extensive testing.

2) **Collect information while it is in transit over the internet.**
   a) Eavesdrop and read the information as it passes through some node on the internet.

      This problem is mostly avoided by encrypting sensitive data. Some statistical data is more or less impossible from not leaking, since it is revealed by just communicating. For example, how often applications contact the server and vice versa.

4.5 Architecture

Out of the requirements mentioned above, an architecture was constructed. For the most parts, it is a rather conventional architecture and it follows most of the "best practices" available in the field. Different aspects of the design are described below.

4.5.1 Client/Server

The most basic architectural paradigm in the system is that of the client/server model. This is quite naturally inherited from the way Internet is constructed. In fact, this pattern occurs multiple times within the system.

First, the system acts as a web server for its users, as is typical for services on the web. Clients here are the users' user agents (typically web browsers). Secondly, the single sign-on system is a server to various client applications requesting authentication. These clients could be virtually anything from a web site or rich client applications to web services and scripts.

In both of these cases, there were not really any other reasonable alternatives. Given that it is a SSO system for the web and the use of SAML, the major blocks of the architecture are basically given.

4.5.2 Critera for Languages and Frameworks

The next major question is what programming language to develop the system in. Three major factors were included in the evaluation; maturity of the language and libraries, available frameworks, and previous experience (i.e. how well the developer knows it).

The maturity and stability of the chosen language is always important. In some cases, one can be a make a bit more risky choice and go with a not yet, or not fully, proven solution. In this case, however, security is of the highest importance. Thus, a stable language with proven track record and well-tested security libraries was highly important.

Secondly, given a stable language, it is very helpful to be able to work with powerful frameworks to speed up development. After all, it is more interesting to develop the project-specific parts than the boiler plate code to accept and route HTTP requests.

In case multiple options fulfilled these two priorities, a selection was made based the developer's previous experience with and knowledge about the systems.

The Java Technology Stack

Based on the above selection process, the Java technology was chosen. Other alternatives such as .NET, Python or Ruby fell either on the third criteria, or a combination of the first two. Java, on the other hand, has a reputation for being a stable platform for business-critical applications and has been used for many enterprise scale web projects.

Except from choosing the latest version of the Java language and virtual machine, which was version 5 at the time, a number of other frameworks to speed up development were also chosen.

Java web development is based on the Servlet specification. A servlet represents a service which handles requests and provides responses. To run a Servlet, an application server is needed. There are a number of web servers available. For this work, Apache Tomcat, which is a reference implementation of the Servlet specification available under an open source license, was chosen. (Apache Software Foundation 1999)

On top of this application server, which can run an arbitrary Servlet-based application, a web framework is often used. While the Servlet specification already includes a basic object model for handling requests and responses, it is further enhanced by the web framework. Here, the choice of of the Apache Struts framework was made. It is a competent yet rather lightweight and unobtrusive framework. The major benefit from using Struts is that it handles the internal information flow, which includes request routing, separating model, views and controllers, and more. For more details on this separation, see the description of the Model View Controller pattern in the Theory section. (Apache Software Foundation 2000)

For persisting data, the highly competent relational database management system MySQL, available under an open source license, was chosen. On top of this, open source persistence framework Hibernate is used. This speeds up development by freeing the developer from communicating directly with the database through SQL. Instead the domain model can be worked with directly.

4.5.3 Criteria For Single Sign-On Protocol

There are a number of goals for the single sign-on system. One of the major objectives is achieving high security. To do this, it is important to stick to tried and true security protocols and solutions, because creating good security algorithms is extremely hard. This suggests choosing an existing single sign-on protocol rather than developing a proprietary one. (Viega and McGraw 2002)

A secondary but important consideration is the ability to be compatible with as many potential clients as possible. This translates into favoring existing de jure or de facto standards, as it is more likely that clients already use them or that high quality libraries exist.

The Security Markup Assertion Language Protocol

A quick survey of available such implementations reveals two primary options and a few other solutions that didn't quite fit. The two primary choices are the Security Assertion Markup Language and the OpenID protocol.

As noted earlier, SAML is a unification of a set of different protocols. Thus, it is has become somewhat of a de facto standard. It also relies on well known standards such as XML and uses RSA encryption and the SHA hashing algorithm. These were the two major reasons behind finally choosing SAML for this thesis.

The other major contender, OpenID is created with a somewhat different goal in mind. It is still a single sign-on protocol, but the guiding idea behind it is to decentralize the identity management. While this can be a strength for the protocol in general, it doesn't encourage a single vendor to base their technology on it unless it is already a established standard. Furthermore, it is somewhat lacking in the security department, where a number of security issues have been reported. (Brands 2007)

Given the choice of SAML, the next question becomes that of finding an implementation of the protocol which can be used. Here, we seem to be out of luck. At the time of writing, the only non-proprietary implementation of the SAML protocol available is OpenSAML. (Internet2 2005) It does however only support the version 1 branch of SAML. Work is under way to create a SAML 2 branch, but no indications regarding when or even if it might be finished are available.

Because of the lack of available implementations of the SAML 2.0 protocol, the decision was made to create an own implementation of the SAML protocol. Compared to reusing an existing implementation this costs a lot of time and also has a higher risk (as noted about writing your own security implementations, above). However, the newer version of SAML includes many useful improvements, especially pseudonyms and session management. Regarding the higher risk of writing an implementation from scratch, much of this risk can be tackled by using well-tested implementations from the Java libraries of actual security algorithms such as RSA and SHA.

5 Result

This section describes the Single Sign-On system called NaviBase, which is the result of the execution phase of this thesis. First, we get an overview of the system which will provide a broad view of the system, and then each of the different components that make up the system are described in detail.

5.1 Overview

There are four major components and two supporting utilities in the system. The two *primary components*, the components which are always in use, are listed below.

- *NaviBase*, the server component which holds all information and processes requests; and
- *SamlLib*, responsible for building, parsing, validating, and processing SAML messages.

Secondary components, components which may or may not be used depending on the circumstances and configuration, include the following.

- *ClientLib*, creates requests and interprets responses according to the rules of SAML on behalf of its client; and
- *ClientWebService*, which acts as a wrapper around ClientLib for all non-Java clients.

The *supporting utilities*, typically used by NaviBase system administrators, are as follows.

- *KeystoreGenerator*, used to generate private/public key pair containers; and
- *Builder*, used to build, package, and deploy the above components.

These components and utilities are first described briefly below in terms of responsibilities and external communication. They are then described in further detail in separate sections further down, including their design and detailed descriptions of their functionality.

5.1.1 Client applications

The server component *NaviBase* is responsible for handling client requests, holding state, providing users and administrators with configuration user interfaces. It communicates with any number of clients.

A *client* is typically written by a third party vendor to perform some work or provide some service. Exactly what it does is irrelevant in this context, but they all have in common that they wish to benefit from the Single Sign-On service provided by NaviBase.

On this level, communication between server and client is performed through the SAML protocol, with HTTP as the carrying transport protocol, as specified by the SAML Web Browser SSO Profile. More details on this protocol can be found in the Theory section.

Three Types of Clients

The clients using the Single Sign-On platform can be divided into three groups, depending on how many of the optional components they make use of. These are displayed in Figure 3: Three types of clients, where the clients labeled A, B, and C, represents the three types.

- Type A: As all clients communicate with the NaviBase server through the SAML protocol, a client is required to master that protocol. Clients already proficient in this protocol are called *SAML-capable clients*. They typically already use the SAML protocol to exchange information about users and authentication in some capacity. Such applications have none or very little need of modification to be able to use NaviBase's Single Sign-On service.

- Type B: Most applications interested in starting to use a Single Sign-On solution do not have support for the SAML protocol as they previously have had no need for it. These applications can delegate the SAML processing related tasks to the helper library ClientLib. If the client is written in Java, it can call ClientLib directly. Such clients are called *Java clients*.

- Type C: Many clients are not written in Java, but in other languages which typically have no way of calling methods in a Java library directly. For such clients there is the ClientWebService. It acts as a wrapper around ClientLib and exposes its methods as Web Service methods. There is Web Service support available for every major programming language. Clients using this method are called *non-Java clients*.

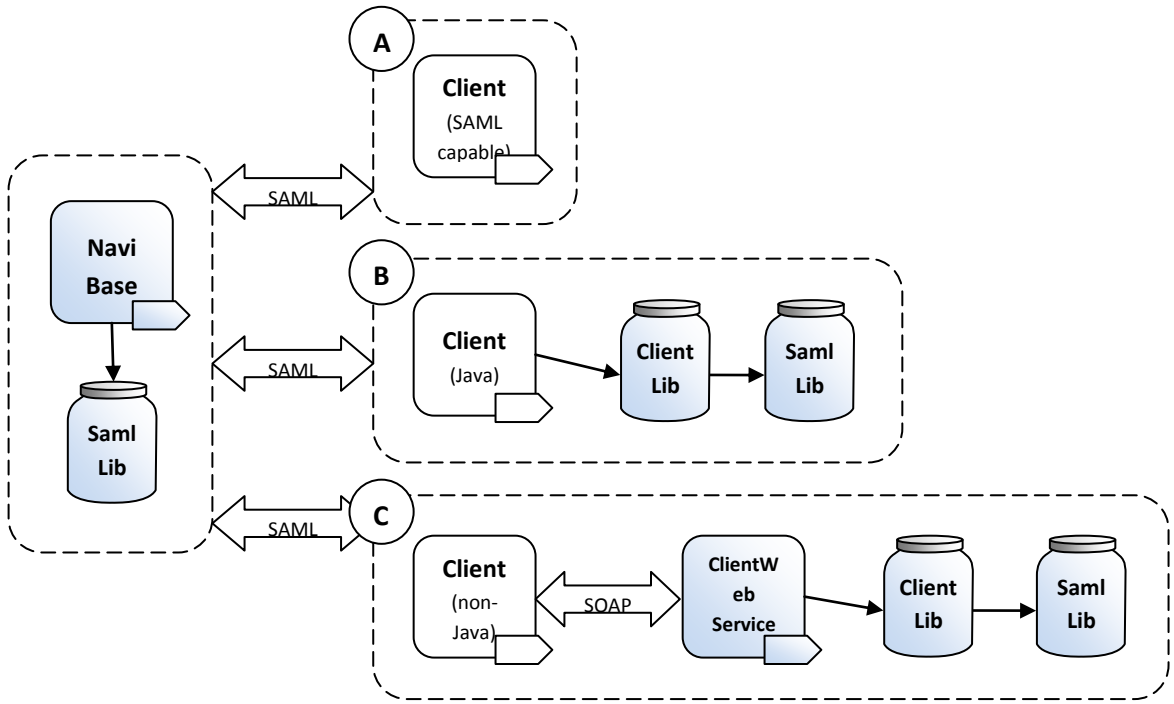

FIGURE 3: THREE TYPES OF CLIENTS

5.1.2 Server components

These six components and utilities are described briefly in terms of responsibilities and external communication.

NaviBase

The most obvious responsibility for NaviBase is handling client requests. This includes receiving and processing them, and finally issuing a response if appropriate. All communication between NaviBase and its clients is performed through the sending of a SAML message. Typically this means one of

`AuthnRequest`, `ManageNameIDRequest` or `Response`. All messages are digitally signed with the issuer's private key. Any communication is also sent over Secure HTTP (HTTPS).

Secondly, the NaviBase server component is responsible for holding the state of the system. It is, in essence, what enables the "single" part of the Single Sign-On system. It knows about all clients and the users which use one or more of these applications. Except from credentials needed for authentication, it also stores extra information about the user, such as contact information and other types of meta information.

Finally, there is a user interface aspect to NaviBase. It needs to provide both normal users as well as administrators of the system with a user interface which lets them use and configure various aspects of the system.

SamlLib

The SamlLib contains the parts of both server and client components which directly deals with the SAML protocol messages. This includes, constructing, modifying, and serializing/deserializing them to/from XML.

The library also takes care of managing digital signatures attached to these messages. SamlLib handles both signing messages as well as validating signatures attached to messages.

ClientLib

To ease client development, client developers have access to a Java-based helper library called ClientLib which can process messages according to all the rules of the SAML protocol on behalf of the client.

ClientWebService

If the client cannot call a Java-based library, it can use the ClientWebService to act as a Web Service wrapper around the ClientLib.

5.2 NaviBase

NaviBase is the main component in the architecture. It is also the server component. It holds in its database with information about all applications and users, and processes authentication requests sent by applications.

This section first describes the general design of the NaviBase component. It then delves into all of the major parts of the system and their functionality. Finally the minor components of the system are briefly described. All of these sub-sections make references to actual classes and files in the code.

5.2.1 Design: Model View Controller

The major design pattern underlying NaviBase is the Model View Controller pattern, described in the Theory section.

The Model

The code of the model lives inside the `com.navibase.model` package. It doesn't at all concern itself with the technicalities of persistence or storage of its information; it leaves all of that to the controller. See Persistence under Supporting Functionality below.

User Management

Perhaps the most basic functionality contained in the model is user management. The user functionality is needed in almost all aspects of the system and king of the user management classes is the `User` class. It is the starting point to find every piece of information there is about a user. It contains the unique email address, password, and contact information (such as first and last name) of that user. It also has methods to retrieve the "application provided name" for that user for any given application, but more about that below.

Two of the most important data members of the `User` class are email and password, represented by classes `Email` and `Password`, respectively. Together they constitute the credentials needed to authenticate a user in the system. Both classes contain functionality to check if a string is a valid email address/password.

Because of the special importance of the password, a user's actual password is never stored in plain text; not in the database or in the object model. Instead, the `Password` class stores a salted hash of the password at all times. Thus, to validate a password, one sends the candidate password into the `Password` class, which adds the salt, hashes it, and compares it with the actual hash.

Applications

Another important aspect of the model is applications. They represent the client applications using NaviBase to provide authentication and other SAML-based services for them. A client application is represented by the `Application` class. It stores an application's unique identifier (which is a URL), a URL at which it expects SAML responses to be sent, the application's public key, and a set of so called application provided names.

To simplify adoption of the NaviBase authentication services an application is able to tell NaviBase under what name it knows a certain user. This name is called an "application provided name" and is used by NaviBase in all future communication with that application (and that application only). Such a name is represented by the `ApplicationProvidedName` class.

Registration

The final major part of functionality in the model is related to registration of new users. Before a new person wanting to use NaviBase is represented by a `User` object she must verify her email address. During this process she is represented by a `Registration` object. This process is described in more detail in the Registration section under Main Functionality below.

The View

The structure of the view layer in NaviBase is inherited from the Struts web framework on which NaviBase is built. As it is a web framework it relies on the concept of pages, which should be familiar to anyone who's used the Internet. More specifically, the view consists of multiple Java Server Pages (abbreviated JSP) pages.

Most of the views are part of the application's graphical user interface, but some views are in XML format which are intended for other computers to consume. This applies particularly to the SAML library, discussed in greater detail elsewhere. The view code in NaviBase exists in the `WebContent` folder but is not covered in detail in this report.

The Controller

Controller code is placed in the `com.navibase.controller` package with sub packages for different types of functionality.

Actions and forwards

All actions are represented by Struts class `Action`. NaviBase then extends this base class with an `ActionBase` class which is the base for all actions in NaviBase. It has a number of helper methods which returns the current request's user, the server's keystore (containing its private and public keys), a data access object (see section Persistence below), and a few other things. It is further specialized by class `ValidatingAction` which encapsulates some forward processing logic.

User input to an action is typically processed through a so called action form. It is a data structure which corresponds to the forms on web pages. The action forms become natural places for any type of validation, parsing, or other type of pre-processing logic. The `ActionFormBase` class is the base class for all action form classes and contains helper methods used by all or most action forms.

The wiring

While controller classes contain much functionality, the flow of the system is defined in a XML-based configuration file called `struts-config.xml`. While it also contains other types of configuration, the main part of the file consists of a set of action mappings. An action mapping links an action class to a specific URL. It also specifies what possible forwards the action can make use of, what action form is used (if any), and more.

The example given in Figure 4 describes a mapping for the `AuthnRequestPreLoginAction` which is used to verify that a authentication request from a client application is valid. It is accessible through the URL `/SSO/Authenticate` and makes use of the `SamlRequestForm` action form. It has two forwards which it can use depending on the input given. The first one is called `failure` leads to another action available at URL `/SSO/SendSamlResponse` and is used if the request is invalid and it needs to send a failure response message. The second is called `login` which points at

```
<form-bean
    name="samlRequestForm"
    type="com.navibase.controller.saml.SamlRequestForm" />
<action
    path="/SSO/Authenticate"
    type="com.navibase.controller.saml.AuthnRequestPreLoginAction"
    name="samlRequestForm">
    <forward
        name="failure"
        path="/SSO/SendSamlResponse.do" />
    <forward
        name="login"
        redirect="true"
        path="/SSO/AuthenticateUser.do" />
```

FIGURE 4: AN EXAMPLE OF AN ACTION MAPPING WITH BELONGING ACTION FORM DEFINITION.

`/SSO/AuthenticateUser` and is used when the request is valid to continue with the next step of the process, actually authenticating the user.

5.2.2 Main Functionality

This section goes through all parts of NaviBase describes what they are, their purpose, and how they work. Some controller functionality has been skipped, as it technical level or importance to this document is low. Providing a welcome page to the user is one such example.

Login and logout

This section describes the controllers handling authentication functionality such logging a user in or out. Relevant code exists in `com.navibase.controller.saml` and `com.navibase.model.saml`.

Any time a user wants to access a protected resource the system needs to authenticate the user and determine if the user is authorized to access the resource. The resource could be a user's personal start page, some administrative functionality, or a SAML request consumer.

Authentication is currently based on username and password. These credentials are looked up in a user database and the user's identity is determined. A HTTP session is established for the user by a unique cookie sent to the user. In this session the identity is stored for future authentication. If the user cannot be found in the user database, he or she is returned to the login prompt with an appropriate error message. The user also has the option to register as a new user. See the Registration section below for more details on this.

If the user's identity was established the next step is to determine if the user is authorized to access the resource in question. All authorization in NaviBase is role based. That means that every user belongs to one or more roles, and each resource requires a specific role to be accessed. The roles available in the system are currently `user` and `register` used for normal users and users who have started but not yet finished registration, respectively.

On authorization, the requested resource is displayed to the user. If authorization failed the user is displayed HTTP status code `403 Forbidden`. Specific actions or views also have the option to customize material based on role membership. For example, after a normal login a normal user is redirected to his or her personal start page while a user who haven't finished registration is sent back to the registration step.

When a logged in user does not use the system any more, the session for that user is destroyed. This means that the next time the user wants to access the system he or she needs to re-authenticate in order to establish a new session. Logging out a user can be triggered either by an explicit request from the user or through a timeout. By default, the user is logged out after two hours of inactivity.

For more details on how user authentication and authorization is performed on a technical level see Security under the Supporting Functionality below.

Registration

One of the first part of the system which a user gets in contact with is the registration system. The registration process is divided into three parts, initial registration, email verification, and contact info entry. The code resides in the `com.navibase.controller.registration` package.

First, the user is presented with a form, `begin.jsp`, where she can enter her email address and a password of her choice. When the form is submitted, the information is passed through `RegisterUserForm` which either rejects the information (based on e.g. an invalid email address) and sends her back to the form, or forwards the user to the next step.

That next step is the `SendRegistrationEmail` controller which sends out an email to the address he user specified. This leads us to part two of the registration process, namely email verification. For multiple reasons, we want to verify that the email address that the user specifies is indeed valid. While it does not guarantee that the user is indeed a human being, it at least makes it a bit harder to create fake accounts. Furthermore, by assuming that only one person has access to that email address, we can use this email in the future to verify that the same user is using the account. For example, when the user forgets her password, she can prove her identity by once again verifying her email address. Finally, a valid email address gives us way to reach the user with information.

In the email sent to the user, there is a so called ticket. It is a randomly chosen sequence of letters and numbers which the user must enter. This ticket is generated when the email is sent and associated with that account. The user enters this ticket at a page, `ticket.jsp`. The user also enters her password, which she provided in step one. These two data items are sent to the ticket verifier, `VerifyTicketAction`, where they are both matched against previously stored values. If they match the email address is considered validated and the user can continue to the last step of the registration process.

Before we let the user in as a member, we require the user to provide some basic information about herself. The required information may vary depending on what the administrator wants, but would typically include first name, surname, street address, city, zip code, and phone number. This information can later be provided to applications, as described in the functional requirements in the Analysis chapter. The files and classes involved here is a fairly typical view-form-controller sequence including `contact-info.jsp`, `ContactInfoForm`, and `FinishRegistrationAction`.

SAML

One of the most critical pieces of the NaviBase server, both in terms of functionality and security, is the SAML controllers. These handle authentication and communicating with client applications. For more details on the requests and responses sent by these actions, please refer to the Theory section. Code can be found in the `com.navibase.controller.saml` and `com.navibase.model.saml` packages, as well as in the `SamlLib` described elsewhere.

Base functionality

As the foundation are a set of classes which handle SAML request and responses. The `SamlRequestActionBase` is a super class for all actions dealing with SAML requests. It decodes the incoming encoded request, parses it using the `SamlLib` library described below, validates it according to the rules of the SAML protocol, and finally creates a `SamlRequest` object representing it. If anything during this process goes wrong, such as if it is unable to decode the message or validate its digital signature, the incident is logged, but message is dropped with no response to the requester. This is done to prevent an attacker to perform a denial of service attack through sending bogus requests which then force the server to generate a proper response, which is a non-trivial task including digital signatures.

After the message has been parsed, the `SamlRequestDispatcherAction` is the first action to see it. The dispatcher figures out the message's type, such as authentication or name ID management, and forwards control to an appropriate action.

Authentication Requests

An incoming authentication request from an application is first handled by the `AuthnRequestPreLoginAction` class. It verifies that the authentication request is valid, using rules specific to authentication. If everything is OK, it continues to verify whether the user is already authenticated. If not, the user is sent to a login form to provide valid credentials needed for authentication. When the user is authenticated, A `AuthnRequestPostLoginAction` continues to generate a response to the requesting application. This response provides the application with information about who the user is, when and how she was authenticated and more.

Applications can request authentication in a number of slightly different ways by providing a set of flags. First, through the `allowCreate` flag an application can indicate that NaviBase is allowed not only to show a login prompt, but also to let users go through a registration process if needed, before returning control to the application. The `isPassive` flag tells NaviBase to not display any kind of graphical user interface to the user. If the user is currently not authenticated, this means that NaviBase will not be able to authenticate the user and will return a negative response. Finally, applications can host the login form on their own domains if they want to. In that case, they can send the username and password provided by the user, in encrypted form, to NaviBase which performs the actual authentication.

NameID management

An incoming request might also be about management of a so called NameID which is what SAML uses to identify a certain user. These requests are handled by the `ManageNameIDAction`.

To protect the privacy of users, applications only know the user under a pseudo-random name which does not reveal anything about the user's actual name or location. This name is also unique for every application. That means that two applications cannot combine their databases to figure anything out about the user as they know the user under different names. The user can choose to allow the application to receive more information, however.

To simplify the life of application developers, applications can also provide a name for each user which NaviBase then will use in future communication with that application. Again, these names are per application.

Also, to enable an application to host not only login form, but change password forms as well, the application can provide (in encrypted form of course) a new password for NaviBase to use. This requires a bit more trust between NaviBase and the application developer however, as the password is not unique to that application but used for all applications. (It is a Single Sign-On system after all.)

5.2.3 Supporting Functionality

Apart from the functionality described above, there is quite some code which is more general purpose, or of supporting character. This supporting functionality is made available to the other functionality through what is called filters. A filter is a piece of code through which every request and response must pass. Filters therefore have the ability to augment the request or response with extra

information or capabilities. This ability is used to provide actions with database access, access to encryption keys and more. The functionality described below is implemented as filters.

Persistence

Persistence of the model in NaviBase relies upon the Hibernate framework, described in the Theory section. In practice, it is done by annotating the model classes to tell Hibernate about relations between different entities. Most classes can be mapped automatically by Hibernate in a rather straightforward fashion. Typically, classes are represented by tables, fields of primitive data types by columns, and fields of object types as foreign keys into other tables. Extra annotations may be needed when creating more advanced associations between objects, however.

To simplify the database access even more and to encapsulate all database related code into a single layer, NaviBase uses a `DataAccessObject` object as *façade*. This also means that all SQL code generation is encapsulated in one place which allows us to ensure in one single place that all data that goes into the database has been properly escaped. A simple implementation of the DataAccessObject pattern was chosen over using the EntityManager pattern as it was judged overly complex.

Most database tables are very straight forward. The only thing worth mentioning is that passwords of course are salted and hashed before storage.

Security

There are a number of helper classes related to security, but the parts which are worth mentioning is the key stores where private keys used for encryption are kept. NaviBase as well as every application has their own pair of private and public keys. In all cases, they are stored in a so called key store, an encrypted file in which the keys are kept safe from prying eyes. The format used for this keystore is defined by Sun in their implementation of Java. The key store is secured by a password, which either is entered when starting the application server, or stored in a file on disc, depending on the computing environment. In the latter case, it is obviously of great importance to protect that file from being read by an attacker.

Also, a file called `securityfiler-config.xml` is used to declare which actions require what security roles. This file decides for example, that login is required to reach a user's desktop, or that not all users may access administrative tools, but only those who hold an administrator role.

Finally, rules regarding passwords are worth mentioning. As discussed in the Theory section, passwords become harder to break the longer they are and the more kinds of characters they contain. But they also become harder to remember. Therefore, it is not obvious where to draw the line. A password is currently required to be at least six characters or longer with no demands regarding what characters to use. Probably, administrators want to set a bit harder rules, such as at least eight characters, and at least one letter, one number and one other character.

5.3 SamlLib

As discussed in the Analysis chapter, an implementation of the SAML 2.0 specification should be created for the project due to lack of existing stable implementations. However, not all parts of SAML 2.0 were needed for the project, so only the parts which were actually needed should be

implemented. It could be described as “a slimmed-down ‘mission-specific’ partial SAML 2.0 implementation.”

This implementation is named SamlLib, short for SAML Library. It consists of two major parts, one for creating messages and turning them into xml, and one part which takes care of the reverse. Both parts are described below. Finally, security threats identified in the Analysis section are discussed.

5.3.1 SAML Object Model

The library contains an object model of (parts of) the SAML 2.0 specification. Ease of use was a higher priority when designing that object model than exact adherence to the specification in terms of names and concepts. The code resides in the `com.navibase.saml` package.

The foundation of this object model is the `SamlObject` class. It represents a generic SAML object and takes care of the basics of XML generation such as namespaces and other more technical details. It is extended by objects `SignableSamlObject` which represents objects which are to be digitally signed, such as messages. It, quite naturally, takes care of all work relating to digital signatures.

Finally, there are a number of objects representing concepts in the specification. For example, `AuthnRequest` represents an authentication request. Each object which is to be converted to XML is responsible for doing so itself, with the help from the basics which are available in `SamlObject`. The package also contains classes which represent profiles and bindings in the SAML specification. See the Theory section for details.

5.3.2 SAML Parser

While the SAML object model itself is responsible for converting itself to XML (know as unmarshalling), the reverse process is a bit more difficult. A lot more validation and conditional logic is required to successfully convert XML into objects in the above object model. Therefore, that responsibility was extracted into its own package, `com.navibase.saml.parser`.

The façade of this package is the `SamlParser` class which parses an incoming request in four steps. At any step of this process, if a part of a message is found which the parser does not fully understand and knows what to handle, the whole parsing is stopped. This is done in order to ensure that we do not provide a response to a message in error.

1) Parse the XML string into an XML document object model. This gives us much better ability to work with the XML document. It also performs basic XML well-formedness tests.
2) Not only does the XML string need to be valid, it has to validate against the SAML 2.0 schema. This ensures that we know what to do with the information given to us, that it is a proper SAML request.
3) Given valid XML, we want to create SAML objects out of the XML – a process know as marshalling. Here, a number of rules are used to ensure a well structured object model.
4) If we could turn the request into an object model, we finish the process by validating all digital signatures in the message. These signatures ensure that we know that the message was sent by the application which claims to be the sender.

5.3.3 Handling identified security threats

In the Analysis section, a number of security threats were identified through the use of attack trees. A number of these affected the implementation of this SAML library directly. These are discussed below.

First, an attacker might try to impersonate a user by using an already established session between that user and the server. For example, this type of attack could be performed if the user leaves her computer while logged in. This problem is primarily dealt with through a session timeout. After a given period of time, currently set to two hours, any session which has not been accessed is invalidated. This period is easily changed if necessary. This is a typical question where security and usability are at conflict with each other. The shorter timeout you use, the better, from a security perspective. But for a user, it would be more convenient if the session never timed out. A second solution to this problem would be to require re-authentication for certain sensitive actions, such as changing password. This is planned for, but not yet implemented.

Another way to impersonate a user would be to hijack an existing session through the network. Here, there's not one single solution to this problem, but a range of partial solutions. The fundamental protection here is based on a randomly generated key being hard for an attacker to guess. Other solutions which are planned but not yet implemented are IP address checking, requiring all requests to originate from the same IP as from which the authentication was performed. Also, the session key could be changed to a new randomly generated value right after authentication, to stop an attacker which somehow might have set the session key for a certain user.

Finally, an attacker might try to perform a man-in-the-middle attack somewhere between the SSO service and the application. This is stopped by using well thought-out protocols and by encrypting and/or digitally signing messages going back and forth.

5.4 ClientLib

In the beginning of this chapter, we discussed different types of clients. For any client not naturally proficient in SAML, a Java based helper library called `ClientLib` is provided. It provides a simple façade for the whole `SamlLib` in order to make creating SAML requests and parse SAML response as simple as possible. For instance, it deals with all encryption key handling on behalf of its client. The code for this library can be found in `com.navibase.client`.

It also enforces a few rules which the client application is required to do according to the SAML 2.0 specification. This includes things such as ensuring that the same response isn't parsed twice if received a second time and to ensure that the response was really intended for us.

5.5 ClientWebService

Again, in reference to the different client types, there are some client applications which neither speak SAML nor are written in Java. Those applications cannot directly use the `ClientLib`. However, they can use the client library through a XML web service based wrapper. This wrapper is called `ClientWebService` and is available in the `com.navibase.client` package. It can be called from virtually every programming language there is, as there almost always is a web service implementation available.

This `ClientWebService` is in fact not simply a library called by the client, but an application in it's own right. It contains a small web server which listens to web service requests, translates them into their corresponding `ClientLib` calls.

5.6 Supporting Components

A list of components which are not major or required parts of the architecture, but which are important to ensure that the system is as easy to operate as possible.

5.6.1 KeystoreGenerator

In order to simplify the process of generating keystores for applications, a certain generator tool was created. It takes as input names of server and client, and outputs a keystore containing private and public key pair, required certificates and more. The code for this is available in `com.navibase.util.keystoregenerator`.

5.6.2 Builder

As a tool for system administrators, the `Builder` component automates the build process after the source code has changed. It automatically compiles the NaviBase server as well as the SamlLib, ClientLib, and ClientWebService components. The new NaviBase server code is automatically uploaded to the server (if desirable). It also regenerates any client kits (see below) complete with new versions of the libraries and their keystore.

5.6.3 NaviBase Client Kit

For simple deployment and ease of use, every client gets a so called *NaviBase Client Kit*. It includes any necessary software library and resource needed by clients to access the NaviBase server. The complete content of the kit is as follows.

- ClientLib, ClientWebService, and SamlLib, described above;
- An encryption key store holding the client's private key, and the NaviBase server's public key; and
- Possibly, client language specific tools to provide simplifications or avoid incompabilities between the client's platform and any of the components mentioned above.

Using the functionality of this kit is completely optional; it is up to the client developer to decide what components might suit their client the best. The only thing that is really necessary in the kit is the included encryption key store. Without it, there is no way to digitally sign a request, something which is required in order for NaviBase to accept it.

6 Discussion

After descriptions on what the objectives were, how the construction was planned and executed, and seeing the analysis and results, this section focuses on how things actually worked out. The discussion starts by looking at the objectives, did we reach them? Then the planning and analysis is investigated and finally we look at the execution and the actual results. We discuss the rationale behind the choices made, and interpretations are made. This section is by its nature more speculative than the previous and the author's own thoughts will be expressed explicitly.

6.1 Objectives

In the Introduction section, we saw that the system had two primary objectives; security and usability. We now look at these two one at a time and see how well the result fulfilled the promises.

Regarding security, the testing performed indicates that the system is indeed secure. Here, the focus on Test-Driven Development was likely a big help. However, more extensive testing needs to be performed before a more definitive conclusion can be drawn. See also notes on future work in the Conclusion section. The thing that could have been done differently and might have a big impact on security would have been to use an existing implementation of SAML. More on that further ahead.

While things look rather good from a security standpoint, the usability of the system doesn't look quite as bright. Usability simply didn't get nearly as much attention as other aspects, and it suffered. It became more a "mirror of the system", than a "mirror of the user's intentions."

So, all in all, does the system do what it was supposed to? Unfortunately, the answer is no, not quite. Two major reasons are unclear requirements, and optimistic time estimation. More on both of these subjects further down.

6.2 Planning and Analysis

We now look at the planning phase, and the analysis that came out of it. We look first at how requirements were gathered, through interviews and using attack trees, and then look at the initial architecture and design, including the use of the SAML protocol.

6.2.1 Requirements

Requirements for the system were gathered in two primary ways, through interviews with project stakeholders, and through the use of attack trees. I believe this structure works very well, covers the whole area effectively, and gives us a good chance of finding requirements relevant to our objectives.

However, while interviews themselves might be good, they don't help if stakeholders are unclear about what they really want. The requirements gathered during the interviews were not completely clear, and a close scrutiny of them would have revealed inconsistencies and various aspects of the system that were never considered.

Here, I am at fault myself for settling with unclear requirements where I instead should have kept on pressing harder to get to the core of what system was needed. Either I thought that I actually had a clear view, or I had a moment of self-delusion and just wanted to think that I did. Either way, the project would have benefited from a more extensive and thorough gathering of requirements.

Regarding attack trees, I believe again that they are a very helpful tool. They introduce a good way of thinking, where you start with thinking about what is actually valuable and needs protection, then how attacks could be performed, to finally delve into the details of such attacks. While I believe they produced a number of highly relevant scenarios, I think an even more carefully constructed attack tree could have revealed further types of attacks.

If I allow myself to second-guess myself, I can imagine a reason for the insufficient requirements gathering. I believe that I at the time, might have been a bit too inspired by (and also partly misunderstood) the Agile software development "revolution" which was raging at the time. That I was a bit too optimistic thinking things "would sort themselves out".

6.2.2 Architecture

The initial architecture and design planning consisted of two primary decisions; what programming language and frameworks to use, and what Single Sign-On protocol to use. As we recall, the choices med were Java with Tomcat/Struts, and the Security Assertion Markup Language, respectively.

Using Java for this project was the right choice, in my opinion. It is a highly competent language with an extensive and well tested framework, including many security related features. Tomcat, Struts and Hibernate also worked out very well. My only regret here is that I should have spent more time on learning the frameworks rather than learning as I went along. That would have saved me from a few somewhat painful experiences and partial rewrites.

Regarding SAML, it is very much the same story – I think it was the right choice. It is a joint effort to create a complete SSO protocol, and it does its job well. Again, more time spent before implementation to completely understand the specification would have saved me some headache.

On the subject of SAML, I also think we can find one of the major reasons for things taking longer time than estimated. I underestimated the effort it took to develop an implementation of the SAML specification. Tasks such as dealing with digital signatures in XML and getting every tiny detail of the XML schema right proved to be very hard and tedious work. In retrospect, I should have tried harder to use an existing implementation of the specification, even if it would have meant using an older version of the protocol, or working with a library under development.

As a final note regarding architecture, I should have spent more time on an initial design of the system. While the things that I did decide in advance (language, frameworks, protocol etc) were a valuable, things could have been even smoother if I had planned the major elements of the system design as well. Again, as noted regarding the requirements, I probably let myself to think that such details would sort themselves out during development. I could point out however; that I don't think the end result was worse than had I planned more, I just think that I could have saved myself a lot of rework.

6.3 Execution and Results

As the last block of the discussion, we look at the actual implementation of the system, and its results. First, we go through a couple of techniques used during execution, and secondly each of the components of there resulting system.

6.3.1 Domain Modelling

I believe that the key to a successful object oriented system is a strong and well functioning domain model. The reason is that while controller code or a view is used in some places, the model is used in all parts of the system. Here, I believe test-driven development serves a nice purpose in that it, as described in the Theory section, tends to result in objects designed for use rather than for implementation. After all, you implement them only once, but use them all the time.

Also, I believe using the Hibernate persistence framework was a very good idea. It makes changing the domain much easier as there is no persistence code with hairy SQL statements which need to be updated. Of course, there is no such thing as a free lunch, Hibernate comes with its own set of problems, but overall it worked very well. One feature that I never got around to actually use, which could have given even bigger benefits, is automatic schema generation. That is, where Hibernate not only writes and stores to the database, but also generates and modifies the database schema if necessary.

Rather little time was spent on areas which traditionally might have been more important, such as database schema design, database normalization, and so on. The reason for this is that the database design is pretty much given from the conventions of Hibernate.

6.3.2 Test-Driven Development

All development was done using Test-Driven Development. As described in the Theory section, its goal is to improve design by focusing on use before implementation, and it increases test coverage.

The latter point is perhaps the most obvious, that it increases test coverage, and indeed it does. It quickly helped building up a large set of fully automated tests. The exact number varies as tests are restructured over time, but were at the end of the project in the 600 to 700 range. Altogether, they took about one and a half minute to run. This is actually rather much, as you want to be able to run the tests as often as possible. However, many of the tests relied on encryption and digital signatures, which are computationally intensive processes, and it wasn't possible to do very much about it.

Secondly, doing test-driven design forces you into designing "testable" code; code with fewer dependencies and which rely less on the state of other objects or even global state. Unfortunately, you cannot always control the design of the code you're working with, such as when using libraries and frameworks, and these were sometimes not quite designed for testability.

An unexpected (for me) side effect of doing test-driven development is that after a while, writing code without tests felt very insecure, even unprofessional.

6.3.3 NaviBase

As the major component of the system, NaviBase quite naturally got most of the attention during development. I also tried to incorporate a number of good principles and patterns while building the system, some of them as described in the Theory section.

One such example is the DRY ("Don't Repeat Yourself") principle. This principle is used in places too many to mention, but everywhere a common method is extracted from duplicated code, or a new super class is created for classes that share much functionality, this principle is the driving spirit. Also, using the Tiles library in Struts is an example of the DRY principle. It gives us the ability to reuse user interface blocks which appear multiple times in different places.

Another pattern, or set of patterns, worth mentioning are Barricade and Choke Point. As described in the Results section, every database request goes through the DataAccessObject. The class therefore becomes a choke point for database queries and a perfect place to ensure that all SQL queries sent to the database are safe. This keeps us free from SQL injections. It also works as a barricade in the system – on one side, the data is assumed to be unescaped, on the other always escaped. This works very well in this system, as the application code in front of the barricade doesn't really care about SQL injections. They aren't a problem until we send the query into the actual database.

Another example of Barricade and Choke Point in cooperation is the various ActionForm subclasses in the system. Every time a piece of information reaches the server, it goes through one of these forms in order to be validated. Here we get an opportunity to remove potential XSS (Cross Site Scripting) attacks before the data reaches the rest of the system.

Finally, we look at cryptography. Most of the issues here, such as choosing secure encryption and hashing algorithms is taken care of by the SAML specification. Some implementation specific details related to security however, still exists. For example, where do you store the password to the keystore containing the server's private encryption key? On one hand, you want to keep it as far away from the keystore as possible, possibly not even in digital form. On the other hand, you want the server to be able to start and restart without manual intervention, thus requiring it to be on a medium which can be reached by the server. I don't have a solution other than that to keep the password in a file on the same or another server, and then ensure that an attacker cannot get hold of that password. In the end, if an attacker can put their hands on the server running NaviBase, they can probably get the encryption key anyway, without having to get the password.

6.3.4 SamlLib

The next biggest component, after NaviBase, is the SAML specification implementation called `SamlLib`. As mentioned previously, pursuing an own implementation of the specification was probably a mistake. I underestimated the amount of time it would take to build it, especially with regards to digital signature handling and encryption.

If at all possible, I should have tried to use an existing SAML implementation, or at least base my own implementation on an existing one. The main problem here was that no open source implementation existed which implemented the SAML 2.0 specification which was needed for this project. Perhaps, a reasonable solution would have been to use the OpenSAML implementation of SAML 2.0 which was at the time under development. Then, one would simply have to hope that it had reached a mature level before the end of the project. But that is also rather risky.

6.3.5 ClientLib and ClientWebService

Finally, we look quickly on the client application helper library `ClientLib` and its complementary `ClientWebService`.

Regarding ClientLib, I can't really say much more than it worked out very well. It was intended to be a simple façade to the more complex SamlLib and encapsulate some of the SAML processing logic required. That is just what it does.

Since ClientLib was implemented in Java, ClientWebService was meant as a way to ensure that programs on any platform, written in any programming language, could benefit from ClientLib. It is simply a XML Web Services wrapper around the client library. While this sounds nice in theory, it

turned out rather clumsy. It is currently a bit too hard to set up and use properly for it to be very convenient.

7 Conclusion

This section provides a brief summary of the report, including conclusions and the most important considerations from the discussion. We then briefly look into possible future work.

7.1 Achievements

While too much time was spent on unplanned things, the things that were actually built works as expected. A focus on sound development principles and using well known design patterns proved to be a successful recipe. The Don't Repeat Yourself (DRY) principle helped in reducing duplication in the project, and the Model View Controller (MVC) pattern gave a very good structure to the NaviBase code base. Using Test-Driven Development we gained not only a comprehensive set of test cases, but also code which is more usage-oriented.

7.2 Lessons Learned

The work described in this report was aimed at creating a Single Sign-On system which was both secure and easy to use. In the end, a Single Sign-On system was produced which worked and according to preliminary security testing is secure enough. However, it is lacking in the usability department. The main reason for this was a lack of usability focus, and inadequate requirements.

Much time could have been saved by learning relevant frameworks and protocols better before starting development – learning while doing did work, but proved to be rather inconvenient.

Finally, creating an own implementation of the SAML 2.0 specification was a hard choice, and while it is hard to tell, it might have been a bad choice. The amount of time it would take to build was underestimated and likely, a solution based on an existing SAML implementation (possibly one under development) could have given better and quicker results.

7.3 Future Work

This section lists a few examples of work that would be suitable in a near future.

As noted above, usability aspects of the system did not get as much focus as they deserved. This means that the system would benefit from having an overhaul. This would be everything from deciding on a graphical profile to site structure and more.

Before the launch of a system which relies so much on security as a Single Sign-On provider, extensive security testing needs to be performed. Thus, a suitable future step is to perform a complete security audit, where every part of the system is evaluated from a security perspective.

Also, the system can become more capable as a Single Sign-On system. One interesting area is for example so called federated identity, where two applications can combine their information about a user to synthesize new information, unavailable to each application separately. This of course would have to be performed with the consent of the user.

8 References

Apache Software Foundation. *Apache Struts.* 2000. http://struts.apache.org/ (accessed December 25, 2008).

—. *Apache Tiles.* 2001. http://tiles.apache.org/ (accessed December 25, 2008).

—. *Apache Tomcat.* 1999. http://tomcat.apache.org/ (accessed 25, December 2008).

Brands, Stefan. ”The problem(s) with OpenID.” *The Identity Corner.* den 22 August 2007. http://idcorner.org/2007/08/22/the-problems-with-openid/ (accessed December 28, 2008).

Hoare, C.A.R. ”The Emperor’s Old Clothes.” *Communications of the ACM*, 1981: 75-83.

Hunt, Andrew, och David Thomas. *The Pragmatic Programmer.* Addison-Wesley, 1999.

Internet2. *OpenSAML.* 2005. http://www.opensaml.org/ (accessed December 28, 2008).

Maler, Eve L. *Minutes of 9 January 2001 Security Services TC telecon.* January 9, 2001. http://lists.oasis-open.org/archives/security-services/200101/msg00014.html (accessed September 28, 2007).

McCabe, Thomas J. ”A Complexity Measure.” *IEEE Transactions on Software Engineering* SE-2, nr 4 (1976): 308-320.

McConnell, Steve. *Code Complete.* Second Edition. Microsoft Press, 2004.

OASIS. *OASIS Standards and Other Approved Work.* 2007. http://www.oasis-open.org/specs/index.php (accessed September 28, 2007).

OASIS. *SAML V2.0 Executive Overview.* Edited by Paul Madsen and Eve Maler. April 12, 2005.

—. "Security Assertion Markup Language (SAML) V2.0 Technical Overview." *OASIS Security Services (SAML) TC.* October 9, 2006. http://www.oasis-open.org/committees/download.php/20645/sstc-saml-tech-overview-2%200-draft-10.pdf (accessed November 22, 2007).

OpenSAML. den 10 June 2008. https://spaces.internet2.edu/display/OpenSAML/Home/ (accessed July 31, 2008).

”PlayStation a hacker's dream.” *The Age.* den 27 November 2007. http://www.theage.com.au/news/security/playstation-a-hackers-dream/2007/11/26/1196036813741.html (accessed February 17, 2008).

Red Hat Middleware. *Hibernate.* 2006. http://www.hibernate.org/ (accessed December 25, 2008).

Reenskaug, Trygve. "MVC." *Trygve Reenskaug homepage.* December 10, 1979. http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf (accessed January 20, 2008).

Schneier, Bruce. "Schneider.com." *Attack Trees.* December 1999. http://www.schneier.com/paper-attacktrees-ddj-ft.html (accessed February 9, 2008).

trscavo@idp.protectnetwork.org. *Differences Between SAML V2.0 and SAML V1.1.* February 5, 2007. https://spaces.internet2.edu/display/SHIB/SAMLDiffs (accessed September 28, 2007).

Viega, John, and Gary McGraw. *Building Secure Software.* Addison-Wesley, 2002.

Wikipedia. *Password fatigue.* April 24, 2007. http://en.wikipedia.org/wiki/Password_fatigue (accessed October 3, 2007).

—. *Single sign-on.* 2009. http://en.wikipedia.org/wiki/Single_sign-on (accessed January 1, 2009).