



IT UNIVERSITY OF COPENHAGEN

BACHELOR PROJECT

---

# Verifiable Secure Open Source Alternative to NemID

---

*Authors:*

Andreas Hallberg KJELDSSEN  
*ahal@itu.dk*

Morten Chabert ESKESEN  
*mche@itu.dk*

*Supervisor:*

Joseph Roland KINIRY  
*josr@itu.dk*

*Co-supervisor:*

Hannes MEHNERT  
*hame@itu.dk*

May 22, 2013

## **Abstract**

Your abstract goes here...

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Objectives . . . . .	4
1.2	Scope . . . . .	4
1.3	Background . . . . .	5
<b>2</b>	<b>Technical background</b>	<b>6</b>
2.1	SAML protocol . . . . .	6
2.1.1	OIOSAML . . . . .	7
2.2	Static analysis . . . . .	7
2.3	Protocol verification tools . . . . .	7
2.4	N-factor authentication . . . . .	8
<b>3</b>	<b>Modelling the protocol</b>	<b>9</b>
3.1	A protocol walkthrough . . . . .	9
3.1.1	Logon protocol . . . . .	9
3.1.2	Identity provider registration protocol . . . . .	11
3.1.3	User registration protocol . . . . .	11
3.2	Protocol prerequisites . . . . .	11
3.2.1	Shared . . . . .	12
3.2.2	NemID specifics . . . . .	12
3.2.3	OpenNemID specifics . . . . .	13
3.3	Formalizing protocol messages . . . . .	13
3.4	Communication model . . . . .	14
3.4.1	Message processing . . . . .	14
3.4.2	Communication diagram for NemID . . . . .	14
3.4.3	Communication diagram for OpenNemID . . . . .	16
<b>4</b>	<b>Modelling with F*</b>	<b>25</b>
4.1	Introducing F* . . . . .	25
4.2	Syntax and semantics . . . . .	26
4.3	Refinement types . . . . .	26
4.4	Formalizing OpenNemID by using Jacob's work - THIS IS A BAD TITLE! . . . . .	27
4.5	OpenNemID specified in F* . . . . .	29

4.5.1	Specification of the type functionality module . . . . .	30
4.5.2	Specification of the SAML Protocol . . . . .	30
4.5.3	Specification of cryptographic elements . . . . .	32
4.5.4	Specification of certificate store module . . . . .	33
4.5.5	Specification of the messaging protocol . . . . .	34
4.5.6	Specification of the Service Provider . . . . .	35
4.5.7	Specification of the Identity Provider . . . . .	36
4.5.8	Specification of the Database Handler . . . . .	39
4.5.9	Specification of the Authentication Provider . . . . .	40
4.5.10	Specification of the Browser . . . . .	44
4.6	Introducing adversaries . . . . .	48
4.7	State of the implementation . . . . .	49
<b>5</b>	<b>Evaluation</b>	<b>50</b>
5.1	Project evaluation . . . . .	50
5.2	Related work . . . . .	50
5.3	Threats to validity . . . . .	51
5.4	Further research . . . . .	51
5.5	Conclusion . . . . .	52
<b>A</b>	<b>SAML Assertion and Messages</b>	<b>55</b>

# List of Figures

3.1	Sequence diagram of authentication with NemID . . . . .	10
3.2	Sequence diagram of authentication with OpenNemID . . . . .	11
3.3	Communication diagram for the complete NemID protocol . . . . .	15
3.4	Communication diagram for the OpenNemID protocol . . . . .	17

# Listings

4.1	Hello World example in $F^*$ . . . . .	26
4.2	Multiplication example in $F^*$ . . . . .	26
4.3	Simple refinement types in $F^*$ . . . . .	26
4.4	NemID identity provider implementation [1] . . . . .	27
4.5	Cryptographic elements . . . . .	29
4.6	TypeFunc module . . . . .	30
4.7	Specification of the SAML Protocol elements . . . . .	30
4.8	Specification of cryptographic elements . . . . .	32
4.9	Abstract certificate store . . . . .	33
4.10	Specification of the Messaging protocol . . . . .	34
4.11	Specification of service provider . . . . .	35
4.12	Handling and delegation of a user's requests . . . . .	36
4.13	The handling of the responses from Authentication Provider . . . . .	38
4.14	Establishing a secure connection with Authentication Provider . . . . .	39
4.15	Specification of the database . . . . .	39
4.16	Specification of the authentication provider . . . . .	40
4.17	The creation and changing of a user's account . . . . .	42
4.18	Established a secure connection with the Identity Provider . . . . .	43
4.19	Browser's side of logging in . . . . .	44
4.20	The Browser's side of the account creation and changing . . . . .	46
4.21	Main module for introducing adversaries . . . . .	48

# Chapter 1

## Introduction

We're extending the work done by Jacob Højgaard in his Masters thesis 'Securing Single Sign-On Systems With Executable Models'. Jacobs research has focused on the current implementation of NemID and therefore describes, outlines and models the current system used in Denmark as of May 2013.

Jacobs report sums up some of the problems with the current implementation of NemID, these problems include but are not limited to the system being very opaque. We're in the age of information, we want to be able to find information about everything regardless if it's in our best interest or not. NemID is not sharing any information about the internals of the system. If a person wanted to test some part of the NemID system, they would first have to analyze the public parts of the system to figure out how to communicate with the NemID system, afterwards all they would be able to do, would be black box testing. We need a system that is transparent, testable by everyone and doesn't cost a whole lot. For the sake of being able to reference this system through out the report, we will refer to it as *OpenNemID*.

### 1.1 Objectives

Some explaining text here

1. Describe and outline the OpenNemID protocol, including but not limited to registration and login.
2. Formalize the specification of OpenNemID in  $F^*$  to the extent possible.

### 1.2 Scope

This project has had it focus towards specifying a new protocol that could replace NemID. The intent of this project is therefore not to develop a complete

system, but to make the specification for a system that could then later be developed based on the specification.

## 1.3 Background

...

## Chapter 2

# Technical background

This chapter will clarify the technical concepts that has been used in this report. It will describe the SAML protocol and the Danish specialization OIOSAML. Furthermore it will describe the concept of static analysis and outline different protocol verification tools and our choice of tool. Lastly it will introduce the concept *N factor authentication* that has been used in the development of this project.

### 2.1 SAML protocol

The Security Assertion Markup Language or SAML is an XML based language created for the exchanging of authentication and authorization (security tokens) between different systems or domains, in particular between a service provider and an identity provider. The latest version of SAML is 2.0 released in 2005. SAML specifies three roles: *principal*, *identity provider* and *service provider*. Principal refers to an entity that can be authenticated, therefore both the user and the identity provider and service provider are principals. SAML addresses the problem of web browser single-sign on (SSO). SSO means that the user only has to login once pr identity provider pr session. Practically this means that the user only has to be redirected once to the identity provider for authentication once pr session. The identity provider creates a session for the user that can be reused for any subsequent authentication request. SAML specifies 5 core elements - *Assertions*, *Protocols*, *Bindings*, *Profiles* and *Metadata*. See appendix A for examples of assertion and SAML messages.

*Assertions* are security tokens containing statements or claims about a principal. These statements in SAML are called attributes and they usually hold information like name, last name, email etc. The principal is referred to as the *subject* in an assertion.

*Protocols* describe the messages that can be exchanged between the service provider and the identity provider when exchanging *assertions*. The protocol used in OpenNemID is called the Authentication Request Protocol. This



protocol consists of AuthnRequest message and a response message. The AuthnRequest message has an id that must be unique and the creator of the message is responsible for ensuring it is. The response message has a field *inResponseTo* that specifies the id of the AuthnRequest the response is to.

*Bindings* specifies how the messages are mapped to the underlying HTTP(s) or SOAP protocols. However in this report only HTTP POST and HTTP REDIRECT bindings will be addressed. The POST binding maps the content of messages to hidden XHTML form fields that are named SAMLRequest and SAMLResponse. The content is BASE64 encoded. For the REDIRECT binding the message content is mapped the URL query string with SAMLRequest and SAMLResponse as the identifier. Since the URL query string has limited capacity the message is compressed by the DEFLATE algorithm and after that it is BASE64 encoded and URL encoded.

*Profiles* specifies how the *assertion*, *protocol* and *binding* are used to fulfill a specific requirement or a use case. The Danish specialization of the SAML Web Browser SSO Profile, OIOSAML, will be described in further detail in section 2.1.1.

*Metadata* is the necessary data exchanged between the involved parties in order to know each other. A service provider's metadata contains the certificate for signing messages and a message that the identity provider should use for encrypting assertion. It also contains endpoints which specifies the addresses to which the identity provider should send response messages. More than one endpoint if different bindings are available. The identity provider's metadata also contains signing and encryption certificates and endpoints that specify the addresses for sending requests to the identity provider. Again more endpoints for different bindings. The identity provider will publish the attributes it is able to serve through it's metadata.

### 2.1.1 OIOSAML

## 2.2 Static analysis

## 2.3 Protocol verification tools

A communications protocol is a system of digital message formats and rules for exchanging those messages in or between computing systems. A cryptographic protocol or security protocol is an abstract or concrete communications protocol that performs a security-related function and applies cryptographic models. Cryptographic protocol can be verified formally on an abstract level. Formal verification is an attempt at proving or disproving the correctness of intended algorithms underlying a system with regards to a certain formal specification. There exists various tools to formally verify a cryptographic protocol on an abstract level. This section will outline the tools we have investigated through the work of this project and the tools' key features.

*ProVerif* [8] is an automatic cryptographic protocol verifier in the Dolev &

Yao model<sup>1</sup>. It is based on a representation of the protocol by Horn clauses<sup>2</sup>. One of *ProVerif*'s key features is its ability to handle many different cryptographic primitives, including shared- and public-key cryptographys, hash functions and Diffie-Hellman key agreements<sup>3</sup>. The other key feature is its ability to handle an unbounded number of sessions of the protocol (even in parallel). *ProVerif* can prove secrecy<sup>4</sup>, authentication, strong secrecy<sup>5</sup> and equivalences between processes that differ only by terms.

*F\** [5] is a new research language from Microsoft Research. The language is dependently typed for secure distributed programming. *F\**'s purpose is the enabling of the construction and communication of proofs of program properties and secondly of properties of a program's environment in a verifiable secure way. *F\** is based on the functional programming language *F#* and compiles to .NET bytecode in a type preserving style. Even though *F\** is formal specification language it is still executable. Furthermore *F\** has a fully abstract compiler from *F\** to JavaScript. The current version of the *F\** compiler is considered to be an  $\alpha$ -release.

*JSCert* [9] is an ongoing project. *JSCert* is short for Certified JavaScript. The goal of the project is to really understand JavaScript. The project is building models of ECMAScript<sup>6</sup> semantics in the Coq proof assistant and automated logical reasoning tools built on those semantics. The project introduces a program logic for reasoning about a broad subset of JavaScript. They have proved a strong soundness result. The libraries written in their subset and proved correct with respect to their specifications will be well-behaved - even when called by arbitrary JavaScript code.

*XOR-ProVerif* [10]

*DH-ProVerif* [11]

*CSPProto* [12]

We have chosen to formally specify OpenNemID using *F\** for several reasons. We both have previous experience working with the functional programming language *F#* which *F\** is based on and a little experience with ML. However the most contributing factors was:

1. Since we are extending the work done by Jacob it made sense to use *F\** as he did
2. The fully abstract compilation from *F\** to JavaScript
3. *F\** is executable

## 2.4 N-factor authentication

<sup>1</sup>Cryptographic primitives are assumed perfect and cyphers cannot be decrypted without the the proper decryption key

<sup>2</sup>Horn clause is a clause (a disjunction of literals) with at most one positive literal

<sup>3</sup>Diffie-Hellman keyexchange is a specific method for the exchanging of cryptographic keys

<sup>4</sup>the adversary cannot obtain the secret

<sup>5</sup>the adversary does not see the difference when the value of the secret changes

<sup>6</sup>ECMAScript is a scripting language often used for client-side scripting on the web

## Chapter 3

# Modelling the protocol

Before formalizing the protocol, it's required to specify and explain some of the words, concepts and meanings used within the protocol. This will be done by using graphical representation of the message flow.

### 3.1 A protocol walkthrough

A sequence diagram depicting the protocol is useful for introducing the basics of the protocol. The sequence diagram is not as informative as a communication diagram, therefore we will first take a look at a couple of sequence diagrams afterwards we will have a more thorough look at the some parts of the protocol using communication diagrams.

#### 3.1.1 Logon protocol

Jacob have depicted the NemID protocol with 12 steps. The sequence diagram shows how the browser must delegate messages from the service provider to the identity provider, while also having to delegate messages from DanId to the identity provider. A single challenge is used to authenticate the user after credentials have been submitted.

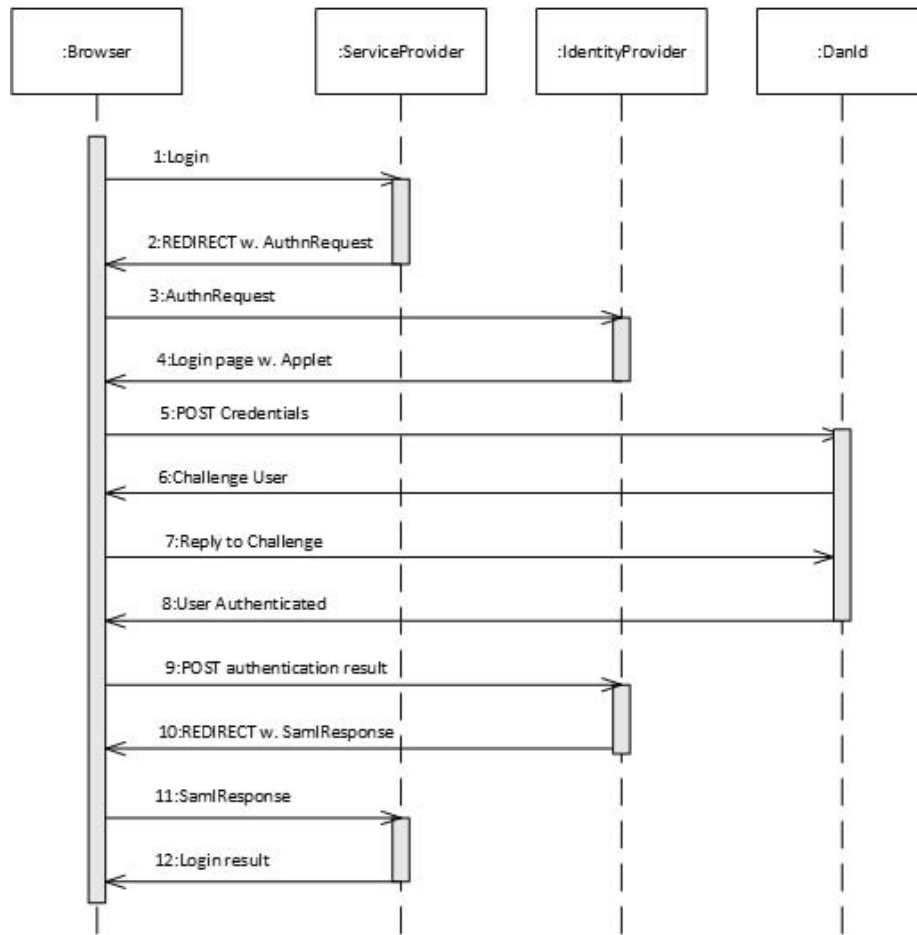


Figure 3.1: Sequence diagram of authentication with NemID [1]

In comparison to NemID, the OpenNemID protocol have modified the way messages are sent. The amount of messages the browser must delegate has been limited to few. There is not just one challenge used to authenticate the user, there is N challenges, as described in section 2.4. The amount of steps in the protocol have been reduced to 8.

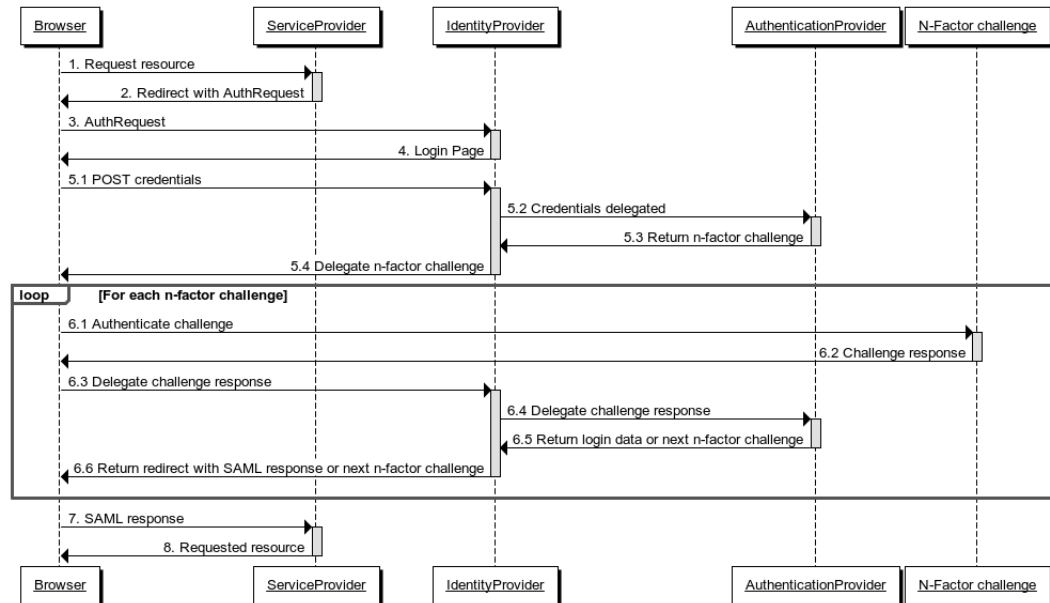


Figure 3.2: Sequence diagram of authentication with OpenNemID

### 3.1.2 Identity provider registration protocol

An identity provider has to register with an authentication provider to establish a trusted relationship and an agreement that the authentication provider will in fact act as a provider for authenticating users.

### 3.1.3 User registration protocol

Before a user is able to authenticate at the authentication provider, they have to register there first. The user has to activate their account using credentials received via snail mail. Upon activation, a new password must be chosen, optionally a userid can also be chosen.

## 3.2 Protocol prerequisites

It's important to have some requirements as to how the systems should function. The requirements helps define certain properties the involved participants must have or obey to. It's important to have some requirements as to how the systems should function. The requirements helps define certain properties the involved participants must have or obey to.

### 3.2.1 Shared

The NemLog-in specification mandates the use of OIOSAML, this will most likely not be excluded, therefore we assume that OpenNemID also has to use it. Further OIOSAML mandates the use of one-way SSL/TLS for all bindings, the mandate does not specify a specific version, though it can be assumed that a minimum version of SSL 3.0 due to the fact that SSL 2.0 is in general considered deprecated. We assume the use of SSL 3.0 or TLS 1.0 in this report.

As mentioned before, SAML uses the browser to transfer messages from one principal to the other. The way to do this is through HTTP REDIRECTs, which could be either a HTTP-GET REDIRECT or a HTTP-POST REDIRECT. The HTTP protocol accepts a Location head in the HTTP RESPONSE which indicates where the browser should redirect to. The location header redirect will act as a HTTP-GET REQUEST which excludes the usage of POST data, thereby limiting the amount of data that can be transferred. To overcome this problem HTTP-POST REDIRECTs are used, these are not a part of the HTTP protocol, but is synthesized by using JavaScript to emulate a regular HTTP-POST REQUEST. Therefore it is required for the users browser to follow redirects and to have JavaScript enabled.

For there to be any actual messages to flow between the service provider and identity provider, it's assumed that they reside in different domains and are different entities.

The identity provider is only to issue assertions to known service providers, this requires that SAML metadata has been changed beforehand. The certificates used for signing and encrypting has to be checked for revocation and validity whenever used.

**To summarize:**

1. OIOSAML mandates the use of SSL(3.0)/TLS(1.0).
2. The browser must follow redirects.
3. The browser must have JavaScript enabled.
4. Service provider and identity provider are different entities residing in different domains.
5. SAML metadata must have been exchanged between the service provider and identity provider.
6. Signature check and encryption requires validity/revocation check of the certificate.

### 3.2.2 NemID specifics

It's required for the OCES certificates used for signing and encrypting to have been issued by DanID.

### 3.2.3 OpenNemID specifics

For the communication between the authentication provider and the identity provider, a secure tunnel must have been set up. Further the user must have registered at the authentication provider.

We have assume the availability of a web cryptography API in this report. A web cryptography API has not yet been standardized, but a standardization is being worked on, a draft is available [3].

## 3.3 Formalizing protocol messages

The UML communication diagrams depicting the protocols are made up of two or more participants, henceforth principals, and the messages flowing through the system. The line between two principals indicates a channel where communication can flow, this channel is assumed to be a secure channel, meaning for HTTP messages, the HTTPS protocol would be used. An arrow indicates the direction of the message and the text on top of the arrow is the message being sent. The messages does not conform to any specific formalism, but follows a simple syntax. Messages are, very applicable, named in accordance with their HTTP protocol verb. The messages are to be interpreted the following way:

**GET** means a HTTP-GET request, the parameter is the resource being requested.

**POST** means a HTTP-POST request, the parameters are the destination for the request followed by the data being submitted.

**REDIRECT** means either a HTTP-REDIRECT or a JavaScript redirect, whichever is used is not important for the purpose of the description. The parameters are the destination for the redirect followed by the parameters to include in the redirect.

**RESPONSE** means a HTTP-RESPONSE messages. The parameters are either the data included in the response or a HTTP status code indicating the type of the response along with a message, this is used for indicating when error happen.

**DELEGATE** means forwarding the data from the previous request, the parameters are the parameters from the previous request that were to be delegated.

**AUTHENTICATE** is to be interpreted as the sequence of actions required to be authenticated at the specified NFactorChallenge. The AUTHENTICATE message is defined this generically on purpose, as the way a user would authenticate for a NFactorChallenge can vary a lot depending on which technology is used (SMS, Facebook, phone call etc.).

## 3.4 Communication model

To make the changes from NemID to OpenNemID clear, we will first show the communication diagram for NemID, afterwards we will show the communication diagram for OpenNemID. Both diagrams make use of abbreviations, these abbreviations are listed at the top of the diagram along the word or phrase they abbreviate. We have tried to comply with Jacobs data as much as possible, due to the fact that his report conforms to the requirements mandated by Digitaliseringsstyrelsen.

### 3.4.1 Message processing

Information regarding the processing of messages have not been included in the diagrams, this is to prevent cluttering of the diagrams. To circumvent this, the processing rules will be described afterwards. The descriptions will be linked to a specific process number, meaning Process 3 would be the handling and response of message 3 in the diagram. Messages that are self-explanatory will not be further described.

### 3.4.2 Communication diagram for NemID

Description of the message processing for this diagram has been left out of our report, they can however be found in Jacob Højgaards report [1].



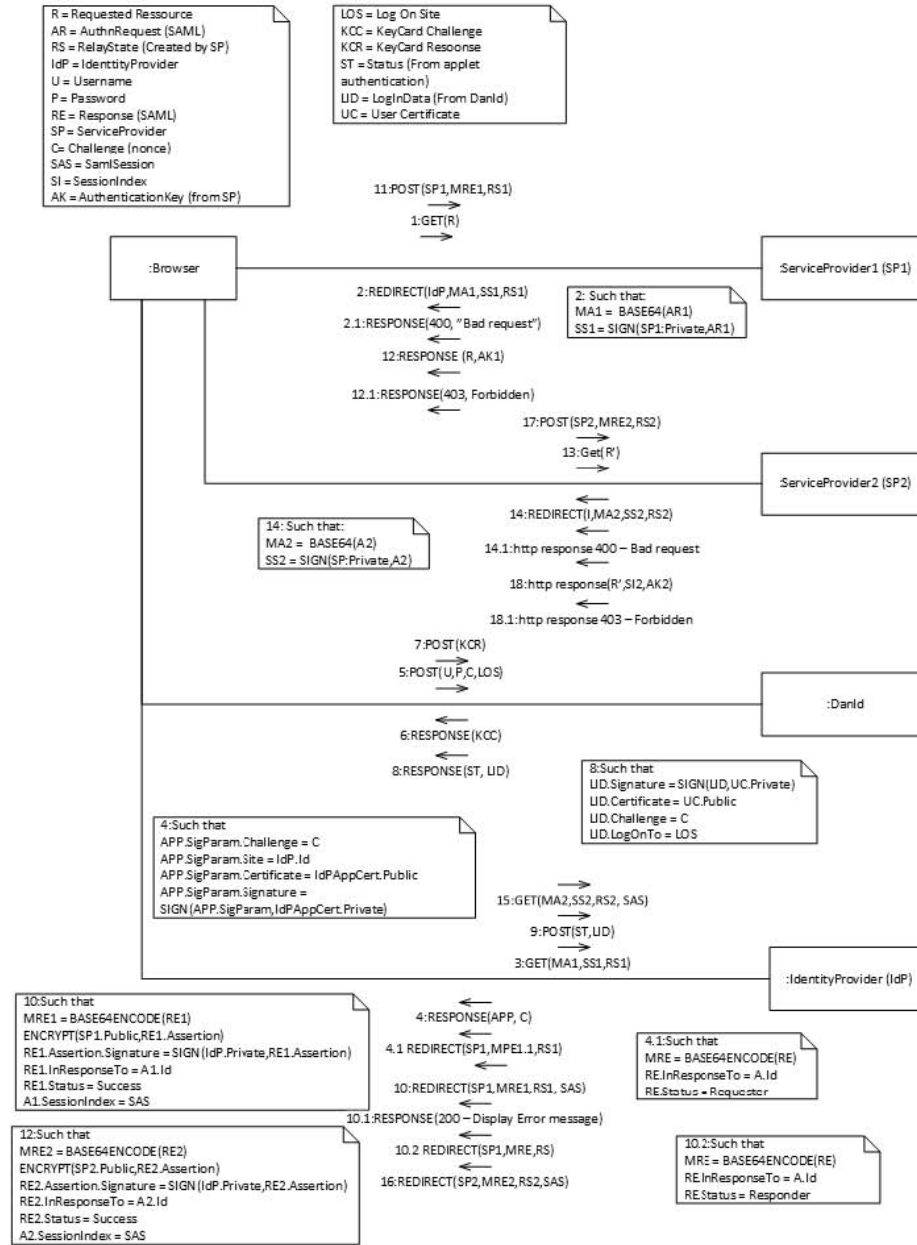


Figure 3.3: Communication diagram for the complete NemID protocol [1]

### 3.4.3 Communication diagram for OpenNemID

In this diagram, we have chosen to leave out the additional request to another service provider than the one initially used, this is due to the communication flow being exactly the same as in Jacobs diagram, see Figure 3.3, message 13 to 18.

In the communication diagram for OpenNemID, DanID have been replaced by AuthenticationProvider, for the sake of our protocol it is of no greater importance which company handles the authentication.

We have strived to minimize the amount of messages flowing through the system to limit the amount of possible attack points for a potential adversary. We have also eliminated the need for transporting sensitive data, such as the users login data, from the authentication provider to the identity provider, by mandating that the identity provider and the authentication provider exchange information using a secure tunnel without the user being able to interfere.

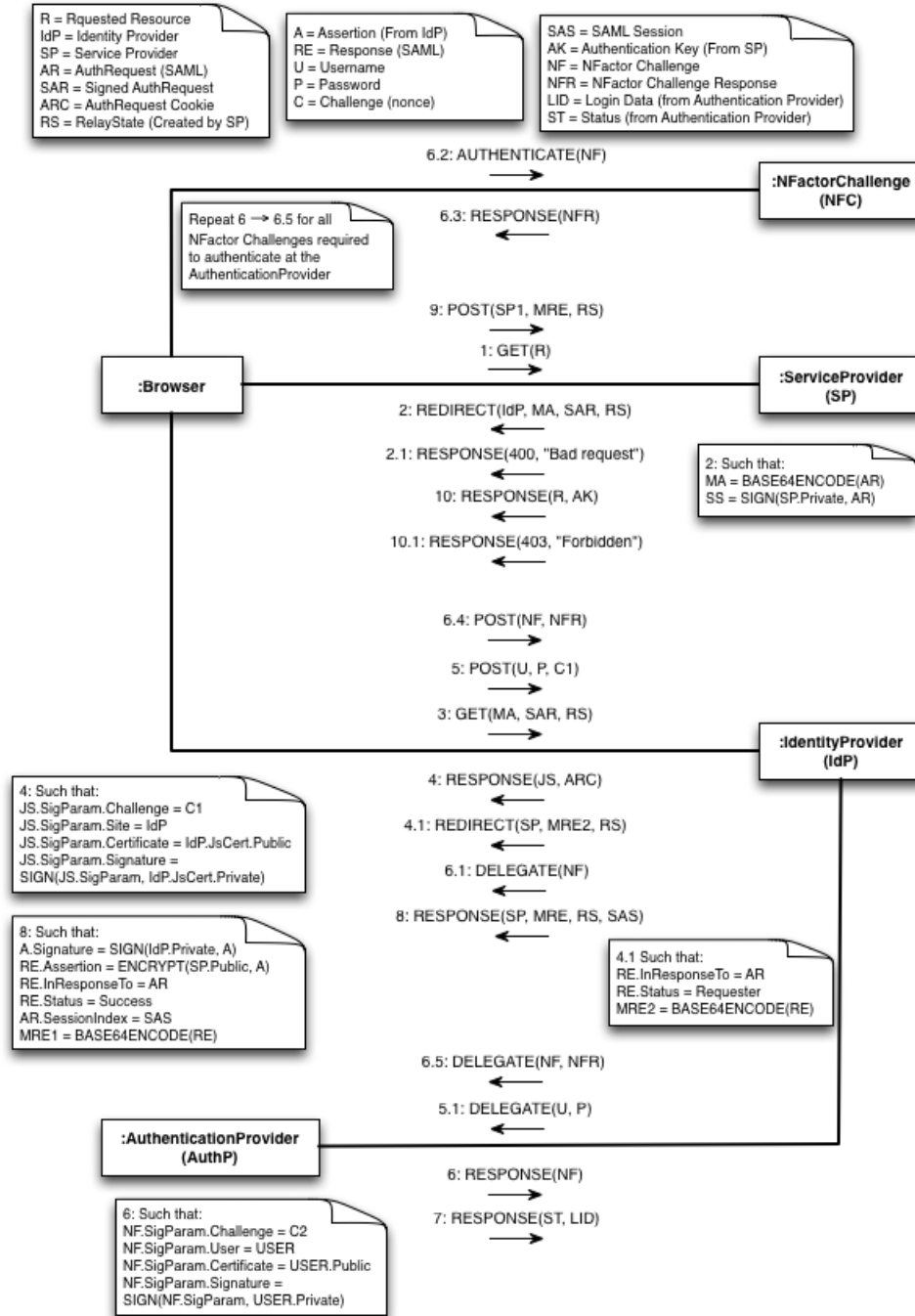


Figure 3.4: Communication diagram for the OpenNemID protocol

### Message descriptions

Process 1 describes the creation of the *AuthRequest* at the service provider when a resource has been requested. If the request resource not found a 400 Bad-Request message is returned. The use of url encoding (*UrlEnc*) and base64 encoding (*Base64Enc*) is to ensure that the data can be transferred as parameters in the URL.

SAML and OIOSAML has a substantial amount of processing rules that dictates the processing of messages, disobeying these will cause an error. For the sake of communication the descriptions have been kept to a minimum, not listening all scenarios that would cause an error.

---

**Algorithm 1** Process 1

---

**Require:** GET is well-formed **and** IdP.Public **and** SP.Private

**if** R exists **then**

    AR  $\leftarrow$  CreateAuthRequest()

    SAR  $\leftarrow$  SIGN(AR, SP.Private)

    MA  $\leftarrow$  UrlEnc(Base64Enc(DeflateCompress(AR)))

    RS  $\leftarrow$  UrlEnc(Base64Enc(R))

**return** REDIRECT(IdP, MA, SAR, RS)

**else**

**return** RESPONSE(400, BadRequest)

**end if**

---

Process 3 describes the handling of a *AuthRequest* at the identity provider. The request will be handled in one of two ways depending on whether the *AuthRequest* could be verified. In the case of verification failing, a SAML response will be returned indicating an error. Otherwise a page for requesting the users credentials is returned. The page returned will also contain the signed JavaScript for handling the OpenNemID specific actions. The JavaScript is signed using a certificate, *IdPJsCert*, this certificate has to have been issued by the authentication provider.

A challenge (nonce) is generated, this is to prevent an attack where an adversary would submit an identical request to one that has already been submitted, also known as a *replay attack*. If the challenge wasn't introduced the second identical request would also be accepted.

A cookie is created containing the *AuthRequest*, signed *AuthRequest* along with the relay state. This is done to free the identity provider from having to store the data, thereby granting more statelessness.

**Algorithm 2** Process 3

---

**Require:** GET is well-formed **and** IdP.Private **and** SP.Public **and** IdPJsCert.Public **and** IdP has JavaScript from AuthP  
 AR  $\leftarrow$  DeflateDecompress(Base64Dec(UrlDec(MA)))  
**if** VERIFY(AR, SAR, SP.Public) **then**  
   C1  $\leftarrow$  GenChallenge()  
   JS  $\leftarrow$  GetStoredJavaScript()  
   JS.SigParams.Challenge  $\leftarrow$  C1  
   JS.SigParams.Certificate  $\leftarrow$  IdPJsCert.Public  
   JS.SigParams.Signature  $\leftarrow$  SIGN(JS.SigParams, IdPJsCert.Private)  
   ARC  $\leftarrow$  CreateCookie(MA, SAR, RS)  
   **return** RESPONSE(JS, ARC)  
**else**  
   RE  $\leftarrow$  CreateResponse()  
   RE.InResponseTo  $\leftarrow$  AR  
   RE.Status  $\leftarrow$  Requester  
   MRE  $\leftarrow$  Base64Enc(RE)  
   **return** REDIRECT(SP, MRE, RS)  
**end if**

---

As specified in the formalization, it's required for the browser to allow JavaScript. The username,  $U$  and password,  $P$  for the user is not stored and fetched directly, but will be input by the user manually, though for the description of process 4 it will be assumed that they are both ready right away. Before the user is prompted for username and password, the JavaScript is verified. The challenge is submitted together with the username and password. Hashing of the username and password has been omitted as it holds no functional enhancement towards the protocol.

**Algorithm 3** Process 4

---

**Require:** U **and** P **and** Browser allows JavaScript  
 SigParams  $\leftarrow$  Js.SigParams  
**if** VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) **then**  
   C1  $\leftarrow$  SigParams.Challenge  
   **return** POST(U, P, C1)  
**else**  
   **print** ERROR  
**end if**

---

In process 5 the identity provider confirms that the challenge received matches a previously issued challenge, and that it has not already been used. A SAML response indicating an error is returned if the challenge is not accepted, otherwise the username and password will be delegated to the authentication provider.

---

**Algorithm 4** Process 5

---

**Require:** POST is well formed  
**if** C1 matches challenge issued by IdP **and** C1 is valid **then**  
    **Delegate** U **and** P **to** AuthP  
**else**  
    **return** RESPONSE(ERROR)  
**end if**

---

Process 5.1 describes how the username and password submitted is used to identify a user at the authentication provider. If a user is found and is valid, a challenge will be generated and the next N-Factor challenge for the user will be fetched. The N-Factor challenge is signed and then returned. Were no valid user found based on the supplied username and password, a SAML response message indicating an error will be returned.

---

**Algorithm 5** Process 5.1

---

USER  $\leftarrow$  GetUser(U, P)  
**if** USER is valid **then**  
    C2  $\leftarrow$  GenChallenge()  
    NF  $\leftarrow$  GetNextNFactorChallenge(USER)  
    NF.SigParam.User  $\leftarrow$  USER  
    NF.SigParam.Challenge  $\leftarrow$  C2  
    NF.SigParam.Certificate  $\leftarrow$  USER.Public  
    NF.SigParam.Signature  $\leftarrow$  SIGN(NF.SigParam, USER.Private)  
    **return** RESPONSE(NF)  
**else**  
    **return** RESPONSE(ERROR)  
**end if**

---

Process 6 show how the identity provider has to delegate messages from the browser to the authentication provider and vice versa. In this case the N-Factor challenge is to be delegated. If the authentication provider returned an error or the identity provider cannot verify the N-Factor challenge received, a SAML response message indicating an error will be returned to the browser. If the N-Factor challenge is verified, it is then delegated to the browser.

---

**Algorithm 6** Process 6

---

SigParams  $\leftarrow$  NF.SigParams  
**if** VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) **then**  
    **Delegate** NF **to** Browser  
**else**  
    **Delegate** ERROR **to** Browser  
**end if**

---

The browser also verifies that the N-Factor challenge is valid in process

6.1. An attempt to authenticate the N-Factor challenge is then performed. Due to the challenges genericity, the authentication process will not be further described.

---

**Algorithm 7** Process 6.1

---

```

SigParams ← NF.SigParams
if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
    AUTHENTICATE(NF)
else
    print ERROR
end if

```

---

In process 6.2, the genericity of the N-Factor challenges once more results in a minimal description. The way the authentication attempts will be handled depends on the N-Factor challenge, therefore to simplify the description, it is assumed that a result conforming to a specific template is returned.

---

**Algorithm 8** Process 6.2

---

```

NFR ← NFactorResult(NF)
return RESPONSE(NFR)

```

---

Process 6.5 describes the handling of a N-Factor challenge result. First off, the N-Factor challenge is verified to make sure that it has not been altered. Then it is checked whether or not the N-Factor challenge result is acceptable for the N-Factor challenge. In the case of acceptance the user is fetched. If the user has not yet completed all N-Factor challenges required to authenticate at the authentication provider, then the next challenge is fetched, signed and returned to the user. Process 6 to 6.5 is then repeated until no more N-Factor challenges are required, in which case login data for the user is created and returned to the identity provider. In case of the N-Factor challenge not passing verification or the N-Factor challenge result is not accepted a SAML message indicating an error is returned.

**Algorithm 9** Process 6.5

---

```

Require: User identifiable by (NF.SigParams.USER,
NF.SigParams.Certificate)
SigParams  $\leftarrow$  NF.SigParams
if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
  if NFR is acceptable result of NF then
    USER  $\leftarrow$  GetUser(SigParams.USER, SigParams.Certificate)
    C2  $\leftarrow$  GenChallenge()
    if USER.HasNextChallenge then
      NF  $\leftarrow$  GetNextNFactorChallenge(USER)
      NF.SigParams.User  $\leftarrow$  USER
      NF.SigParams.Challenge  $\leftarrow$  C2
      NF.SigParams.Certificate  $\leftarrow$  USER.Public
      NF.SigParams.Signature  $\leftarrow$  SIGN(NF.SigParams, USER.Private)
      return RESPONSE(NF)
    else
      LID  $\leftarrow$  CreateLogInData(USER)
      ST  $\leftarrow$  "OK"
      return RESPONSE(ST, LID)
    end if
  else
    return RESPONSE(ERROR)
  end if
else
  return RESPONSE(ERROR)
end if

```

---

In process 7 it is described how the identity provider handles when a user has been authenticated at the authentication provider. If the returned status does not equal an acceptance criteria, the status is returned to the browser to indicate an error, the status would further describe the error. When the status is accepted the cookie created in process 3 is fetched and the contents on the cookie are extracted.

The initial AuthRequest extracted from the cookie is verified. Verification failure results in the user being redirected to the service provider with information indicating that the AuthRequest could not be granted.

When verification succeeds, an assertion is build based on the login data received from the authentication provider. The assertion is then encrypted using the service providers public key. A SAML response is created, the assertion is appended to it along with the AuthRequest. The browser is then redirected to the service provider.



**Algorithm 10** Process 7**Require:** SP.Public and LID is well-formed **and** ARC cookie present

---

```

if ST = "OK" then
  MA ← ARC.AR
  SAR ← ARC.SAR
  RS ← ARC.RS
  AR ← DeflateDecompress(Base64Dec(UrlDec(MA)))
  if VERIFY(AR, SAR, SP.Public) then
    A ← BuildAssertion(LID.Certificate)
    SI ← GenerateSessionIndex()
    A.InResponseTo ← AR
    A.Issuer ← IdP
    A.Audience ← SP
    A.SessionIndex ← SI
    A.Signature ← SIGN(A, IdP.Private)
    EA ← ENCRYPT(A, SP.Public)
    RE ← CreateResponse()
    RE.Assertion ← EA
    RE.InResponseTo ← AR
    RE.Status ← "Success"
    MRE ← DeflateCompress(Base64Enc(UrlEnc(RE)))
    SAS ← CreateSAMLSession(SI, SP, LID.CertificateSubject)
    return REDIRECT(SP, MRE, RS, SAS)
  else
    RE ← CreateResponse()
    RE.InResponseTo ← AR
    RE.Status ← "Requester"
    MRE ← DeflateCompress(Base64Enc(UrlEnc(RE)))
    return REDIRECT(SP, MRE, RS)
  end if
else
  return RESPONSE(ST)
end if

```

---

Process 9 describes the last step in the diagram. The SAML response is received at the service provider. The assertion is decrypted and verified. If the verification fails an error message is returned to the browser. If the verification succeeds, an AuthKey is generated. If the user supplies the AuthKey in subsequent requests to the service provider, the service provider will know that the user already allowed to view its resources thereby eliminating the need to authenticate again until the AuthKey expires. The initial requested resource is retrieved and returned to the browser together with the AuthKey.

---

**Algorithm 11** Process 9

---



---

**Require:** POST is well-formed **and** SP.Private **and** IdP.Public

RE  $\leftarrow$  UrlDec(Base64Dec(DeflateDecompress(MRE)))

A  $\leftarrow$  DECRYPT(RE.Assertion, SP.Private)

**if** VERIFY(A, A.Signature, IdP.Public) **then**

    AK  $\leftarrow$  GenAuthKey()

    R  $\leftarrow$  Base64Dec(UrlDec(RS))

    RES  $\leftarrow$  GetResource(R)

**return** RESPONSE(RES, AK)

**else**

**return** RESPONSE(403, Forbidden)

**end if**

---

This concludes the formalizing of the OpenNemID protocol and processing, the subsequent chapter will show how to transform this model into an executable modal using F\*.

## Chapter 4

# Modelling with F\*

Even though we have introduced F\* in chapter 2 this chapter will more detailed introduce the language F\* further. F\* can be used to model a security protocol. Despite being a formal specification language F\* is also executable. F\* is described as a *A Verifying Compiler for Distributed Programming*. This chapter will describe how we have used F\* to build a formal specification of OpenNemID.

### 4.1 Introducing F\*

F\* is a research language from Microsoft Research. F\* primarily subsumes two research languages from Microsoft Research, F7<sup>1</sup> and Fine<sup>2</sup>. F\* is at this time considered to be an  $\alpha$ -release. The purpose of designing F\* is to enable the construction and communication of proofs of program properties and of properties of a program's environment in a verifiable secure way. F\* is a dialect of ML and compiles to .NET bytecode in type-preserving style. This means that it can inter op with other .NET languages and the types defined in F\* can be used by other .NET languages without losing type information. Furthermore there also exists a fully abstract compiler from F\* to JavaScript. This makes it possible to deploy F\* programs on web pages as JavaScript meanwhile there is a formal guarantee that the program still behaves just as they would according to F\* semantics. The compiling and type-checking of F\* code utilizes the Z3<sup>3</sup> SMT solver for proving assumptions made with refinement types. F\* has been formalized and verified using Coq<sup>4</sup>.

---

<sup>1</sup><http://research.microsoft.com/en-us/projects/f7/>

<sup>2</sup><http://research.microsoft.com/en-us/projects/fine/>

<sup>3</sup><http://z3.codeplex.com/>

<sup>4</sup>Coq is an interactive theorem prover written in OCaml

## 4.2 Syntax and semantics

F\* inherits syntax and semantics from ML. F\* is a functional language which means that it has features like immutability by default, polymorphic types and type inference. In Listing 4.1 we have shown the classic Hello World example in F\*. This is the simplest way this example could have been written. This example shows how to specify a main method by defining a function `_` (underscore) at the end of a module. This will instruct the compiler to make an `.exe` file instead of a `.dll`.

```

1  module HelloWorld
2
3  let _ = print "Hello world!"

```

Listing 4.1: Hello World example in F\*

The example in listing 4.2 shows how to explicitly specify types with the *colon* operator and the *val* declaration for defining function signatures. This example defines the function `multiply` that takes two ints as parameters and returns an int. After that it defines the corresponding *let* binding which defines `multiply` the 2 arguments. It is important to note that not defining the corresponding *let* binding will not cause the compilation to fail but give the following warning: *Warning: Admitting value declaration Multiplication.multiply as an axiom without a corresponding definition.* So the value declaration was valid but there is no concrete implementation supporting this claim.

```

1  module Multiplication
2
3  val multiply: x:int -> y: int -> int
4
5  let multiply x y = x * y
6
7  let mul = multiply 3 4

```

Listing 4.2: Multiplication example in F\*

## 4.3 Refinement types

F\* has derived the feature refinement types from the Microsoft Research projects, F7 and Fine. Refinement types are used to make type safe refinements of existing types. Thus it is possible to restrict or refine values more than their original type. Listing 4.3 shows an example with the refinement *nat* of *int* that states that *nat* will always be zero or larger, i.e. a natural number. The example also shows an attempt to assign -1 to a type of *nat* which will fail type checking.

```

1  (*Declare a type nat of natural numbers*)
2  type nat = i:int{0 <= i}
3
4  let x:nat = 1
5  let y:nat = 0

```

```
6 let z:nat = 1 - 2 (*Will fail type check*)
```

Listing 4.3: Simple refinement types in F\*

Refinement types have the form  $x : t\{t'\}$  as shown above. So a refinement type is created by taking an existing type and decorate it with an expression in curly brackets. In the example above the refinement type is a simple boolean expression but refinements are not limited to boolean expression. This is extended in F\* by its kind-system. Kinds can be seen as an abstraction over types - types can either *have* or be *of* a kind. The \*-kind indicates 'regular types' in F\*. This covers all the possible types to create in a regular type system for a programming language like Java. Refinement types are of the E-kind and not of the \*-kind. The E(rasable)-kinds have no significance at runtime. They only have an effect at the compiling time during type checking.

## 4.4 Formalizing OpenNemID by using Jacob's work - THIS IS A BAD TITLE!

Since we are extending Jacob's work with the authentication part of the protocol we used his code as a reference for implementing the rest of the OpenNemID protocol. In listing 4.4 we show Jacob's implementation of the Identity Provider. He has implemented a recursive function *identityprovider* declared with the *val* binding just above it. The function declared takes 2 arguments and returns *unit*, which is the same as void in Java and C#.

### The arguments

1. a principal for identifying the identity provider
2. a principal for identifying the client.

```
1 module Identityprovider
2
3 open SamlProtocol
4 open Crypto
5
6 let handleUserAuthenticated me user client authnrequest =
7   match authnrequest with
8   | MkAuthnRequest(reqid,issueinst,dest,sp,msg,sigSP) ->
9     let pubksp = CertStore.GetPublicKey sp in
10
11     if (VerifySignature sp pubksp msg sigSP) then
12       (assert (Log sp msg));
13       let assertion = IssueAssertion me user sp reqid in
14       let myprivk = CertStore.GetPrivateKey me in
15       assume(Log me assertion);
16       let sigAs = Sign me myprivk assertion in
17       let signAssertion = AddSignatureToAssertion assertion
18         sigAs in
19       let encryptedAssertion = EncryptAssertion sp pubksp
20         signAssertion in
```

```

20         let resp = AuthResponseMessage me sp encryptedAssertion
21         in
22         SendSaml client resp) (*10*)
23     else
24         SendSaml client (Failed Requester)(*10.2*)
25
26 val identityprovider: me:prin -> client:prin -> unit
27
28 let rec identityprovider me client =
29     let req = ReceiveSaml client in (*3 & 11*)
30     match req with
31     | AuthnRequestMessage (issuer, destination, message, sigSP) ->
32         let pubkissuer = CertStore.GetPublicKey issuer in
33         if (VerifySignature issuer pubkissuer message sigSP) then
34             (assert (Log issuer message);
35              let challenge = GenerateNonce me in
36              let resp = UserCredRequest challenge in
37              SendSaml client resp; (*4*)
38              identityprovider me client (*Start over*))
39         else
40             SendSaml client (Failed Requester);(*4.1*)
41             identityprovider me client (*Start over*)
42
43     | UserAuthenticated (status, logindata, authnrequest) ->
44         match logindata with
45         | MkLoginData (user,sig,cert,challenge,site,data) ->
46             if (status = "OK") && (VerifySignature user cert data sig
47              ) then
48                 (assert (Log user data);
49                  handleUserAuthenticated me user client authnrequest;
50                  identityprovider me client (*Start over*))
51             )
52         else
53             SendSaml client (DisplayError 400);(*10.1*)
54             identityprovider me client (*Start over*)
55
56     | _ -> SendSaml client (DisplayError 400);(*10.1*)
57             identityprovider me client (*Start over*)

```

Listing 4.4: NemID identity provider implementation [1]

The recursive function *identityprovider* starts by receiving a SAML message from the client. It then matches the request with a *SamlMessage.AuthnRequestMessage* or *SamlMessage.UserAuthenticated* type. When matched with an *AuthnRequestMessage* message it verifies the Service Provider's signature of the message by the function *VerifySignature* which is shown in listing 4.5. The function takes a principal, the principals public key, a message and a signature. It returns a boolean indicating if the check passed. The return type however has a refinement type that relates the message to the principal if the verification passes.  $\Rightarrow$  should be as implication therefore stating that the predicate is valid. If the verification of the Service Provider's signature of the *AuthnRequestMessage* passes it creates a nonce to be related to this user when the user has authenticated himself/herself by NemID and sends the response to the user. When the user has authenticated through NemID the function *handleUserAuthenticated* is called. The function's purpose is to issue a signed assertion and sending the *AuthResponseMessage* to

the user. The signature for signing messages takes 4 arguments - the principal, the signer, the private key of the principal and the message to be signed. The message is annotated with a refinement type  $\{Log\ p\ msg\}$ . This refinement type is an E-kinded type that takes a principal and a string as constructor elements. The val declaration for a method Sign in listing 4.5 requires the predicate  $\{Log\ p\ msg\}$  to be true before it can typecheck. This means that the message to sign is related to the principal signing the message. Securing this is done by calling *assume* $\{Log\ me\ assertion\}$  before signing the message. The predicate is by virtue of this "verified". After this the assertion is encrypted by using the Service Provider's public key and sent within an AuthResponseMessage to the user.

```

1  type pubkey :: prin => *
2  type privkey :: prin => *
3
4  type Log :: prin => string => E
5
6  val Sign: p:prin
7    -> privkey p
8    -> msg:string{Log p msg}
9    -> dsig
10
11 val VerifySignature: p:prin
12   -> pubkey p
13   -> msg:string
14   -> dsig
15   -> b:bool{b=true ==> Log p msg}

```

Listing 4.5: Cryptographic elements

In listing 4.5 we show the declaration of the types for private key (*privkey*) and public key (*pubkey*). These types are declared by using the F\* syntax for constructing dependent types (the double colon). This means that a type *pubkey* will have a constructor that takes a *prin* (principal) and returns a type of \*-kind. This is still abstract and the type has no actual constructor.

## 4.5 OpenNemID specified in F\*

The code in this section represents the state of the project now. This is in no way a complete implementation of the protocol. Implementation was carried out in an incremental manner. First the focus was on understanding Jacob's work and expanding that with the authentication part (Authentication Provider) of the protocol, which before was done by NemID, and then adding the functionality of creating login, establishing connection between Identity Provider and the Authentication Provider and so on. All source code that has been produced in this project can be found on the source code sharing community Github<sup>5</sup>. The F\* code for the protocol is organized in 10 modules:

1. The TypeFunc module

<sup>5</sup><https://github.com/kiniiry-supervision/OpenNemID>

2. The SamlProtocol module
3. The Crypto module
4. The CertStore module
5. The Messaging module
6. The Service Provider module
7. The Identity Provider module
8. The Database module
9. The Authentication Provider module
10. The Browser module

The modeling follows the principles for cryptographic protocol modeling outlined by Dolev & Yao<sup>6</sup>. In the following we will explain the important principles for each module and the relation to the algorithms outlined in chapter 3.

#### 4.5.1 Specification of the type functionality module

```

1  module TypeFunc
2
3  type Authentication =
4    | Facebook: id:int -> Authentication
5    | SMS: generated:int -> Authentication
6    | Google: id:int -> Authentication
7    | OpenId: id:int -> Authentication

```

Listing 4.6: TypeFunc module

The *TypeFunc* module provides the type authentication which is used for the different kinds of n factor authentication. Note that currently there is only an id associate with each type of authentication for simplicity. This needs to be modified so that each type is more explicit and holds the correct information for authentication.

#### 4.5.2 Specification of the SAML Protocol

```

1  module SamlProtocol
2
3  open Crypto
4  open TypeFunc
5
6  type assertiontoken = string (*Add refinements*)
7  type signedtoken = string (*Add refinements*)

```

<sup>6</sup>Cryptographic primitives are assumed perfect and cyphers cannot be decrypted without the the proper decryption key



```

8  type id = string
9  type endpoint = string
10 type uri = string
11
12
13 type AuthnRequest =
14   | MkAuthnRequest: IssueInstant:string ->
15     Destination:endpoint -> Issuer:prin ->
16     message:string -> sig:dsig ->
17     AuthnRequest
18
19 type LoginData =
20   | MkLoginData: user:prin -> signature:dsig ->
21     cert:pubkey user -> challenge:nonce ->
22     site:string -> data:string ->
23     LoginData
24
25 type LoginInfo =
26   | UserLogin: userid:string -> password:string ->
27     LoginInfo
28
29 type AuthInfo =
30   | UserAuth: userid:string -> authmethod:Authentication ->
31     authresponse:Authentication -> AuthInfo
32
33 type Assertion =
34   | SignedAssertion: assertiontoken -> dsig -> Assertion
35   | EncryptedAssertion: cypher -> Assertion
36
37 type SamlStatus =
38   | Success: SamlStatus
39   | Requester: SamlStatus
40   | Responder: SamlStatus
41
42 type LoginError =
43   | AuthError: LoginError
44   | CredentialError: LoginError
45
46 type SamlMessage =
47   | SPLogin: uri -> SamlMessage
48   | Login: loginInfo:LoginInfo -> challenge:nonce ->
49     SamlMessage
50   | LoginResponse: string -> SamlMessage
51   | AuthnRequestMessage: issuer:prin -> destination:endpoint
52     -> message:string -> dsig -> SamlMessage
53   | LoginRequestMessage: issuer:prin -> destination:endpoint
54     -> loginInfo:LoginInfo -> SamlMessage
55   | NfactAuthRequest: issuer:prin -> destination:endpoint ->
56     authInfo:AuthInfo -> challenge:nonce -> dsig ->
57     SamlMessage
58   | AuthResponseMessage: issuer:prin -> destination:endpoint ->
59     Assertion -> SamlMessage
60   | LoginResponseMessage: issuer:prin -> destination:endpoint
61     -> auth:Authentication -> challenge:nonce -> dsig ->
62     SamlMessage
63   | UserAuthenticated: status:string -> logindata:LoginData ->
64     authnReq:AuthnRequest -> SamlMessage

```

```

56 | UserCredRequest: javascript:string -> challenge:nonce ->
    dsig -> SamlMessage
57 | UserAuthRequest: authmethod:Authentication -> challenge:
    nonce -> dsig -> SamlMessage
58 | UserAuthResponse: authInfo:AuthInfo -> challenge:nonce ->
    dsig -> SamlMessage
59 | LoginSuccess: status:string -> issuer:prin -> destination:
    endpoint -> SamlMessage
60 | Failed: SamlStatus -> SamlMessage
61 | LoginFailure: LoginError -> SamlMessage
62 | DisplayError: int -> SamlMessage
63
64
65 val SendSaml: prin -> SamlMessage -> unit
66 val ReceiveSaml: prin -> SamlMessage
67
68 val CreateAuthnRequestMessage: issuer:prin -> destination:prin
    -> string
69 val IssueAssertion: issuer:prin -> subject:prin -> audience:
    prin -> inresto:AuthnRequest -> assertiontoken
70 val AddSignatureToAssertion: assertiontoken -> dsig ->
    signedtoken
71 val EncryptAssertion: receiver:prin -> pubkey receiver ->
    signedtoken -> Assertion
72 val DecryptAssertion: receiver:prin -> privkey receiver ->
    Assertion -> (signedtoken * dsig)

```

Listing 4.7: Specification of the SAML Protocol elements

The *SamlProtocol* module is taken directly from Jacob’s code and only modified to support more and different *SamlMessage* that are needed in our specification of OpenNemID. This module’s purpose is the specification of messages and to provide functions for sending and receiving messages. Note that the functions for sending and receiving messages have no runtime implementation. They are only specified by the *val* declaration. The SAML Protocol is used for the communication between the principals in the OpenNemID protocol in a login session. The intention with these functions is that they will handle the mapping of protocol elements to the network.

### 4.5.3 Specification of cryptographic elements

```

1  module Crypto
2
3  open Protocol
4  open TypeFunc
5
6  type prin = string
7  type pubkey :: prin => *
8  type privkey :: prin => *
9  type dsig
10 type nonce = string
11 type cypher
12
13 (*Verification*)

```

```

14 type Log :: prin => string => E
15
16 type LogAuth :: prin => Authentication => E
17
18 val Keygen: p:prin
19   -> (pubkey p * privkey p)
20
21 val Sign: p:prin
22   -> privkey p
23   -> msg:string{Log p msg}
24   -> dsig
25
26 val SignAuth: p:prin
27   -> privkey p
28   -> msg:Authentication{LogAuth p msg}
29   -> dsig
30
31 val VerifySignature: p:prin
32   -> pubkey p
33   -> msg:string
34   -> dsig
35   -> b:bool{b=true ==> Log p msg}
36
37 val VerifySignatureAuth: p:prin
38   -> pubkey p
39   -> msg:Authentication
40   -> dsig
41   -> b:bool{b=true ==> LogAuth p msg}
42
43 val Encrypt: p:prin
44   -> pubkey p
45   -> string
46   -> cypher
47
48 val Decrypt: p:prin
49   -> privkey p
50   -> cypher
51   -> string
52
53 val GenerateNonce: prin -> nonce (*Add refinement to ensure
    uniqueness*)

```

Listing 4.8: Specification of cryptographic elements

The *crypto* module is taken directly from Jacob’s code and only modified to support signing and verification of the authentication type. The purpose of the *crypto* is providing the cryptographic functions to sign and verify both messages and the authentication type also the encryption and decryption of messages. The *crypto* module utilizes the refinement type to ensure that signed messages and authentications have typed dependency to the signing principal. It does not have a concrete implementation as of now.

#### 4.5.4 Specification of certificate store module

```

1  module CertStore
2
3  open Crypto
4
5  val GetPublicKey: p:prin -> pubkey p
6  val GetJSPublicKey: p:prin -> pubkey p
7  (*Prin needs to be updated to include credentials*)
8  val GetPrivateKey: p:prin -> privkey p
9  val GetJSPrivateKey: p:prin -> privkey p

```

Listing 4.9: Abstract certificate store

The *CertStore* module is taken from Jacob's code and expanded with functionality to support JavaScript public and private keys. This module provides four abstract functions for retrieving certificates from a certificate store. The functions use the value dependent syntax for relating a principal to the certificate keys. As Jacob has written in a comment the principal could be updated to include credentials because this is a quite naive implementation. It is quite naive because all you need to obtain the private key of a principal is the name of the principal.

#### 4.5.5 Specification of the messaging protocol

```

1  module Messaging
2
3  open Crypto
4  open TypeFunc
5
6  type Status =
7  | Successful: Status
8  | Unsuccessful: Status
9
10 type Message =
11 | NewSiteRequest: idp:prin -> Message
12 | ChallengeResponse: challenge:nonce -> Message
13 | IdpChalResponse: challenge:nonce -> Message
14 | AcceptedIdp: idp:prin -> pubkey:pubkey idp -> authp:prin ->
    authpubkey:pubkey authp -> signedjavascript:string ->
    Message
15 | RequestForLogin: userid:string -> password:string -> email:
    string -> Message
16 | ReqLoginResponse: challenge:nonce -> Message
17 | CreateLogin: generatedpassword:string -> challenge:nonce ->
    Message
18 | ChangeUserId: userid:string -> newUserId:string -> password:
    string -> Message
19 | ChangePassword: userid:string -> password:string ->
    newPassword:string -> Message
20 | UserRevokeIdp: userid:string -> password:string -> idp:
    string -> Message
21 | AddNfactor: userid:string -> password:string -> nfact:
    Authentication -> Message
22 | RemoveNfactor: userid:string -> password:string -> nfact:
    Authentication -> Message

```

```

23 | StatusMessage: Status -> Message
24
25
26 val SendMessage: prin -> Message -> unit
27 val ReceiveMessage: prin -> Message

```

Listing 4.10: Specification of the Messaging protocol

The *Messaging* module is responsible for 2 things - the specification of messages and providing functions for sending and receiving these messages. As the *Saml-Protocol* module the functions for sending and receiving are specified only by the *val* declaration and has no concrete runtime implementation. This module is used to model the communication between Identity Provider / user and the Authentication Provider when wanting to establish a secure connection and creating and/or changing an user's login.

#### 4.5.6 Specification of the Service Provider

```

1  module Serviceprovider
2
3  open SamlProtocol
4  open Crypto
5
6  val serviceprovider: me:prin -> client:prin -> idp:prin ->
    unit
7
8  let rec serviceprovider me client idp =
9    let req = ReceiveSaml client in
10   match req with
11   | SPLLogin (url) ->
12     let authnReq = CreateAuthnRequestMessage me idp in
13     assume(Log me authnReq);
14     let myprivk = CertStore.GetPrivateKey me in
15     let sigSP = Sign me myprivk authnReq in
16     let resp = AuthnRequestMessage me idp authnReq sigSP in
17     SendSaml client resp;
18     serviceprovider me client idp
19   | AuthResponseMessage (issuer, destination, encassertion) ->
20     let myprivk = CertStore.GetPrivateKey me in
21     let assertion = DecryptAssertion me myprivk encassertion in
22     match assertion with
23     | SignedAssertion (token, sigIDP) ->
24       let pubkissuer = CertStore.GetPublicKey idp in
25       if VerifySignature idp pubkissuer token sigIDP
26       then
27         (assert(Log idp token);
28          let resp = LoginResponse "You are now logged in" in
29          SendSaml client resp)
30       else SendSaml client (DisplayError 403);
31     serviceprovider me client idp
32
33   | _ -> SendSaml client (DisplayError 400);
34     serviceprovider me client idp

```

Listing 4.11: Specification of service provider

The service provider is taken and directly from Jacob's code and it is not modified in any way. The service provider implements algorithm 1 and 11 in section 3.4. The module is constructed to accept SAML messages of type *SPLogin* and *AuthResponseMessage*. If the service provider receives another type of message it will return a HTTP error. Contrary to algorithms 1 and 11 the service provider does not implement encoding and decoding because this is expected to be handled by the *SamlProtocol* module.

### 4.5.7 Specification of the Identity Provider

The specification of the Identity Provider is divided into several listings for the sake of understandability.

```

1  module Identityprovider
2
3  open SamlProtocol
4  open Crypto
5  open TypeFunc
6  open Messaging
7
8  val userloggedin: user:prin -> bool
9  val getjavascript: string
10 val userlogin: user:prin -> unit
11 val decodeMessage: message:string -> AuthnRequest
12 val getauthnrequest: user:prin -> challenge:nonce ->
    AuthnRequest
13 val getuserchallenge: user:prin -> nonce
14 val relatechallenge: user:prin -> challenge:nonce -> unit
15 val verifychallenge: user:prin -> challenge:nonce -> bool
16 val relate: user:prin -> challenge:nonce -> authnReq:
    AuthnRequest -> unit
17
18 val identityprovider: me:prin -> user:prin -> authp:prin ->
    unit
19
20 let rec identityprovider me user authp =
21   let request = ReceiveSaml user in
22   match request with
23   | AuthnRequestMessage(issuer, destination, message, sigSP) ->
24     let pubkissuer = CertStore.GetPublicKey issuer in
25     if (VerifySignature issuer pubkissuer message sigSP) then
26       (assert (Log issuer message);
27        let authnReq = decodeMessage message in
28        let myprivk = CertStore.GetPrivateKey me in
29        if not (userloggedin user) then
30          let challenge = GenerateNonce me in
31          relate user challenge authnReq;
32          relatechallenge user challenge;
33          let js = getjavascript in
34          assume(Log me js);
35          let myprivk = CertStore.GetJSPrivateKey me in
36          let sigIdP = Sign me myprivk js in
37          let resp = UserCredRequest js challenge sigIdP in
38          SendSaml user resp;
39          identityprovider me user authp

```

```

40   else
41     let assertion = IssueAssertion me user issuer authnReq in
42     assume(Log me assertion);
43     let myprivk = CertStore.GetPrivateKey me in
44     let pubksp = CertStore.GetPublicKey issuer in
45     let sigAs = Sign me myprivk assertion in
46     let signAssertion = AddSignatureToAssertion assertion sigAs
47     in
48     let encryptedAssertion = EncryptAssertion issuer pubksp
49     signAssertion in
50     let resp = AuthResponseMessage me issuer encryptedAssertion
51     in
52     SendSaml user resp)
53   else
54     SendSaml user (Failed Requester);
55     identityprovider me user authp
56   | Login (loginInfo, challenge) ->
57     if (verifychallenge user challenge) then
58       let req = LoginRequestMessage me authp loginInfo in
59       SendSaml authp req;
60       handleauthresponse me user authp;
61       identityprovider me user authp
62     else
63       SendSaml user (DisplayError 400);
64       identityprovider me user authp
65   | UserAuthResponse(authInfo, challenge, sigAuth) ->
66     let req = NfactAuthRequest me authp authInfo challenge
67     sigAuth in
68     SendSaml authp req;
69     handleauthresponse me user authp;
70     identityprovider me user authp
71   | _ -> SendSaml user (DisplayError 400);
72     identityprovider me user authp

```

Listing 4.12: Handling and delegation of a user's requests

This part of the identity provider implements the algorithms (INDST NR p algoritmerne). The identity provider accepts the three SAML messages *AuthnRequestMessage*, *Login* and *UserAuthResponse* from the user.

1. The *AuthnRequestMessage* branch decodes the message and if the user has not logged in previously it sends a *UserCredRequest* back with the JavaScript and a nonce to be used for relating the login at the Identity Provider prompting the user to give his or her login information. If the user has already logged in previously it issues an assertion to the user.
2. The *Login* branch handles the user's login information which is the response the user provides after receiving the *UserCredRequest*. This branch verifies that the nonce from the user is the correct one and if it is correct it delegates the login information to the Authentication Provider and then calls the function *handleauthresponse* which we will explain later in this section. If the nonce is incorrect it returns a HTTP error.
3. The *UserAuthResponse* branch handles the user's n factor authentication information and delegates the information to the Authentication Provider.

```

1  val handleUserAuthenticated: me:prin -> user:prin -> authnReq:
    AuthnRequest -> unit
2
3  let handleUserAuthenticated me user authnReq =
4    match authnReq with
5    | MkAuthnRequest(issueinst,dest,sp,msg,sigSP) ->
6      let pubksp = CertStore.GetPublicKey sp in
7      if (VerifySignature sp pubksp msg sigSP) then
8        (assert (Log sp msg);
9         let assertion = IssueAssertion me user sp authnReq in
10        let myprivk = CertStore.GetPrivateKey me in
11        assume(Log me assertion);
12        userlogin user;
13        let sigAs = Sign me myprivk assertion in
14        let signAssertion = AddSignatureToAssertion assertion sigAs
            in
15        let encryptedAssertion = EncryptAssertion sp pubksp
            signAssertion in
16        let resp = AuthResponseMessage me sp encryptedAssertion in
17        SendSaml user resp)
18      else
19        SendSaml user (Failed Requester)
20  val handleauthresponse: me:prin -> user:prin -> authp:prin ->
    unit
21
22  let handleauthresponse me user authp =
23    let resp = ReceiveSaml authp in
24    match resp with
25    | LoginResponseMessage(issuer, destination, authmethod,
        challenge, sigUser) ->
26      let pubkeyuser = CertStore.GetPublicKey user in
27      if VerifySignatureAuth user pubkeyuser authmethod sigUser
          then
28        (assert (LogAuth user authmethod);
29         relatechallenge user challenge;
30         let resp = UserAuthRequest authmethod challenge sigUser in
31         SendSaml user resp)
32      else
33        SendSaml user (DisplayError 403)
34    | LoginSuccess(status, issuer, destination) ->
35      if (status = "OK") then
36        let challenge = getuserchallenge user in
37        let authnReq = getauthnrequest user challenge in
38        handleUserAuthenticated me user authnReq
39      else
40        SendSaml user (DisplayError 403)
41    | _ -> SendSaml user (DisplayError 400)

```

Listing 4.13: The handling of the responses from Authentication Provider

This part of the identity provider handles the information received from the Authentication Provider. It has two match branches:

1. *LoginResponseMessage* which will prompt the user for a n factor authentication method while it relates the challenge generated by the Authentication Provider to the user for verification



2. *LoginSuccess* which specifies that the user has passed all the n factor authentication methods.

If the user has been successfully logged in the user will be issued an assertion which is done in the *handleUserAuthenticated* function. This function will also save a cookie that specifies that this user has logged in which the Identity Provider will search for when getting an *AuthnRequestMessage* from a user.

```

1  val savejavascript: javascript:string -> unit
2  val savepublickey: owner:prin -> publickey:pubkey owner -> unit
3
4  val connectwithauthp: me:prin -> authp:prin -> unit
5
6  let connectwithauthp me authp =
7    let req = NewSiteRequest me in
8    let _ = SendMessage authp req in
9    let resp = ReceiveMessage authp in
10   match resp with
11   | ChallengeResponse(challenge) ->
12     let _ = SendMessage authp (IdpChalResponse challenge) in
13     let res = ReceiveMessage authp in
14     match res with
15     | AcceptedIdp(idp, idppubkey, authp, authppubkey, signedjs)
16       ->
17       (*Established secure connection*)
18       savejavascript signedjs;
19       savepublickey authp authppubkey;
20       savepublickey idp idppubkey
21   | _ -> res; ()
22   | _ -> resp; ()

```

Listing 4.14: Establishing a secure connection with Authentication Provider

This part of the Identity Provider is the establishing of the secure connection between the Identity Provider and the Authentication Provider. Right now the challenge response from the Authentication Provider is just a nonce to illustrate that the Identity Provider needs to be investigated thoroughly by the Authentication Provider for the purpose of finding out if it is a non-evil Identity Provider.

#### 4.5.8 Specification of the Database Handler

```

1  module Database
2
3  open Crypto
4  open CertStore
5  open TypeFunc
6
7  (*Identity provider functionality*)
8  val whitelist: idp:prin -> unit
9  val blacklist: idp:prin -> unit
10 val addidp: idp:prin -> bool
11 val whitelisted: idp:prin -> bool
12

```

```

13  (*User functionality*)
14  val createuser: user:prin -> userid:string -> password:string
    -> bool
15  val usercreation: user:prin -> generatedPassword:string -> bool
16  val changeuserid: user:string -> newuser:string -> password:
    string -> bool
17  val changeuserpassword: user:string -> password:string ->
    newpassword:string -> bool
18
19  val addnfactor: user:string -> password:string -> nfactor:
    Authentication -> bool
20  val removenfactor: user:string -> password:string -> nfactor:
    Authentication -> bool
21
22  val getnfactor: user:string -> Authentication
23  val checknfactor: user:string -> Authentication -> bool
24  val allnfactauthed: user:string -> bool
25  val resetnfact: user:string -> unit
26
27  val checklogin: user:string -> password:string -> bool
28
29  val revokeidp: user:string -> password:string -> idp:string ->
    bool
30
31  val revokedidp: user:string -> idp:prin -> bool

```

Listing 4.15: Specification of the database

The *Database* module is responsible for the communication with the database and therefore checking the information provided by the user. The database is also responsible for keeping track of how many n factor authentications the user has gone through. Note that as of now these functions are just specified by the *val* declaration and therefore has no concrete implementation.

#### 4.5.9 Specification of the Authentication Provider

The specification of the Authentication Provider is divided into several listings for the sake of understandability.

```

1  module Authenticationprovider
2
3  open SamlProtocol
4  open Crypto
5  open Database
6  open TypeFunc
7  open Messaging
8
9  val relatechallenge: user:prin -> challenge:nonce -> unit
10
11  val verifychallenge: user:prin -> challenge:nonce -> bool
12
13  val nfactauth: me:prin -> idp:prin -> user:prin -> userid:
    string -> unit
14
15  let nfactauth me idp user userid =

```

```

16 if (allnfactauthed userid) then
17   resetnfact userid;
18   let status = "OK" in
19   let resp = LoginSuccess status me idp in
20   SendSaml idp resp
21 else
22   let challenge = GenerateNonce me in
23   let authmethod = getnfactor userid in
24   assume(LogAuth user authmethod);
25   let userprivkey = CertStore.GetPrivateKey user in
26   let sigUser = SignAuth user userprivkey authmethod in
27   let resp = LoginResponseMessage me idp authmethod challenge
28     sigUser in
29   SendSaml idp resp
30
31 val authenticationprovider: me:prin -> idp:prin -> user:prin ->
32   unit
33
34 let rec authenticationprovider me idp user =
35   let req = ReceiveSaml idp in
36   match req with
37   | LoginRequestMessage (issuer, destination, loginInfo) ->
38     if (whitelisted idp) then
39       match loginInfo with
40       | UserLogin(userid, password) ->
41         if not (revokedidp userid idp) && (checklogin userid
42           password) then
43           let challenge = GenerateNonce me in
44           let authmethod = getnfactor userid in
45           assume(LogAuth user authmethod);
46           let userprivkey = CertStore.GetPrivateKey user in
47           let sigUser = SignAuth user userprivkey authmethod in
48           relatechallenge user challenge;
49           let resp = LoginResponseMessage me idp authmethod
50             challenge sigUser in
51           SendSaml idp resp;
52           authenticationprovider me idp user
53         else
54           SendSaml idp (LoginFailure CredentialError);
55           authenticationprovider me idp user
56       | _ -> SendSaml idp (Failed Requester);
57       authenticationprovider me idp user
58   else
59     SendSaml idp (Failed Requester);
60     authenticationprovider me idp user
61   | NfactAuthRequest(issuer, destination, authInfo, challenge,
62     sigAuth) ->
63     if (whitelisted idp) then
64       match authInfo with
65       | UserAuth(userid, authmethod, authresponse) ->
66         let userpubkey = CertStore.GetPublicKey user in
67         if VerifySignatureAuth user userpubkey authmethod sigAuth
68           && verifychallenge user challenge then
69           if not (revokedidp userid idp) && (checknfactor userid
70             authresponse) then
71             nfactauth me idp user userid;
72             authenticationprovider me idp user

```

```

66     else
67       SendSaml idp (LoginFailure AuthError);
68       authenticationprovider me idp user
69     else
70       SendSaml idp (LoginFailure AuthError);
71       authenticationprovider me idp user
72   | _ -> SendSaml idp (Failed Requester);
73       authenticationprovider me idp user
74   else
75     SendSaml idp (Failed Requester);
76     authenticationprovider me idp user
77   | _ -> SendSaml idp (Failed Requester);
78     authenticationprovider me idp user

```

Listing 4.16: Specification of the authentication provider

The Authentication Provider implements algorithms ..... . The Authentication Provider accepts two SAML messages *LoginRequestMessage* and *NfactAuthRequest* from the Identity Provider.

1. The *LoginRequestMessage* branch will check the login information. If the correct login information has been provided by the user it generates a nonce to be related to this user and specifies which type of n factor authentication the user has to go through and that will be sent to the Identity Provider.
2. The *NfactAuthRequest* branch is the receiving of n factor authentication response from the user. It verifies that the sender of the message is the correct user and the n factor information. The correct n factor information will make the function *nfactauth* which will handle if the user has gone through all n factor authentication method or needs to specify more. The function will the information to the Identity Provider.

```

1  val usercommunication: me:prin -> user:prin -> unit
2
3  let rec usercommunication me user =
4    let req = ReceiveMessage user in
5    match req with
6    | RequestForLogin(userid, password, email) ->
7      if createuser user userid password email then
8        let challenge = GenerateNonce me in
9        relatechallenge user challenge;
10       SendMessage user (ReqLoginResponse challenge);
11       usercommunication me user
12    else
13       SendMessage user (StatusMessage Unsuccessful);
14       usercommunication me user
15    | CreateLogin(generatedpassword, challenge) ->
16      if (verifychallenge user challenge) && (usercreation user
17        generatedpassword) then
18        let challenge = GenerateNonce me in
19        relatechallenge user challenge;
20        SendMessage user (StatusMessage Successful);
21        usercommunication me user

```

```

21   else
22     SendMessage user (StatusMessage Unsuccessful);
23     usercommunication me user
24   | ChangePassword(userid, password, newPassword) ->
25     if changeuserpassword userid password newPassword then
26       SendMessage user (StatusMessage Successful);
27       usercommunication me user
28   else
29     SendMessage user (StatusMessage Unsuccessful);
30     usercommunication me user
31   | ChangeUserId(userid, newUserId, password) ->
32     if changeuserid userid newUserId password then
33       SendMessage user (StatusMessage Successful);
34       usercommunication me user
35   else
36     SendMessage user (StatusMessage Unsuccessful);
37     usercommunication me user
38   | UserRevokeIdp(userid, password, idp) ->
39     if revokeidp userid password idp then
40       SendMessage user (StatusMessage Successful);
41       usercommunication me user
42   else
43     SendMessage user (StatusMessage Unsuccessful);
44     usercommunication me user
45   | AddNfactor(userid, password, nfact) ->
46     if addnfactor userid password nfact then
47       SendMessage user (StatusMessage Successful);
48       usercommunication me user
49   else
50     SendMessage user (StatusMessage Unsuccessful);
51     usercommunication me user
52   | RemoveNfactor(userid, password, nfact) ->
53     if removenfactor userid password nfact then
54       SendMessage user (StatusMessage Successful);
55       usercommunication me user
56   else
57     SendMessage user (StatusMessage Unsuccessful);
58     usercommunication me user
59   | _ -> SendMessage user (StatusMessage Unsuccessful);
60     usercommunication me user

```

Listing 4.17: The creation and changing of a user's account

This part of the Authentication Provider is responsible for creating and changing a user's account. It is pretty intuitive what the different messages does. When creating a login the user will give an email account where they will receive an email with a one-time password to verify their account. The database will handle all the information and the checking of the information.

```

1   val getsignedjavascript: string
2
3   val establishidp: me:prin -> idp:prin -> unit
4
5   let rec establishidp me idp =
6     let req = ReceiveMessage idp in
7     match req with
8     | NewSiteRequest(idp) ->

```

```

9   let challenge = GenerateNonce me in
10  relatechallenge idp challenge;
11  SendMessage idp (ChallengeResponse challenge);
12  establishidp me idp
13  | IdpChalResponse(challenge) ->
14    if (verifychallenge idp challenge) && (addidp idp) then
15      let idppubkey = CertStore.GetPublicKey idp in
16      let mypubk = CertStore.GetPublicKey me in
17      let signedjs = getsignedjavascript in
18      let resp = AcceptedIdp idp idppubkey me mypubk signedjs in
19      SendMessage idp resp;
20      establishidp me idp
21    else
22      SendMessage idp (StatusMessage Unsuccessful);
23      establishidp me idp
24  | _ -> SendMessage idp (StatusMessage Unsuccessful);
25  establishidp me idp

```

Listing 4.18: Established a secure connection with the Identity Provider

This part of the Authentication Provider will handle the establishing of a secure connection between the Identity Provider and the Authentication Provider. As we mentioned when we described the Identity Provider’s specification of this model there needs to be some investigation of the Identity Provider and not just a generated nonce. This is just specified to give an idea of how the model is designed.

#### 4.5.10 Specification of the Browser

The specification of the Browser is divided into two listings for the sake of understandability. Note that we can not model the user’s input therefore the input from the user is specified by a bunch of *val* declarations.

```

1   module Browser
2
3   open SamlProtocol
4   open Crypto
5   open CertStore
6   open TypeFunc
7   open Messaging
8
9   val loginWithFb: Authentication
10  val loginWithGoogle: Authentication
11  val loginWithSMS: Authentication
12  val loginWithOpenId: Authentication
13  val userid: string
14  val password: string
15  val email: string
16  val fakeprint: str:string -> unit
17
18  val handleAuthMethod: auth:Authentication -> Authentication
19
20  let handleAuthMethod auth =
21    match auth with
22    | Facebook(fbid) -> loginWithFb

```

```

23 | Google(gid) -> loginWithGoogle
24 | SMS(gen) -> loginWithSMS
25 | OpenId(oid) -> loginWithOpenId
26
27 val loop: user:string -> idp:prin -> sp:prin -> unit
28
29 let rec loop userid idp sp =
30   let loginresp = ReceiveSaml idp in
31   match loginresp with
32   | UserAuthRequest(authmethod, challenge, sigAuth) ->
33     let authresponse = handleAuthMethod authmethod in
34     let authInfo = UserAuth userid authmethod authresponse in
35     let authresp = UserAuthResponse authInfo challenge sigAuth
36       in
37     SendSaml idp authresp;
38     loop userid idp sp
39   | AuthResponseMessage(idenp, dest, assertion) ->
40     SendSaml sp loginresp
41   | _ -> loginresp; ()
42
43 val browser: sp:prin -> res:uri -> unit
44
45 let browser sp resource =
46   let req = SPLogin resource in
47   let _ = SendSaml sp req in
48   let res = ReceiveSaml sp in
49   match res with
50   | AuthnRequestMessage(sp, idp, message, sigSP) ->
51     let _ = SendSaml idp res in
52     let idpResp = ReceiveSaml idp in
53     match idpResp with
54     | UserCredRequest(javascript, challenge, sigIdP) ->
55       let pubkissuer = CertStore.GetJSPublicKey idp in
56       if VerifySignature idp pubkissuer javascript sigIdP then
57         (assert (Log idp javascript));
58         let loginInfo = UserLogin userid password in
59         let loginreq = Login loginInfo challenge in
60         SendSaml idp loginreq;
61         loop userid idp sp;
62         let spResp = ReceiveSaml sp in
63         match spResp with
64         | LoginResponse(str) ->
65           fakeprint str
66         | _ -> spResp; ()
67       else
68         fakeprint "Validation Error"
69     | _ -> idpResp; ()
70   | _ -> res; ()

```

Listing 4.19: Browser's side of logging in

This part of the *Browser* module is used to model the client's side of a logging in session. It is worth noticing that the *Browser* verifies the JavaScript is actually received from the correct Identity Provider. The function *fakeprint* is used to give the user messages about errors and if they are logged in. The recursive function *loop* will provide n factor authentication methods until the client re-

ceives a *AuthResponseMessage* which it then will send to the Service Provider and the user is now logged in.

```

1  val newUserId: string
2  val newPassword: string
3  val idpToRevoke: string
4  val nfactToRemove: Authentication
5  val nfactToAdd: Authentication
6
7  val retrieveGeneratedPassword: string
8
9  val createUser: authp:prin -> unit
10
11 let createUser authp =
12   let name = userid in
13   let pw = password in
14   let req = RequestForLogin name pw email in
15   let _ = SendMessage authp req in
16   let resp = ReceiveMessage authp in
17   match resp with
18   | ReqLoginResponse(challenge) ->
19     let reqlresp = CreateLogin retrieveGeneratedPassword
20       challenge in
21     let _ = SendMessage authp reqlresp in
22     let createlloginresp = ReceiveMessage authp in
23     match createlloginresp with
24     | StatusMessage(status) ->
25       match status with
26       | Successful -> fakeprint "You have created an account"
27       | Unsuccessful -> fakeprint "Something went wrong. No
28         account has been created"
29       | _ -> createlloginresp; ()
30     | _ -> resp; ()
31
32 val changeUserPassword: authp:prin -> unit
33
34 let changeUserPassword authp =
35   let name = userid in
36   let pw = password in
37   let newpw = newPassword in
38   let req = ChangePassword name pw newpw in
39   let _ = SendMessage authp req in
40   let resp = ReceiveMessage authp in
41   match resp with
42   | StatusMessage(status) ->
43     match status with
44     | Successful -> fakeprint "You have change your password"
45     | Unsuccessful -> fakeprint "Something went wrong. You have
46       not changed your password"
47     | _ -> resp; ()
48
49 val changeUserUserId: authp:prin -> unit
50
51 let changeUserUserId authp =
52   let name = userid in
53   let pw = password in
54   let newid = newUserId in

```



```

52 let req = ChangeUserId name newid pw in
53 let _ = SendMessage authp req in
54 let resp = ReceiveMessage authp in
55 match resp with
56 | StatusMessage(status) ->
57   match status with
58   | Successful -> fakeprint "You have change your userid"
59   | Unsuccessful -> fakeprint "Something went wrong. You have
60     not changed your userid"
61 | _ -> resp; ()
62
63 val identityrevoke: authp:prin -> unit
64
65 let identityrevoke authp =
66   let name = userid in
67   let pw = password in
68   let idp = idpToRevoke in
69   let req = UserRevokeIdp name pw idp in
70   let _ = SendMessage authp req in
71   let resp = ReceiveMessage authp in
72   match resp with
73   | StatusMessage(status) ->
74     match status with
75     | Successful -> fakeprint "You have revoked the
76       identityprovider"
77     | Unsuccessful -> fakeprint "Something went wrong. You have
78       not revoked the identityprovider"
79 | _ -> resp; ()
80
81 val addNfact: authp:prin -> unit
82
83 let addNfact authp =
84   let name = userid in
85   let pw = password in
86   let nfact = nfactToAdd in
87   let req = AddNfactor name pw nfact in
88   let _ = SendMessage authp req in
89   let resp = ReceiveMessage authp in
90   match resp with
91   | StatusMessage(status) ->
92     match status with
93     | Successful -> fakeprint "You have added this
94       authentication method"
95     | Unsuccessful -> fakeprint "Something went wrong. You have
96       not added this authentication method"
97 | _ -> resp; ()
98
99 val removeNfact: authp:prin -> unit
100
101 let removeNfact authp =
102   let name = userid in
103   let pw = password in
104   let nfact = nfactToRemove in
105   let req = RemoveNfactor name pw nfact in
106   let _ = SendMessage authp req in
107   let resp = ReceiveMessage authp in
108   match resp with

```

```

104 | StatusMessage(status) ->
105 |   match status with
106 |   | Successful -> fakeprint "You have removed this
      authentication method"
107 |   | Unsuccessful -> fakeprint "Something went wrong. You have
      not removed this authentication method"
108 |   _ -> resp; ()

```

Listing 4.20: The Browser's side of the account creation and changing

This part of the *Browser* module is pretty straightforward. It specifies a lot of functions that will create the user's account and update the account by the user's wish. The nonce created when a user creates an account is used to relate the creation of an account to a user.

## 4.6 Introducing adversaries

In the previous section we have been focused on implementation of the protocol according to the specification. This section will introduce adversaries into the protocol verification however we have not managed in this project to incorporate dedicated adversaries. Jacob introduced adversaries in his protocol verification through an abstract program and a main function to execute a protocol run. We have adopted this way of introducing an adversary and applied it to the OpenNemID protocol as shown in listing 4.21. The difference between our way of introducing an adversary and Jacob's is ours have the Authentication Provider also. The abstract *attacker* function is a parameter to the *main* function. This means that it is able to use any function defined in the modules. However it will not be able to call any *assume* command, i.e. every assertion in the Service Provider, Identity Provider and Authentication Provider will succeed.

```

1  module Main
2
3  open SamlProtocol
4  open Crypto
5  open Serviceprovider
6  open Identityprovider
7  open Authenticationprovider
8
9  val Fork: list (unit -> unit) -> unit
10
11  let main attacker =
12    Fork [ attacker;
13      (fun () -> serviceprovider "serviceprovider.org" "browser" "
        identityprovider.org");
14      (fun () -> identityprovider "identityprovider.org" "browser"
        "authenticationprovider.org");
15      (fun () -> authenticationprovider "authenticationprovider.org"
        "identityprovider.org" "browser")]

```

Listing 4.21: Main module for introducing adversaries

It would be possible to model and mitigate known attacks on the protocol like *Man In the Middle*, *authentication replay* and *session hijacking* by modeling a browser as part of the protocol model. We have modeled the browser but we have not modeled the aforementioned attacks due to time constraints.

## 4.7 State of the implementation

We have implemented the Identity, Service and Authentication Provider in the OpenNemID protocol and defined their abstract implementation but they are all missing session handling. Furthermore the implementations of the cryptographic elements and networking are abstract signatures only at the moment. If time had allowed it we could have incorporated the crypto and networking experiments done by Jacob into the model. As the previous section explained we have not modeled dedicated adversaries as a part of the present implementation either.

# Chapter 5

## Evaluation

This chapter is an evaluation of the entire project. It will describe what has been accomplished during the project and what the work in this project can contribute with. It will also outline the most sensible areas to research further and discuss what related work have been done.

### 5.1 Project evaluation

In this report we have outlined the work of this project. The primary work in this project has been the specification of the OpenNemID protocol including a service provider, identity provider and authentication provider written in F\*.

Even though F\* is a research project from Microsoft Research which can be a difficult challenge itself it has still been a challenge to work with but it has been an enjoyable challenge. This is because of the fact that we have extended the work done by Jacob and the F\*-project is well documented with a tutorial. This means that we could always use Jacob's work as a reference and the compiler was easily to set up because of his work. We both have a little experience with ML and the functional oriented language F# which F\* is based on - these facts also helped us during the development of this project. Furthermore we both also have a strong background in C# and .NET.

Evaluating the project from a software development perspective

### 5.2 Related work

We have extended the work done by Jacob Højgaard [1] in his masters thesis *Securing Single Sign-On System With Executable Models*. His work has had a huge impact on this project as we have stated throughout the report. The impact has been both in understanding the language F\* and designing and specifying the OpenNemID protocol. In the masters thesis *Using static analysis to validate SAML Protocols* written by Hansen and Skriver [4] they analyze and formalize the SAML login protocol. Their analysis goes towards the older version

1.1 SAML protocol. In their analysis they recommend mandating of HTTPS network transport. This has been incorporated in the OIOSAML specification done by Jacob.

## 5.3 Threats to validity

When working in the area of security protocols there will be a lot of threats to validity. Therefore it is important to scrutinize the work of this report.

As stated earlier we both have a little experience with the functional programming language F# which F\* is based on. However since we are no experts in F# and have worked mostly with object oriented languages there is a good chance that the capabilities of F\* has not been fully utilized. Furthermore we have not worked with any formal specification tools before beginning this project.

It is also important to remember that the F\* compiler is still a  $\alpha$ -release. Microsoft Research has used it to verify 20.000 lines of code [5] but the results should still be treated with some degree of caution. Still the concepts in F\* are based on are derived from the work in the previous projects, F7 and Fine. In these projects the concepts have been investigated thoroughly.

Lastly it is worth noting we have only touched the subject of modeling adversaries very briefly. This leaves room for uncertainty and therefore would be a natural course to take for further research.

## 5.4 Further research

As we have mentioned the modeling of adversaries is the natural path to take because this is required to make a more robust framework for protocol verification. In section 4.6 we described that we have modeled a browser but we have not modeled adversaries as part of the browser. This could be a way to continue the research. We mentioned briefly that F\* could be compiled to JavaScript. The compilation of F\* to JavaScript is interesting because NemID uses a Java applet which forces all the user to install a Java browser plug-in. Given the facts that there have been many security issues reported lately with the Java browser plug-in and that DanID are working on a version of NemID in JavaScript F\*'s compilation to JavaScript is very to have in mind. This means that a specification in F\* could be compiled to JavaScript and used on the web and thereby eliminate the large security risk involved with using the Java applet. Further research could also be into the server side of the Authentication Provider (as we have called it). As we have specified the Authentication Provider the next step would be to actually creating the server side of it so it could be an executable protocol on the web.

## 5.5 Conclusion

Vi skal vel konkludere p introduktionen? Konklusionen skal vel skrives efter den? :)

# Bibliography

- [1] Jakob Højgaard: *Securing Single Sign-On System With Executable Models*. Masters thesis, IT University of Copenhagen, 2013.
- [2] David Basin, Patrick Schaller, Michael Schläpfer: *Applied Information Security - A Hands-on Approach*. Springer, Berlin Heidelberg, 2011.
- [3] W3.org: *Web Cryptography API*. <http://www.w3.org/TR/WebCryptoAPI/>
- [4] Steffen M. Hansen and Jakob Skriver: *Using Static Analysis to Validate SAML Protocols*. Masters thesis, Technical University of Denmark, 2004.
- [5] Microsoft Research: *F\*: A Verifying ML Compiler for Distributed Programming*. <http://research.microsoft.com/en-us/projects/fstar/>
- [6] Microsoft Research: *F\* guide* <http://rise4fun.com/FStar/tutorial/guide>
- [7] The WAYF secretariat: *Trusted third party based ID federation, lowering the bar for connecting and enhancing privacy*. Explanation of WAYF, 2009.
- [8] ProVerif: *Cryptographic protocol verifier in the formal model*. <http://prosecco.gforge.inria.fr/personal/bblanche/proverif/>
- [9] JSCert: *Certified JavaScript*. <http://jscert.org/index.html>
- [10] XOR-ProVerif: *Tool for analyzing protocols with XOR*. <http://www.infsec.uni-trier.de/publications/paper/KuestersTruderung-CCS-2008.pdf>
- [11] DH-ProVerif: *Tool for analyzing protocol with Diffie-Hellman exponentiation*. <http://www.infsec.uni-trier.de/publications/paper/KuestersTruderung-CSF-2009.pdf>
- [12] CSPProto: *Tool for analyzing contract signing protocols* <http://www.infsec.uni-trier.de/publications/paper/KaehlerKuestersWilke-TOCL-2010.pdf>

- [13] Sergio Maffeis: *An expert in computer security*. Various publications on web security, security protocols and so on. <http://www.doc.ic.ac.uk/~maffeis/>
- [14] NaCl: *Networking and Cryptography library*. [nacl.cr.yp.to](http://nacl.cr.yp.to)
- [15] Sodium: *A new cross-compilable Cryptography library*. A variation of the NaCl library, <http://labs.umbrella.com/2013/03/06/announcing-sodium-a-new-cryptographic-library/>
- [16] Nets-Danid: *NemID guide* [https://www.nets-danid.dk/produkter/for\\_tjenesteudbydere/nemid\\_tjenesteudbyder/nemid\\_tjenesteudbyder\\_support/tjenesteudbyderpakken/](https://www.nets-danid.dk/produkter/for_tjenesteudbydere/nemid_tjenesteudbyder/nemid_tjenesteudbyder_support/tjenesteudbyderpakken/)
- [17] Danny Dolev and Andrew C. Yao: *On the security of public key protocols*. Technical report, Stanford, CA, USA 1981.



## Appendix A

# SAML Assertion and Messages