IT UNIVERSITY OF COPENHAGEN

BACHELOR PROJECT

Verifiable Secure Open Source Alternative to NemID

Authors:

Andreas Hallberg KJELDSEN
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...

Acknowledgements

Thanks Joe! Thanks Hannes! Party on!

Contents

List of Figures

Listings

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 get 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

2.1 SAML protocol

2.1.1 OIOSAML

2.2 Static analysis

2.3 Selection of verification tool

F* - formal specification language that is also executable

2.4 N-factor authentication

Viderudvikling af two factor authentication

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 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.

3.1.1 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.

3.1.2 NemID specifics

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

3.1.3 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].

3.2 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.).

3.3 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 Digitaliseringsstyrrelsen.

3.3.1 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.

3.3.2 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].

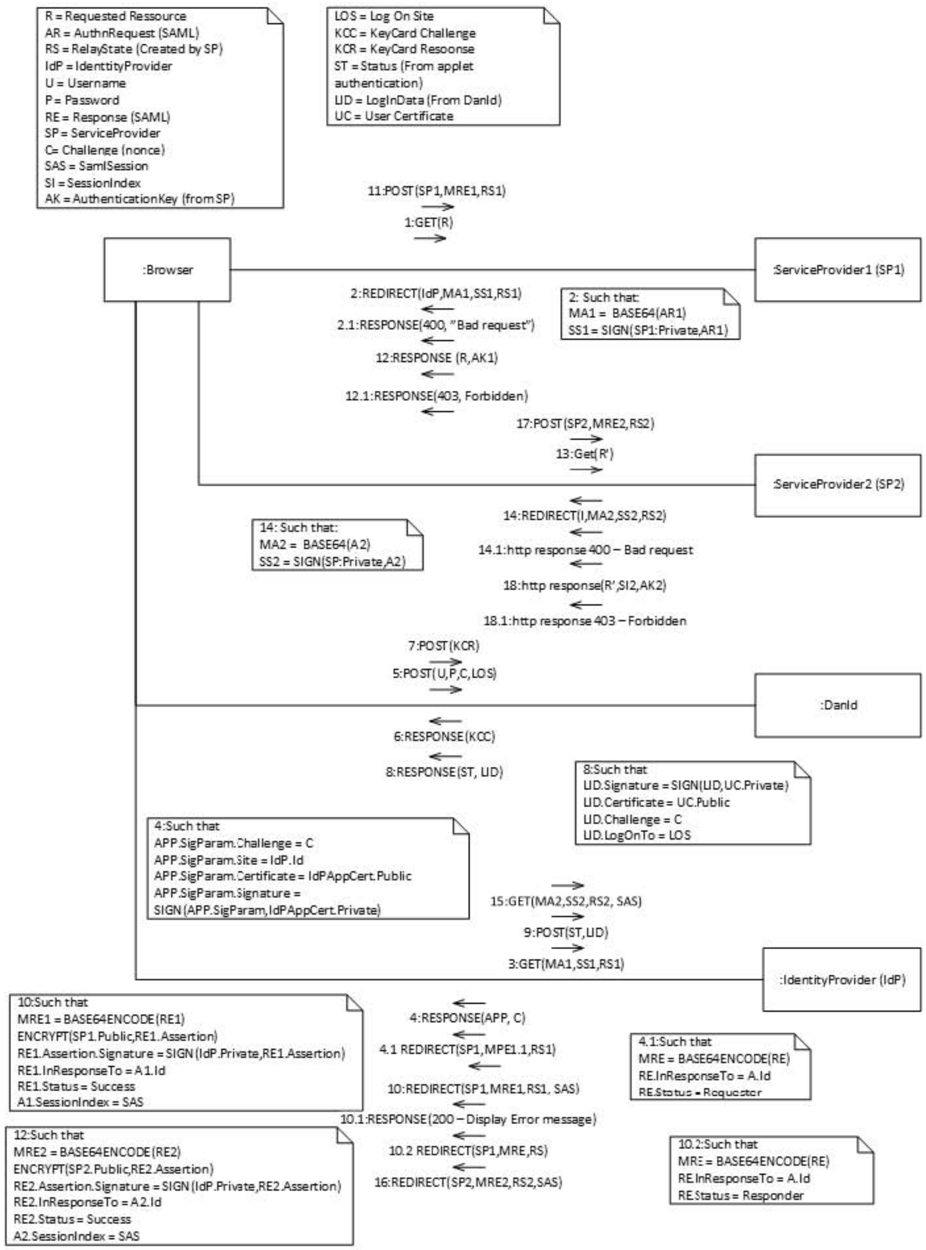

Figure 3.1: Communication diagram for the complete NemID protocol [1]

3.3.3 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 3.1, 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.

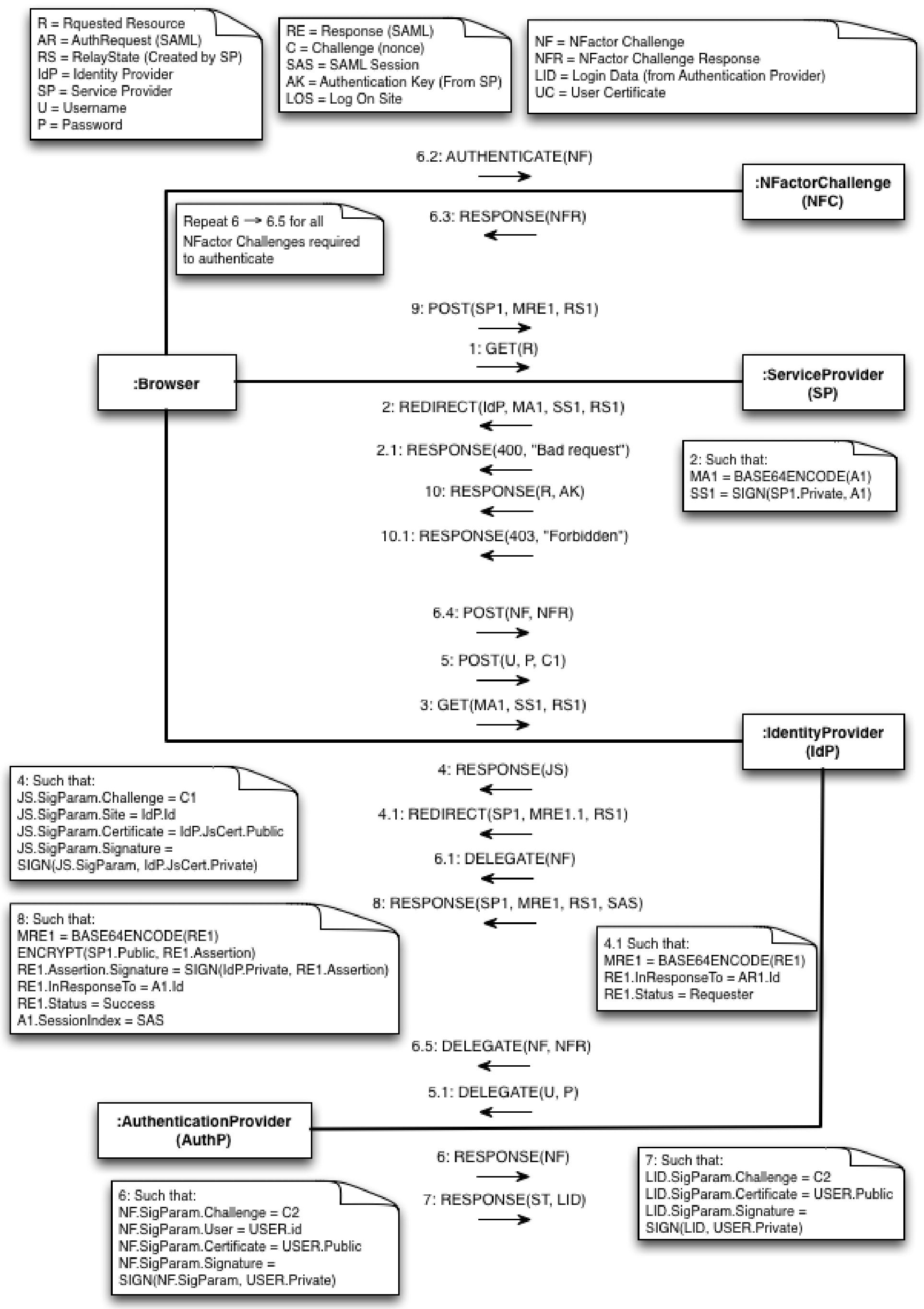

Figure 3.2: Communication diagram for the OpenNemID protocol

Message descriptions

Process 1 describes the creation of the AuthRequest at the service provider when a resource has been requested. Is 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 contains 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 ← DeflateDecompress(Base64Dec(UrlDec(MA)))
if VERIFY(AR, SAR, SP.Public) then
    C1 ← GenChallenge()
    JS ← GetStoredJavaScript()
    JS.SigParams.Challenge ← C1
    JS.SigParams.Certificate ← IdPJsCert.Public
    JS.SigParams.Signature ← SIGN(JS.SigParams, IdPJsCert.Private)
    ARC ← CreateCookie(MA, SAR, RS)
    return RESPONSE(JS, ARC)
else
    RE ← CreateResponse()
    RE.InResponseTo ← AR
    RE.Status ← VerificationFailied
    MRE ← 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 ← Js.SigParams
if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
    C1 ← 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 is 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 ← GetUser(U, P)
  if USER is valid **then**
    C2 ← GenChallenge()
    NF ← GetNextNFactorChallenge(USER)
    NF.SigParam.User ← USER
    NF.SigParam.Challenge ← C2
    NF.SigParam.Certificate ← USER.Public
    NF.SigParam.Signature ← 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 ← 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 ← NF.SigParams
  if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
    if NFR is acceptable result of NF then
      USER ← GetUser(SigParams.USER, SigParams.Certificate)
      C2 ← GenChallenge()
      if USER.HasNextChallenge then
        NF ← GetNextNFactorChallenge(USER)
        NF.SigParams.User ← USER
        NF.SigParams.Challenge ← C2
        NF.SigParams.Certificate ← USER.Public
        NF.SigParams.Signature ← SIGN(NF.SigParams, USER.Private)
        return RESPONSE(NF)
      else
        LID ← CreateLogInData(USER)
        ST ← "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. Failed verification 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.

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

TEXT DESCRIBING ALGORITHM 11

Algorithm 11 Process 9

```
Require: POST is well.formed and SP.Private and IdP.Public
  RE ← UrlDec(Base64Dec(DeflateDecompress(MRE)))
  A ← DECRYPT(RE.Assertion, SP.Private)
  if VERIFY(A, A.Signature, IdP.Public) then
    AK ← GenAuthKey()
    R ← Base64Dec(UrlDec(RS))
    RES ← GetResource(R)
    return RESPONSE(RES, AK)
  else
    return RESPONSE(403, Forbidden)
  end if
```

Chapter 4

Modelling with F*

This chapter will introduce the language F* that 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[1] and Fine[2]. F* is at this time considered to be an α-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 loosing 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[3] SMT solver for proving assumptions made with refinement types. F* has been formalized and verified using Coq[4].

[1] http://research.microsoft.com/en-us/projects/f7/
[2] http://research.microsoft.com/en-us/projects/fine/
[3] http://z3.codeplex.com/
[4] 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.

```
module HelloWorld

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 multiplies 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.

```
module Multiplication

val multiply: x:int -> y: int -> int

let multiply x y = x * y

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 of 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.

```
(*Declare a type nat of natural numbers*)
type nat = i:int{0 <= i}

let x:nat = 1
let y:nat = 0
```

```
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

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*.

The arguments

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

Unit means the same as void in Java.

```
module Identityprovider

open SamlProtocol
open Crypto

let handleUserAuthenticated me user client authnrequest =
  match authnrequest with
  | MkAuthnRequest(reqid,issueinst,dest,sp,msg,sigSP) ->
      let pubksp = CertStore.GetPublicKey sp in

      if (VerifySignature sp pubksp msg sigSP) then
        (assert (Log sp msg);
        let assertion = IssueAssertion me user sp reqid in
        let myprivk = CertStore.GetPrivateKey me in
        assume(Log me assertion);
        let sigAs = Sign me myprivk assertion in
        let signAssertion = AddSignatureToAssertion assertion
            sigAs in
        let encryptedAssertion = EncryptAssertion sp pubksp
```

```
signAssertion in
         let resp = AuthResponseMessage me sp encryptedAssertion in
         SendSaml client resp) (*10*)
       else
         SendSaml client (Failed Requester)(*10.2*)

 val identityprovider: me:prin -> client:prin -> unit

 let rec identityprovider me client =
  let req = ReceiveSaml client in (*3 & 11*)
  match req with
  | AuthnRequestMessage (issuer, destination, message, sigSP) ->
     let pubkissuer = CertStore.GetPublicKey issuer in
     if (VerifySignature issuer pubkissuer message sigSP) then
       (assert (Log issuer message);
       let challenge = GenerateNonce me in
       let resp = UserCredRequest challenge in
       SendSaml client resp; (*4*)
       identityprovider me client (*Start over*))
     else
       SendSaml client (Failed Requester);(*4.1*)
       identityprovider me client (*Start over*)

   | UserAuthenticated (status, logindata, authnrequest) ->
     match logindata with
     | MkLoginData (user,sig,cert,challenge,site,data) ->
       if (status = "OK") && (VerifySignature user cert data sig) then
         (assert (Log user data);
           handleUserAuthenticated me user client authnrequest;
           identityprovider me client (*Start over*)
         )
       else
         SendSaml client (DisplayError 400);(*10.1*)
         identityprovider me client (*Start over*)
   | _ -> SendSaml client (DisplayError 400);(*10.1*)
         identityprovider me client (*Start over*)
```

Listing 4.4: Jacob's Identity Provider Implementation

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 AuthnRequest 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. ==> 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.

```
type pubkey :: prin => *
type privkey :: prin => *

type Log :: prin => string => E

val Sign: p:prin
 -> privkey p
 -> msg:string{Log p msg}
 -> dsig

val VerifySignature: p:prin
 -> pubkey p
 -> msg:string
 -> dsig
 -> 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[5].
The F* code for the protocol is organized in 10 modules:

[5] https://github.com/kiniry-supervision/OpenNemID

1. The TypeFunc module
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[6]. 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

```
module TypeFunc

type Authentication =
   | Facebook: id:int -> Authentication
   | SMS: generated:int -> Authentication
  | Google: id:int -> Authentication
  | 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

```
module SamlProtocol

open Crypto
open TypeFunc

```

[6]Cryptographic primitives are assumed perfect and cyphers cannot be decrypted without the the proper decryption key

```
type assertiontoken = string (*Add refinements*)
type signedtoken = string (*Add refinements*)
type id = string
type endpoint = string
type uri = string


type AuthnRequest =
  | MkAuthnRequest: IssueInstant:string ->
      Destination:endpoint -> Issuer:prin ->
      message:string -> sig:dsig ->
      AuthnRequest

type LoginData =
  | MkLoginData:  user:prin -> signature:dsig ->
    cert:pubkey user -> challenge:nonce ->
    site:string -> data:string ->
    LoginData

type LoginInfo =
  | UserLogin:  userid:string -> password:string ->
  LoginInfo

type AuthInfo =
  | UserAuth:   userid:string -> authmethod:Authentication ->
  authresponse:Authentication -> AuthInfo

type Assertion =
  | SignedAssertion: assertiontoken -> dsig -> Assertion
  | EncryptedAssertion: cypher -> Assertion

type SamlStatus =
  | Success: SamlStatus
  | Requester: SamlStatus
  | Responder: SamlStatus
  | User: SamlStatus

type SamlMessage =
  | SPLogin: uri -> SamlMessage
  | Login: loginInfo:LoginInfo -> challenge:nonce ->
      SamlMessage
  | LoginResponse: string -> SamlMessage
  | AuthnRequestMessage: issuer:prin ->  destination:endpoint
     -> message:string -> dsig -> SamlMessage
  | LoginRequestMessage: issuer:prin ->  destination:endpoint
     -> loginInfo:LoginInfo -> SamlMessage
  | NfactAuthRequest: issuer:prin -> destination:endpoint ->
      authInfo:AuthInfo -> challenge:nonce -> dsig ->
      SamlMessage
  | AuthResponseMessage: issuer:prin -> destination:endpoint ->
       Assertion -> SamlMessage
  | LoginResponseMessage: issuer:prin -> destination:endpoint
      -> auth:Authentication -> challenge:nonce -> dsig ->
      SamlMessage
  | UserAuthenticated: status:string -> logindata:LoginData ->
      authnReq:AuthnRequest -> SamlMessage
```

```
  | UserCredRequest: javascript:string -> challenge:nonce -> dsig -> SamlMessage
  | UserAuthRequest: authmethod:Authentication -> challenge:nonce -> dsig -> SamlMessage
  | UserAuthResponse: authInfo:AuthInfo -> challenge:nonce -> dsig -> SamlMessage
  | LoginSuccess: status:string -> issuer:prin -> destination:endpoint -> SamlMessage
  | Failed: SamlStatus -> SamlMessage
  | DisplayError: int -> SamlMessage


val SendSaml: prin -> SamlMessage -> unit
val ReceiveSaml: prin -> SamlMessage

val CreateAuthnRequestMessage: issuer:prin -> destination:prin -> string
val IssueAssertion: issuer:prin -> subject:prin -> audience:prin -> inresto:AuthnRequest -> assertiontoken
val AddSignatureToAssertion: assertiontoken -> dsig -> signedtoken
val EncryptAssertion: receiver:prin -> pubkey receiver -> signedtoken -> Assertion
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 provice 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

```
module Crypto

open Protocol
open TypeFunc

type prin = string
type pubkey :: prin => *
type privkey :: prin => *
type dsig
type nonce = string
type cypher

(*Verification*)
type Log :: prin => string => E
```

```

type LogAuth :: prin => Authentication => E

val Keygen: p:prin
   -> (pubkey p * privkey p)

val Sign: p:prin
 -> privkey p
 -> msg:string{Log p msg}
 -> dsig

val SignAuth: p:prin
 -> privkey p
 -> msg:Authentication{LogAuth p msg}
 -> dsig

val VerifySignature: p:prin
 -> pubkey p
 -> msg:string
 -> dsig
 -> b:bool{b=true ==> Log p msg}

val VerifySignatureAuth: p:prin
 -> pubkey p
 -> msg:Authentication
 -> dsig
 -> b:bool{b=true ==> LogAuth p msg}

val Encrypt: p:prin
 -> pubkey p
 -> string
 -> cypher

val Decrypt: p:prin
 -> privkey p
 -> cypher
 -> string

val GenerateNonce: prin -> nonce (*Add refinement to ensure
    unqueness*)
```

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

```
module CertStore

```

```
open Crypto

val GetPublicKey: p:prin -> pubkey p
val GetJSPublicKey: p:prin -> pubkey p
(*Prin needs to be updated to include credentials*)
val GetPrivateKey: p:prin -> privkey p
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

```
module Messaging

open Crypto
open TypeFunc

type Status =
 | Successful: Status
 | Unsuccessful: Status

type Message =
 | NewSiteRequest: idp:prin -> Message
 | ChallengeResponse: challenge:nonce -> Message
 | IdpChalResponse: challenge:nonce -> Message
 | AcceptedIdp: idp:prin -> pubkey:pubkey idp -> authp:prin ->
   authpubkey:pubkey authp -> signedjavascript:string ->
   Message
 | RequestForLogin: userid:string -> password:string -> email:
   string -> Message
 | ReqLoginResponse: challenge:nonce -> Message
 | CreateLogin: generatedpassword:string -> challenge:nonce ->
   Message
 | ChangeUserId: userid:string -> newUserId:string -> password:
   string -> Message
 | ChangePassword: userid:string -> password:string ->
   newPassword:string -> Message
 | UserRevokeIdp: userid:string -> password:string -> idp:
   string -> Message
 | AddNfactor: userid:string -> password:string -> nfact:
   Authentication -> Message
 | RemoveNfactor: userid:string -> password:string -> nfact:
   Authentication -> Message
 | StatusMessage: Status -> Message
```

```

val SendMessage: prin -> Message -> unit
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

```
module Serviceprovider

open SamlProtocol
open Crypto

val serviceprovider:  me:prin -> client:prin -> idp:prin ->
    unit

let rec serviceprovider me client idp =
 let req = ReceiveSaml client in
 match req with
   | SPLogin (url) ->
     let authnReq = CreateAuthnRequestMessage me idp in
     assume(Log me authnReq);
     let myprivk = CertStore.GetPrivateKey me in
     let sigSP = Sign me myprivk authnReq in
     let resp = AuthnRequestMessage me idp authnReq sigSP in
     SendSaml client resp;
     serviceprovider me client idp
   | AuthResponseMessage (issuer, destination, encassertion) ->
     let myprivk = CertStore.GetPrivateKey me in
     let assertion = DecryptAssertion me myprivk encassertion in
     match assertion with
     | SignedAssertion (token,sigIDP) ->
       let pubkissuer = CertStore.GetPublicKey idp in
       if VerifySignature idp pubkissuer token sigIDP
       then
         (assert(Log idp token);
         let resp = LoginResponse "You are now logged in" in
         SendSaml client resp)
       else SendSaml client (DisplayError 403);
       serviceprovider me client idp

   | _ -> SendSaml client (DisplayError 400);
         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 3.3. 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.

```
module Identityprovider

open SamlProtocol
open Crypto
open TypeFunc
open Messaging

val userloggedin: user:prin -> bool
val getjavascript: string
val userlogin: user:prin -> unit
val decodeMessage: message:string -> AuthnRequest
val getauthnrequest: user:prin -> challenge:nonce ->
    AuthnRequest
val getuserchallenge: user:prin -> nonce
val relatechallenge: user:prin -> challenge:nonce -> unit
val verifychallenge: user:prin -> challenge:nonce -> bool
val relate: user:prin -> challenge:nonce -> authnReq:
    AuthnRequest -> unit

val identityprovider: me:prin -> user:prin -> authp:prin ->
    unit

let rec identityprovider me user authp =
 let request = ReceiveSaml user in
 match request with
 | AuthnRequestMessage(issuer, destination, message, sigSP) ->
  let pubkissuer = CertStore.GetPublicKey issuer in
 if (VerifySignature issuer pubkissuer message sigSP) then
  (assert (Log issuer message);
  let authnReq = decodeMessage message in
  let myprivk = CertStore.GetPrivateKey me in
  if not (userloggedin user) then
   let challenge = GenerateNonce me in
   relate user challenge authnReq;
   relatechallenge user challenge;
   let js = getjavascript in
   assume(Log me js);
   let myprivk = CertStore.GetJSPrivateKey me in
   let sigIdP = Sign me myprivk js in
   let resp = UserCredRequest js challenge sigIdP in
   SendSaml user resp;
   identityprovider me user authp
```

```
   else
    let assertion = IssueAssertion me user issuer authnReq in
    assume(Log me assertion);
    let myprivk = CertStore.GetPrivateKey me in
    let pubksp = CertStore.GetPublicKey issuer in
    let sigAs = Sign me myprivk assertion in
    let signAssertion = AddSignatureToAssertion assertion sigAs
        in
    let encryptedAssertion = EncryptAssertion issuer pubksp
        signAssertion in
    let resp = AuthResponseMessage me issuer encryptedAssertion
        in
    SendSaml user resp)
  else
   SendSaml user (Failed Requester);
   identityprovider me user authp
  | Login (loginInfo, challenge) ->
   if (verifychallenge user challenge) then
    let req = LoginRequestMessage me authp loginInfo in
    SendSaml authp req;
    handleauthresponse me user authp;
    identityprovider me user authp
   else
    SendSaml user (DisplayError 400);
    identityprovider me user authp
  | UserAuthResponse(authInfo, challenge, sigAuth) ->
   let req = NfactAuthRequest me authp authInfo challenge
       sigAuth in
   SendSaml authp req;
   handleauthresponse me user authp;
   identityprovider me user authp
  | _ -> SendSaml user (DisplayError 400);
   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.

```
val handleUserAuthenticated: me:prin -> user:prin -> authnReq: AuthnRequest -> unit

let handleUserAuthenticated me user authnReq =
 match authnReq with
 | MkAuthnRequest(issueinst,dest,sp,msg,sigSP) ->
  let pubksp = CertStore.GetPublicKey sp in
   if (VerifySignature sp pubksp msg sigSP) then
  (assert (Log sp msg);
  let assertion = IssueAssertion me user sp authnReq in
  let myprivk = CertStore.GetPrivateKey me in
  assume(Log me assertion);
  userlogin user;
  let sigAs = Sign me myprivk assertion in
  let signAssertion = AddSignatureToAssertion assertion sigAs in
  let encryptedAssertion = EncryptAssertion sp pubksp signAssertion in
  let resp = AuthResponseMessage me sp encryptedAssertion in
  SendSaml user resp)
      else
 SendSaml user (Failed Requester)
val handleauthresponse: me:prin -> user:prin -> authp:prin -> unit

let handleauthresponse me user authp =
 let resp = ReceiveSaml authp in
 match resp with
 | LoginResponseMessage(issuer, destination, authmethod, challenge, sigUser) ->
  let pubkeyuser = CertStore.GetPublicKey user in
  if VerifySignatureAuth user pubkeyuser authmethod sigUser then
   (assert (LogAuth user authmethod);
   relatechallenge user challenge;
   let resp = UserAuthRequest authmethod challenge sigUser in
   SendSaml user resp)
  else
   SendSaml user (DisplayError 403)
 | LoginSuccess(status, issuer, destination) ->
  if (status = "OK") then
   let challenge = getuserchallenge user in
   let authnReq = getauthnrequest user challenge in
   handleUserAuthenticated me user authnReq
  else
   SendSaml user (DisplayError 403)
 | _ -> 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.

```
val savejavascript: javascript:string -> unit
val savepublickey: owner:prin -> publickey:pubkey owner -> unit

val connectwithauthp: me:prin -> authp:prin -> unit

let connectwithauthp me authp =
 let req = NewSiteRequest me in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
 match resp with
 | ChallengeResponse(challenge) ->
  let _ = SendMessage authp (IdpChalResponse challenge) in
  let res = ReceiveMessage authp in
  match res with
  | AcceptedIdp(idp, idppubkey, authp, authppubkey, signedjs)
     ->
   (*Established secure connection*)
   savejavascript signedjs;
   savepublickey authp authppubkey;
   savepublickey idp idppubkey
  | _ -> res; ()
 | _ -> resp; ()
```

Listing 4.14: Establising 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

```
module Database

open Crypto
open CertStore
open TypeFunc

(*Identity provider functionality*)
val whitelist: idp:prin -> unit
val blacklist: idp:prin -> unit
val addidp: idp:prin -> bool
val whitelisted: idp:prin -> bool

```

```
(*User functionality*)
val createuser: user:prin -> userid:string -> password:string
    -> bool
val usercreation: user:prin -> generatedPassword:string -> bool
val changeuserid: user:string -> newuser:string -> password:
    string -> bool
val changeuserpassword: user:string -> password:string ->
    newpassword:string -> bool

val addnfactor: user:string -> password:string -> nfactor:
    Authentication -> bool
val removenfactor: user:string -> password:string -> nfactor:
    Authentication -> bool

val getnfactor: user:string -> Authentication
val checknfactor: user:string -> Authentication -> bool
val allnfactauthed: user:string -> bool
val resetnfact: user:string -> unit

val checklogin: user:string -> password:string -> bool

val revokeidp: user:string -> password:string -> idp:string ->
    bool

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.

```
module Authenticationprovider

open SamlProtocol
open Crypto
open Database
open TypeFunc
open Messaging

val relatechallenge: user:prin -> challenge:nonce -> unit

val verifychallenge: user:prin -> challenge:nonce -> bool

val nfactauth: me:prin -> idp:prin -> user:prin -> userid:
    string -> unit

let nfactauth me idp user userid =
 if (allnfactauthed userid) then
```

```
  resetnfact userid;
  let status = "OK" in
  let resp = LoginSuccess status me idp in
  SendSaml idp resp
 else
  let challenge = GenerateNonce me in
  let authmethod = getnfactor userid in
  assume(LogAuth user authmethod);
  let userprivkey = CertStore.GetPrivateKey user in
  let sigUser = SignAuth user userprivkey authmethod in
  let resp = LoginResponseMessage me idp authmethod challenge sigUser in
  SendSaml idp resp

val authenticationprovider: me:prin -> idp:prin -> user:prin -> unit

let rec authenticationprovider me idp user =
 let req = ReceiveSaml idp in
 match req with
 | LoginRequestMessage (issuer, destination, loginInfo) ->
  if (whitelisted idp) then
   match loginInfo with
   | UserLogin(userid, password) ->
    if not (revokedidp userid idp) && (checklogin userid password) then
     let challenge = GenerateNonce me in
     let authmethod = getnfactor userid in
     assume(LogAuth user authmethod);
     let userprivkey = CertStore.GetPrivateKey user in
     let sigUser = SignAuth user userprivkey authmethod in
     relatechallenge user challenge;
     let resp = LoginResponseMessage me idp authmethod challenge sigUser in
     SendSaml idp resp;
     authenticationprovider me idp user
    else
     SendSaml idp (Failed User);
     authenticationprovider me idp user
   | _ -> SendSaml idp (Failed Requester);
    authenticationprovider me idp user
  else
   SendSaml idp (Failed Requester);
   authenticationprovider me idp user
 | NfactAuthRequest(issuer, destination, authInfo, challenge, sigAuth) ->
  if (whitelisted idp) then
   match authInfo with
   | UserAuth(userid, authmethod, authresponse) ->
    let userpubkey = CertStore.GetPublicKey user in
    if VerifySignatureAuth user userpubkey authmethod sigAuth && verifychallenge user challenge then
     if not (revokedidp userid idp) && (checknfactor userid authresponse) then
      nfactauth me idp user userid;
      authenticationprovider me idp user
     else
```

```
      SendSaml idp (Failed User);
      authenticationprovider me idp user
    else
     SendSaml idp (Failed User);
     authenticationprovider me idp user
   | _ -> SendSaml idp (Failed Requester);
    authenticationprovider me idp user
  else
   SendSaml idp (Failed Requester);
   authenticationprovider me idp user
 | _ -> SendSaml idp (Failed Requester);
  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.

```
val usercommunication: me:prin -> user:prin -> unit

let rec usercommunication me user =
 let req = ReceiveMessage user in
 match req with
 | RequestForLogin(userid, password, email) ->
  if createuser user userid password email then
   let challenge = GenerateNonce me in
   relatechallenge user challenge;
   SendMessage user (ReqLoginResponse challenge);
   usercommunication me user
  else
   SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
 | CreateLogin(generatedpassword, challenge) ->
  if (verifychallenge user challenge) && (usercreation user
      generatedpassword) then
   let challenge = GenerateNonce me in
   relatechallenge user challenge;
   SendMessage user (StatusMessage Successful);
   usercommunication me user
  else
```

```
   SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
 | ChangePassword(userid, password, newPassword) ->
  if changeuserpassword userid password newPassword then
   SendMessage user (StatusMessage Successful);
   usercommunication me user
  else
   SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
 | ChangeUserId(userid, newUserId, password) ->
  if changeuserid userid newUserId password then
   SendMessage user (StatusMessage Successful);
   usercommunication me user
  else
   SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
 | UserRevokeIdp(userid, password, idp) ->
  if revokeidp userid password idp then
   SendMessage user (StatusMessage Successful);
   usercommunication me user
  else
   SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
 | AddNfactor(userid, password, nfact) ->
  if addnfactor userid password nfact then
   SendMessage user (StatusMessage Successful);
   usercommunication me user
  else
   SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
 | RemoveNfactor(userid, password, nfact) ->
  if removenfactor userid password nfact then
   SendMessage user (StatusMessage Successful);
   usercommunication me user
  else
   SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
 | _ -> SendMessage user (StatusMessage Unsuccessful);
  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.

```
val getsignedjavascript: string

val establishidp: me:prin -> idp:prin -> unit

let rec establishidp me idp =
 let req = ReceiveMessage idp in
 match req with
 | NewSiteRequest(idp) ->
  let challenge = GenerateNonce me in
```

```
   relatechallenge idp challenge;
   SendMessage idp (ChallengeResponse challenge);
   establishidp me idp
  | IdpChalResponse(challenge) ->
   if (verifychallenge idp challenge) && (addidp idp) then
    let idppubkey = CertStore.GetPublicKey idp in
    let mypubk = CertStore.GetPublicKey me in
    let signedjs = getsignedjavascript in
    let resp = AcceptedIdp idp idppubkey me mypubk signedjs in
    SendMessage idp resp;
    establishidp me idp
   else
    SendMessage idp (StatusMessage Unsuccessful);
    establishidp me idp
  | _ -> SendMessage idp (StatusMessage Unsuccessful);
   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.

```
module Browser

open SamlProtocol
open Crypto
open CertStore
open TypeFunc
open Messaging

val loginWithFb: Authentication
val loginWithGoogle: Authentication
val loginWithSMS: Authentication
val loginWithOpenId: Authentication
val userid: string
val password: string
val email: string
val fakeprint: str:string -> unit

val handleAuthMethod: auth:Authentication -> Authentication

let handleAuthMethod auth =
 match auth with
 | Facebook(fbid) -> loginWithFb
 | Google(gid) -> loginWithGoogle
```

```
  | SMS(gen) -> loginWithSMS
  | OpenId(oid) -> loginWithOpenId

val loop: user:string -> idp:prin -> sp:prin -> unit

let rec loop userid idp sp =
 let loginresp = ReceiveSaml idp in
  match loginresp with
  | UserAuthRequest(authmethod, challenge, sigAuth) ->
   let authresponse = handleAuthMethod authmethod in
   let authInfo = UserAuth userid authmethod authresponse in
   let authresp = UserAuthResponse authInfo challenge sigAuth
       in
   SendSaml idp authresp;
   loop userid idp sp
  | AuthResponseMessage(idenp, dest, assertion) ->
   SendSaml sp loginresp
  | _ -> loginresp; ()

val browser: sp:prin -> res:uri -> unit

let browser sp resource =
 let req = SPLogin resource in
 let _ = SendSaml sp req in
  let res = ReceiveSaml sp in
  match res with
  | AuthnRequestMessage(sp, idp, message, sigSP) ->
   let _ = SendSaml idp res in
   let idpResp = ReceiveSaml idp in
   match idpResp with
   | UserCredRequest(javascript, challenge, sigIdP) ->
    let pubkissuer = CertStore.GetJSPublicKey idp in
    if VerifySignature idp pubkissuer javascript sigIdP then
     (assert (Log idp javascript);
     let loginInfo = UserLogin userid password in
     let loginreq = Login loginInfo challenge in
     SendSaml idp loginreq;
     loop userid idp sp;
     let spResp = ReceiveSaml sp in
     match spResp with
     | LoginResponse(str) ->
       fakeprint str
     | _ -> spResp; ())
    else
     fakeprint "Validation Error"
   | _ -> idpResp; ()
  | _ -> 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 receives a *AuthResponseMessage* which it then will send to the Service Provider and the user is now logged in.

```
val newUserId: string
val newPassword: string
val idpToRevoke:string
val nfactToRemove: Authentication
val nfactToAdd: Authentication

val retrieveGeneratedPassword: string

val createUser: authp:prin -> unit

let createUser authp =
 let name = userid in
 let pw = password in
 let req = RequestForLogin name pw email in
 let _ = SendMessage authp req in
  let resp = ReceiveMessage authp in
  match resp with
  | ReqLoginResponse(challenge) ->
   let reqlresp = CreateLogin retrieveGeneratedPassword
       challenge in
   let _ = SendMessage authp reqlresp in
   let createloginresp = ReceiveMessage authp in
   match createloginresp with
   | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have created an account"
    | Unsuccessful -> fakeprint "Something went wrong. No
        account has been created"
   | _ -> createloginresp; ()
  | _ -> resp; ()

val changeUserPassword: authp:prin -> unit

let changeUserPassword authp =
 let name = userid in
 let pw = password in
 let newpw = newPassword in
 let req = ChangePassword name pw newpw in
 let _ = SendMessage authp req in
  let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have change your password"
    | Unsuccessful -> fakeprint "Something went wrong. You have
         not changed your password"
  | _ -> resp; ()

val changeUserUserId: authp:prin -> unit

let changeUserUserId authp =
 let name = userid in
 let pw = password in
 let newid = newUserId in
 let req = ChangeUserId name newid pw in
 let _ = SendMessage authp req in
```

```
  let resp = ReceiveMessage authp in
   match resp with
   | StatusMessage(status) ->
     match status with
     | Successful -> fakeprint "You have change your userid"
     | Unsuccessful -> fakeprint "Something went wrong. You have not changed your userid"
   | _ -> resp; ()

val identityrevoke: authp:prin -> unit

let identityrevoke authp =
 let name = userid in
 let pw = password in
 let idp = idpToRevoke in
 let req = UserRevokeIdp name pw idp in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have revoked the identityprovider"
    | Unsuccessful -> fakeprint "Something went wrong. You have not revoked the identityprovider"
  | _ -> resp; ()

val addNfact: authp:prin -> unit

let addNfact authp =
 let name = userid in
 let pw = password in
 let nfact = nfactToAdd in
 let req = AddNfactor name pw nfact in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have added this authentication method"
    | Unsuccessful -> fakeprint "Something went wrong. You have not added this authentication method"
  | _ -> resp; ()

val removeNfact: authp:prin -> unit

let removeNfact authp =
 let name = userid in
 let pw = password in
 let nfact = nfactToRemove in
 let req = RemoveNfactor name pw nfact in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
```

```
      | Successful -> fakeprint "You have removed this authentication method"
      | Unsuccessful -> fakeprint "Something went wrong. You have not removed this authentication method"
    | _ -> 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.

```
module Main

open SamlProtocol
open Crypto
open Serviceprovider
open Identityprovider
open Authenticationprovider

val Fork: list (unit -> unit) -> unit

let main attacker =
 Fork [ attacker;
   (fun () -> serviceprovider "serviceprovider.org" "browser" "identityprovider.org");
   (fun () -> identityprovider "identityprovider.org" "browser" "authenticationprovider.org");
   (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 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] from his master project *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.

5.3 Threats to validity

5.4 Further research

5.5 Conclusion

Bibliography

[1] Jakob Højgaard: *Securing Single Sign-On System With Executable Models.* Master Project, 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/WebCryptographyAPI/`

[4] Steffen M. Hansen, Jakob Skriver and Hanne Riis Nielson: *Using Static Analysis to Validate the SAML Single Sign-on Protocol.* Master Project, 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] Alternative tools: *Tools for analyzing protocols* `http://www.infsec.uni-trier.de/tools.html`

[11] Sergio Maffeis: *An expert in computer security.* Various publications on web security, security protocols and so on. `http://www.doc.ic.ac.uk/~maffeis/`

[12] NaCl: *Networking and Cryptography library.* `nacl.cr.yp.to`

[13] 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/`

[14] Nets-Danid: *NemID guide* https://www.nets-danid.dk/produkter/for_tjenesteudbydere/nemid_tjenesteudbyder/nemid_tjenesteudbyder_support/tjenesteudbyderpakken/

[15] Danny Dolev and Andrew C. Yao: *On the security of public key protocols.* Technical report, Stanford, CA, USA 1981.