IT University of Copenhagen

Master project

Securing Single Sign-On systems with executable models

Author:
Jakob Højgaard
jhoe@itu.dk

Supervisor:
Joseph Roland Kiniry
josr@itu.dk

February 28, 2013

Abstract

The use of digital identities has gradually become almost inevitable. Therefore securing digital identities have become an increasingly important topic. Reusing these identities across different security domains to provide Single Sign-On, only increases the importance. Given the intrinsic distributed nature of Single Sign-On systems, ensuring reliability and security is challenging at best. The Danish government operates such a system - *Nemlog-in* - providing Single Sign-On for all citizens when interacting on-line with any government authority. In this report I proposes the use of executable models to prove the security properties of such a system. I use Nemlog-in as the case and provide a detailed description of the Nemlog-in protocol which is based on the SAML Single Sign-On protocols. I use the F* programming language, a value-dependent, typed language with refinement types, to formalize the Nemlog-in protocol. I have formalize key parts of the protocol and proved it feasible to formalize the Nemlog-in protocol as a whole with F*. I have thereby shown that it is indeed possible to use a language like F* to formally specify the protocol and at the same time have an executable model.

Contents

Listings

List of Figures

Chapter 1

Introduction

Todays use of the Internet increasingly require some level of authentication for the user to be able to access the content. Some claim this could turn the Internet into a collection of walled gardens[1]. Never the less authentication is here to stay for good or for bad, which means that securing protocols for authenticating and authorizing users is getting even more important. The increase in protected content also means that people will have to either reuse passwords across a large amount of applications, remember a lot of different passwords (alternatively use password managers) or the content providers can reuse user identities across Internet domains. The latter is getting increasingly more popular since it seen from a user perspective makes it possible for the user to reuse very few identities on-line.

Large protocols for authentication and session management have been developed bt industry leaders. The two most elaborate and complex protocols are SAML, specified in [12], by the OASIS group and in WS-Federation specified by a joint effort by large companies like Microsoft, IBM and VeriSign. Lately other more lightweight efforts like OpenId and OAuth have gained popularity.

These protocols have been scrutinized by computer science researchers and by hackers, and flaws have been reported for most protocols. But very often the flaws reported are related to the implementation of the protocol rather than the specification of the protocol itself. For instance Somorovsky et al.[17] describes how SAML can be exploited because of an error in the way XML signature checks are performed in different XML libraries. Another example of errors in the implementation of SAML is described in [1] by Armando et al., where the error was caused by the fact that not all protocol elements were checked.

Vulnerabilities are not only reported through traditional research papers. Internet blogs are often used to publish new vulnerabilities. Two resent examples of this is Homakov[2] reposting the ability to hack Facebook exploiting flaws in the Chrome browser and the OAuth protocol. Another report[3] showed how deserialization of configuration data in the popular Ruby-On-Rails web framework allowed attackers to bypass authentication.

Therefore finding methods for building frameworks for such systems with built-in mechanisms for securing key properties will be of great advantage.

[1]http://www.guardian.co.uk/technology/2012/apr/17/walled-gardens-facebook-apple-censors

[2]http://homakov.blogspot.dk/

[3]http://ronin-ruby.github.com/blog/2013/01/09/rails-pocs.html

1.1 Objectives

In this project I set out to explore the plausibility of implementing a protocol like the Nemlog-in protocol in a provable secure way. The reason for this is that the methods for doing so today, vastly depend on best practice efforts. Searching for ways to formalize a protocol like Nemlog-in, I discovered the work done by Swamy et al.[18] and Fournet et al.[6]. They introduce F* Value-Dependent types to a programming language based on ML, in order to secure program security and reliability by typing.

In my protocol description I will utilize earlier work analysis of the SAML protocol by Hansen and Skriver[10] and Armando et al.[1]. Even though the analysis done by Hansen and Skriver related to a previous version of the SAML protocol, their experiences on modelling SAML are very useful.

The goals of this project is to:

1. Describe the Nemlog-in protocol as a whole, including the authentication done by DanId, to the extend that a formal specification of the entire protocol can be constructed.

2. Formalize the specification on Nemlog-in in F* (pronounced F Star) to the extent possible and assess the feasibility of completing the formalization of the protocol.

1.2 Scope

In this project I have been focused on describing the Nemlog-in protocol as accurate as possible, thus my description and analysis of the protocol will seek to capture as much as possible before formalizing the specification in F*. I have implemented the protocol in F* to the extend possible given the time-frame of the project. This means that the implementation will focus on proving feasibility rather than considering all aspects of the protocol. Note that this project only looks at the SSO profile (explained later) of SAML even though it would make perfect sense to include other profiles too e.g, the Single Logout Profile. Further it should be noted that this approach does not include all security properties for the protocol but rather the set of properties specifically modeled. I will however, suggest how to further secure implementation of security protocols like Nemlog-in.

1.3 Background

Nemlog-in is a public login federation[4] owned by the Danish government through the Agency of Digitization, henceforth DIGST. It is an important component in a very ambitious digitization strategy outlined by the DIGST, where all interaction between citizens and any public institution should be possible online from 2015. A key component for this to succeed is Nemlog-in. Nemlog-in has been in operation for several years supporting basic federated login and Single

[4]Login federation refers to the concept of federated identity where a user can link identities stored in different identity management system. The term *Federation* refers to a union of self-governing states, which is very applicable for this term

Sign-On. But as part of the digitization strategy it was decided to to upgrade Nemlog-in and consolidate it with functionality provided by other governmental systems. The tender for this contract was won by NNIT in December 2011. The new system called Nemlog-in 2, should be able to handle: 1) *Single Sign-On* as the existing Nemlog-in, 2) The *initial connection flow*, when adding new systems to the federation. In order for this to be possible a new service should be added, 3) *Signing.* This service should make it possible to sign legal document digitally. 4) A *user rights management* system that will make it possible for all the connected systems, called service providers, to manage user rights in the same system. Note here that Nemlog-in supports login for both citizens and employees. Late in the process a fifth system as decided called *Delegation*, which will make it possible for citizens to delegate authority to others, being institutions or other citizens.

Nemlog-in uses NemId for authentication of users. NemId provides 2-factor authentication through a cardboard key card. When the user attempts to login she is challenged with a number, that she can locate in the key card. The matching response is entered by the user and she is authenticated if the key matches. This means that keys can only be used once. Alternatively the user can purchase a digital version which works the same way as a RSA SecureID. Finally it has just been made possible to use a digital signature on own hardware through a small hardware device with USB. NemId is operated by Nets DanId which is owned by the Federal Bank of Denmark together with other larger banks in Denmark and Norway.

The release on Nemlog-in is done in 2 phases. First release was done on the 19th of December 2012 and contained only the SSO part of the solution. The remaining parts are scheduled for release during the summer of 2013. Compared to the previous Nemlog-in version solution the first release will not provide any new features apart from a change in technology.

In the contract[5] between DIGST and NNIT, several security measures were specified in order to secure a reliable and secure system. Among these requirements were:

- The system should build on COTS[6] software.
- A development process to ensure secure development.
  - Training of developers in secure development and common security issues.
  - Review of all code and all changes to it.
  - Unit-test on all public methods.
  - Static code analysis.
  - Static security analysis of source code.
- All source code should be reviewed by 3rd party.
- The system should be penetration tested in a production environment setup, by 3rd party.
- Official tests of the system should be approved by DIGST.

[5]The contract is publicly available and can be obtained by request to DIGST
[6]Commercial Off-the-Shelf Software

These initiatives are all result of mitigation by best practice, and for very good reasons. But from an academic point of view it would be interesting to be able to provide sound proof for some of the properties these method tries to mitigate.

1.4 Disclaimer

For the sake of transparency it should be noted that I work for NNIT, the vendor responsible for delivering the Nemlog-in 2 system to the DIGST. I have been responsible for the development of some of the core components included in the solution and therefore have deep insight and knowledge of how the protocol is actually implemented in production. However for this project I have, for the sake of openness only described properties that are publicly available, or at least indirectly public. Indirectly public should be understood as, information available to anyone possessing the necessary skills to do a login trace on the test Nemlog-in site, with a web debugger. To document this the appendix C will describe a login trace using the Fiddler 2[7] Web Debugger. The attached CD source with code will also contain data (a web debugger trace) to back this up.

[7]`http://www.fiddler2.com/fiddler2/`

Chapter 2

Protocol Description

In this section I will describe the basic concepts of the Nemlog-in system seen as a protocol. I will layout the description of the protocol by describing the protocol concepts individually, finishing the section with a complete protocol description. The protocol description(s) will be in the form of UML diagrams and formal notation depending on what will communicate the clearest.

2.1 Single Sign-on

The reason for having to login to a website, usually have to do with the content being protected. The protected content is often related to privacy issues, intellectual property or plain copyrights. As described in the introductory sections of this paper, single sign-on systems are getting more and more used in all sorts of online systems. This can be useful for both users and system owners in that the user only has to provide credential one time and that the system owners can reuse "standard" providers. The concept of single sign-on or SSO is thus the ability to reuse a login across different systems or domains - in this context often referred to as security domains. To explain this lets look at what a login means. A login or a session, is the outcome of a user authentication where the user provides his or her credentials to the system and in return obtains a session handle. Usually this handle is a session cookie[1] that on the server side can be used to identify the user and link to the result of the latest authentication. This session can then be terminated by the user when requesting a logout or when closing the browser[2] or by session time-out i.e., the user is inactive for some period of time.

In a SSO scenario or a federated setup, a 3rd party will be added to the login/logout flow usually referred to as the Identity Provider or IdP. In this scenario the system where the user request a login is referred to as the Service Provider or SP. Figure 2.1 illustrates a federated login (for now don't worry about the message labels, they are SAML specific and will be explained in 2.2) where the user request a login or a protected resource at the service provider and is then redirected to the IdP. The figures 2 and 2.2 includes a step where

[1]Session cookies only exists as long as the browser is open

[2]Later versions of the Google Chrome browser has a option making it possible to recover a session after the browser has been closed

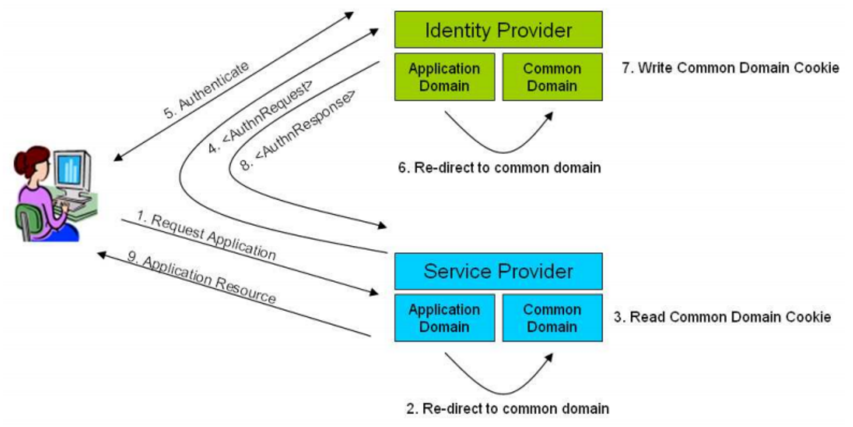


Figure 2.1: Sign-On in a federated setup from [4]

a *Common Domain Cookie* is read and written again later. This part is not important to the modelling of the protocol and will not be addressed any further. The IdP then prompts the user for credentials and if authenticated, redirects the user back the the service provider. The service provider can now serve the user with the requested content.

What actually happens is that, when the user is authenticated at the IdP, the IdP creates a session for the user that can be reused for any subsequent authentication requests. When the user is redirected to the service provider the service provider also creates a session with the user, so that the user will not have to be redirected to the IdP for every subsequent request for protected content. Figure 2.2 illustrates a login from the same user but at another service provider in the federation i.e., a SSO scenario.

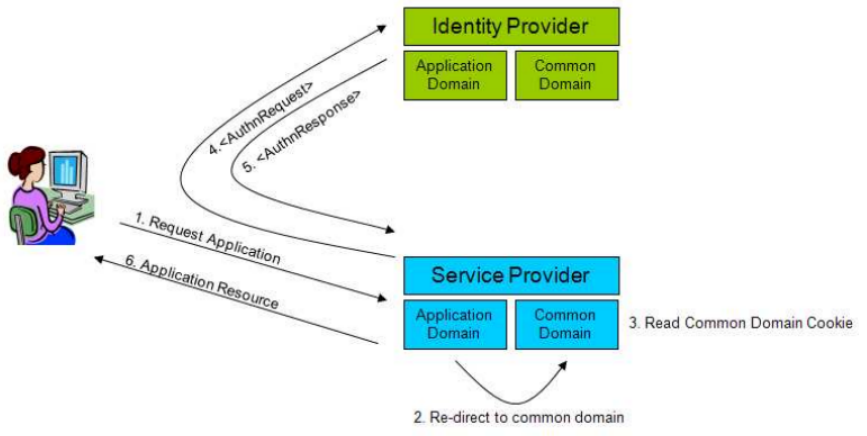


Figure 2.2: Single Sign-On in a federated setup from [4]

The difference between the two scenarios is that in the latter the user is never prompted for credentials. The initiation of the scenario however, is the same. The user request a login and since the user does not have a session with the service provider she is redirected to the IdP. Having just logged in through another service provider i.e., having a session, she is redirected back to the service provider (with authentication OK). The service provider can now create a session with the user.

This serves as an abstract description of SSO concepts. Examples of concrete protocol specifications that cover SSO are SAML and WS-Federation[3]. Nemlog-in has chosen the SAML protocol which will be described in further detail in the next section.

2.2 The SAML 2.0 protocol and artifacts

SAML or the Security Assertion Markup Language is an XML based language or standard created for exchange of authentication and authorization (also sometimes referred to as *security tokens*) between domains as explained in the previous section. SAML 2.0 is the latest version of the specification[12] and was released in 2005. Unless specifically specified when referring to SAML, version 2.0 is assumed. SAML specifies 5 core concepts, *Assertion*, *Protocols*, *Bindings*, *Profiles* and *Metadata*.

Assertions are security tokens containing statements or claims about a principal[4]. In SAML these statements are called attributes and will usually be things like name, last name, email etc. The principal is referred to as the *subject*. A representation of the SAML assertion and its sub elements is described i figure 2.3. For examples of assertion and SAML messages, please see appendix A.

$$
\begin{aligned}
Assertion := [&Id : String\\
&IssueInstant : DateTime\\
&Issuer : String\\
&Signature : String\\
&Subject : Subject\\
&Conditions : Conditions\\
&AttributeStatement : List < attribute >\\
&AuthnStatement : AuthnStatement]\\
\\
Subject := [&NameId : String\\
&InResponseTo : String\\
&NotOnOrAfter : DateTime\\
&Recipient : String]\\
\\
Conditions := [&NotBefore : DateTime\\
&NotOnOrAfter : DateTime\\
&AudienceRestriction : List < String >]\\
\\
AuthnStatement := [&AuthnInstant : DateTime\\
&SessionIndex : String]
\end{aligned}
$$

Figure 2.3: The SAML Assertion elements

[3]`http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-fed/WS-Federation-V1-1B.pdf`

[4]A principal in this context refers to an entity that can be authenticated, thus both the user and the IdP and SP are principals

Protocols describe the messages that are exchanged between the service provider and the identity provider when exchanging *assertions*. The protocol used in Nemlog-in SSO scenario is called the Authentication Request Protocol. Another protocol used in Nemlog-In but not addressed any further in this report is the Single Logout Protocol (SLO). The Authentication Request Protocol consists of a request message - AuthnRequest (figure 2.4) - and a response (figure 2.5). The field of the AuthnRequest are as follows; 1) The initiator of the authentication - called the *Issuer* in SAML. 2) The *Destination* i.e., the intended receiver of the request. Note that the content of the issuer field in the message referrers to the id of the service provider - denoted entity Id in SAML - where as the *Destination* referrers to the actual endpoint address, in this case the IdP. 3)The *Id* field for identifying the message, which needs to be unique. The creator of the message must ensure never to use the same id more than once. 5) *IssueInstant* is a time stamp from when the message was issued (UTC).

$$\begin{aligned} AuthnRequest := [&Id : String \\ &IssueInstant : DateTime \\ &Destination : URI \\ &Issuer : String \\ &Conditions : Conditions] \end{aligned}$$

Figure 2.4: The SAML AuthnRequest message

The response message (figure 2.5) contains the *Id* and the *IssueInstant* fields as described for the request message. The *Destination* is the endpoint address of the service provider and the *Issuer* is the entity Id of the IdP. Further the message contains an *InResponseTo* field containing the Id of the AuthnRequest for which it is a response. The status element of the response can have only one of 3 three possible values, *Requester*, *Responder* or *Success*. The first two indicates who is responsible for the error. An *Assertion* can only be contained in a response with success status (explained earlier). However the assertion can be in two different forms, either as plain text or as an *EncryptedAssertion* where it will be encrypted under the public key of the recipient of the message i.e., the service provider.

$$\begin{aligned} Response := [&Id : String \\ &IssueInstant : DateTime \\ &Destination : URI \\ &InResponseTo : String \\ &Issuer : String \\ &Status : String \\ &Assertion : Assertion] \end{aligned}$$

Figure 2.5: The SAML Response message

Bindings describe how the messages are mapped to the underlying HTTP(s) or SOAP protocols. In this report only the HTTP POST and HTTP REDIRECT bindings will be addressed, however for the SLO protocol in Nemlog-in the SOAP binding can also be used. For the REDIRECT binding the message content is mapped the URL query string with SAMLRequest or SAMLResponse as identifier. But since a URL query string has a very limited capacity the message is compressed using the DEFATE algorithm, then BASE64 encoded and URL encoded (in that order) to ensure interoperability. Usually only short messages like AuthnRequests use this binding due to the capacity issues. The POST binding maps message content to hidden XHTML form fields named SAMLRequest, SAMLResponse respectively. The message content is BASE64 encoded. For both bindings more information like message signature, signature algorithm and Relay State (further explained in section 2.3) might be included and is mapped to the binding in the same way as the message itself.

Profiles describes how the *assertion*, *protocol* and *binding* are used together to fulfill a specific requirement or use case like SSO. It might mandate otherwise optional protocol or binding elements. Thus viewing the profile concept as sort of a materialization of a whole use case, will serve as an appropriate abstraction. In the following section (section 2.3) the Danish specialization of the SAML Web Browser SSO Profile called "OIO Web Browser Profile" or simply "OIOSAML" will be described in further detail.

SAML *Metadata* is necessary in order for the involved parties to know each other. Usually exchange of metadata is part of a more formalized "connection" process. For service providers metadata contains the certificate and endpoints. The service providers specifies a certificate for signing messages and a message that the IdP should use for encrypting content (assertions). The endpoints specify the addresses to which the IdP should send response messages. The metadata can contain more endpoints if different bindings are available. For the SSO profile the metadata also contains the attributes in the assertion expected when performing sending an *AuthnRequst.* Metadata for Identity Providers also contains signing and encryptions certificates. The endpoints specified describe the addresses for sending requests to the IdP. Here the IdP can also provide endpoints with more than one binding. The IdP will publish the attributes it is able to serve through it's metadata.

2.3 Nemlog-in and the OIOSAML SSO profile

The SAML profile used for Nemlog-in is a dialect of the SAML "Web Browser SSO" called "OIO Web SSO Profile" or simply OIOSAML. OIO is a Danish abbreviation for "Public Information Online" ("Offentlig Information Online")[5] and serves as an umbrella for most standards when interacting with the government and other public institutions.

The profile is as mentioned a dialect or rather a specialization of the SAML "Web Browser SSO" profile, with specific requirements for binding and protocol. These measures have been added to further secure the protocol and some of the extensibility of the SAML profile is simply not necessary in the setup of Nemlog-In. One example is that in plain SAML it is possible to issue authentication requests without prior exchange of metadata. This is not desirable for Nemlog-in since security will be greatly improved by adding a controlled connection flow, where service providers are approved before entering the federation. Note that in the following description some aspects might have been left out if deemed insignificant to the level of description necessary for this report.

[5]The OIO standards is catered by `http://digitaliser.dk`

For the Service Provider OIOSAML mandates the following:

- Service provider metadata should be known to the IdP i.e., a formal exchange of metadata must have been carried out.
- Binding for *AuthnRequest* should be HTTP REDIRECT with the message DEFLATE encoded.
- Transport should be over (one-way) SSL/TLS
- The *AuthnRequest* must be signed and the signature placed as a query parameter identified by "Signature".
- The *RelayState* must be opaque[6]

For the Identity Provider OIOSAML mandated that:

- Service provider metadata should be known to the IdP i.e., a formal exchange of metadata must have been carried out.
- If the response is an error response the IdP must not include any assertions
- Successful responses can contain only one assertion
- The assertions must state the level of authentication achieved[7]
- Response messages should be sent to the service provider using the HTTP POST binding.
- Transport should be over (one-way) SSL/TLS
- Response messages should not be signed, but instead the embedded *Assertion* should be signed and encrypted.

OIOSAML also mandates that the certificates specified in metadata should be OCES certificates (explained in section 2.4) i.e., issued by DanId.

For assertions OIOSAML mandates the following

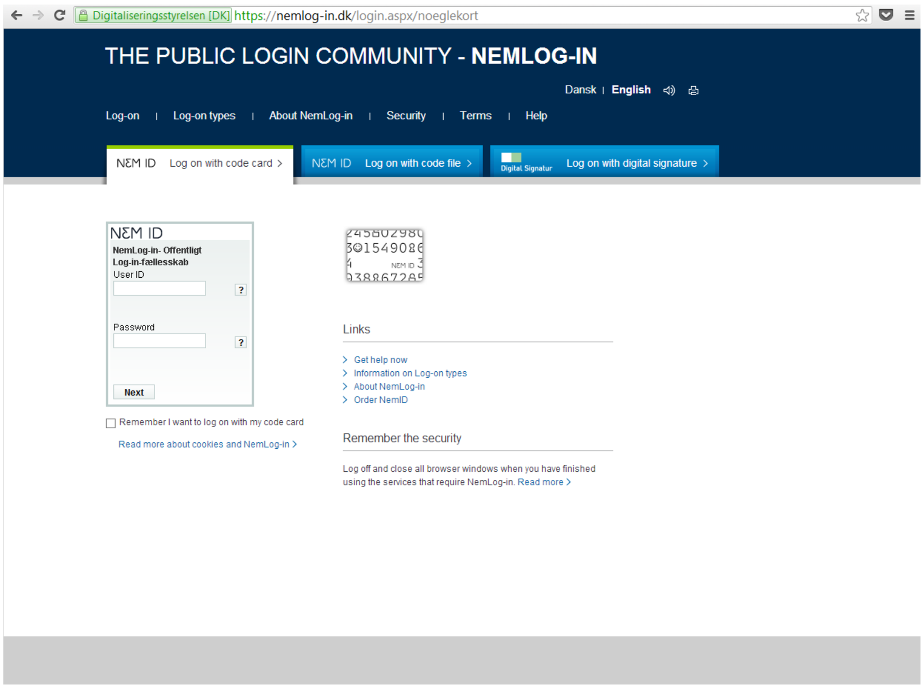
- Assertions must contain only one AuthnStatement and one AttributeStatement.
- The assertion must be encrypted with the certificate received from the service provider through metadata.
- The issuer field (the entity id of the IdP) must be included in the assertion
- The assertion must be signed with the certificate described in the metadata exchanged with the service provider.

[6]RelayState is a mechanism that the service providers can use to associate any subsequent profile messages with the original request (could be the protected resource requested by the user)

[7]Authentication level referrers to what level of authentication achieved from the subject on a scale from 1-5 where 5 is highest - `http://ec.europa.eu/information_society/activities/ict_psp/documents/authentication_levels.pdf`. Since the current Nemlog-in solution only supports level 3 this will not be addressed any further in this report

2.4 Authentication with NemID

In the context of a federated login system the authentication of the users is done by the identity provider. In the Nemlog-in setup the authentication is brokered to a 4th participant called DanId through the national digital identification system *NemId* (easy Id)[11] or the older variant *Digital Signatur*. The two are actually aliases for their certificate type OCES II[8] and OCES I respectively. As noted earlier it important to stress that NemId (the OCES II variant) come in 2 variants one for citizens and one for employees, which means that Nemlog-in has 3 different authentication mechanisms in total2.6. This section will explain the work-flow for identifying the user at the IdP.


Digitaliseringsstyrelsen [DK] https://nemlog-in.dk/login.aspx/noeglekort
THE PUBLIC LOGIN COMMUNITY - NEMLOG-IN
Dansk | English
Log-on | Log-on types | About NemLog-in | Security | Terms | Help
NEM ID Log on with code card >
NEM ID Log on with code file >
Digital Signatur Log on with digital signature >
NEM ID
NemLog-in- Offentligt Log-in-fællesskab
User ID
?
Password
?
Next
Remember I want to log on with my code card
Read more about cookies and NemLog-in >
Links
Get help now
Information on Log-on types
About NemLog-in
Order NemID
Remember the security

Log off and close all browser windows when you have finished using the services that require NemLog-in. Read more >

Figure 2.6: Screenshot of the Nemlog-in website

NemId is actually a mechanism that spans several concepts. First of all it is a way of digital authentication, but it is also a way to centrally store user certificates used for authentication. The vehicle for authentication has in spite of quite a bit of controversy been chosen to be a Java Applet and a card board key-card. The justifications for this choice have be many, and I will not discuss these further. Instead I will describe the basic messages exchanged during authentication in detail. In later sections I will discuss the possible benefits of using other approaches for this part of the protocol. NemId is a joint venture between the Danish governmental authorities and the financial sector in Denmark. In the context of Nemlog-in, the result of the authentication, is that the public part of the users OCES certificate is returned to the IdP, but when used by the banks the applet returns a nonce.

[8] OCES - Offentlige Certifikater til Elektronisk Service - Public Certificates for Electronic Services

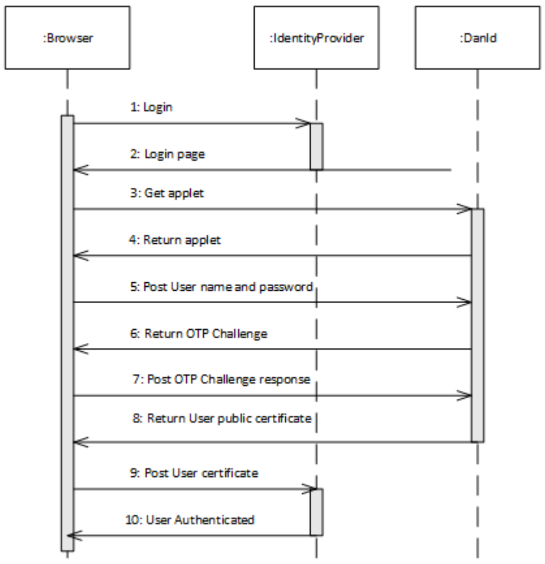


Figure 2.7: Sequence diagram of NemId authentication

Figure 2.4 shows the message flow during authentication with the NemId OTP (One Time Password) applet - the first tab in figure 2.6. When the user request a login at the IdP she is presented with a HTML page that will request an applet from DanId. The applet has been configured with a challenge generated by the IdP and a string specifying the site using the applet. These parameters are signed by the IdP for verification by DanId. This applet will contain 2 fields, one for user name and one for password. Posting these to DanId will trigger a user lookup, a validation of the password and a look up of a challenge key. This challenge will be transmitted back to the user in the applet. The user then responds with the code matching the presented challenge. If authenticated the applet posts a response back to the IdP via the browser. This response will contain the applet, the 2 properties configured on the IdP (challenge and site) and also the users public part of her OCES certificate. The response data is signed with the user certificate. If all passes, an assertion is issued by the IdP.

2.5 A protocol walk-through

Compiling the descriptions from the previous sections a complete login flow can be described with the sequence diagram in figure 2.8.

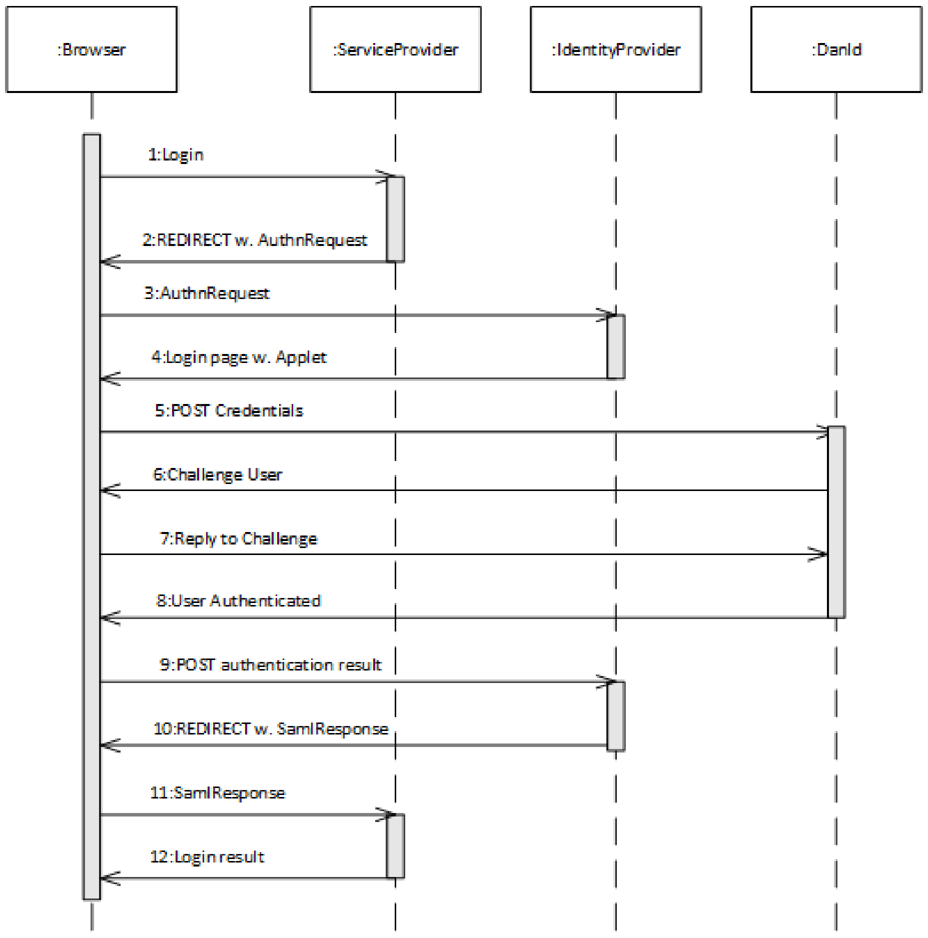


Figure 2.8: Sequence diagram of simple login

Chapter 3

Modelling the protocol

In order to build a model of the protocol, the descriptions in the previous chapter needs to be further refined and detailed. Each message flowing between protocol participants needs to be formalized and every processing rule needs to be formally described. In this section I will build on the information described in chapter 2 and break these into protocol element and processing rules.

3.1 Protocol prerequisites

In Section 2.3 I described OIOSAML. In this section these properties and their derived requirements are treated as prerequisites for the modelling. OIOSAML mandates the use of one-way SSL/TLS for all bindings. Note that there is no explicit requirement regarding version of SSL or TLS, even though version 2.0 of SSL is generally considered deprecated. In this report I assume SSL version 3.0 or TLS version 1.0. In order for the identity provider to only issue assertions to known service providers, SAML Metadata has to be exchanges before any assertion can be issued. In order to further restrict the service providers, the certificates used for signing and encryption needs to be issued by DanId i.e., the certificates have to be OCES certificates. Both the service provider and the identity provider are required to check that the certificates are valid and have not been revoked, when performing signature check and encryption. The general way for SAML requests and responses to travel from the service provider to the identity provider is trough the users browser via HTTP-GET or HTTP-POST redirects. Hence it is assumed that the browser will follow redirects. For response messaged, OIOSAML mandates the use of HTTP-POST REDIRECT binding. Thus the users browser needs to have JavaScript enabled. The service provider and the identity are assumed to be different entities residing in different domains.

To summarize:

1. All transports are SSL(3.0)/TLS(1.0).
2. Metadata has been exchanged as mandated by OIO SAML.
3. Certificate used by IdP and SP for signing and encryption must be issued by DanId (Only OCES certificates).

4. Signature check and encryption causes certificate validation/revocation check.
5. The browser will follow redirects.
6. JavaScript must be enabled in the users browser.
7. SP and IDP are different entities.

3.2 Formalizing protocol messages

Figure 2.8 in section 2.5 shows the log-in flow using a UML sequence diagram. It has all the necessary protocol participants, henceforth Principals, and messages, but the message content is not explicit. The flow also only describes one success scenario with one login, thus neither the SSO scenario or any failure scenarios are described.

Figure 3.1 depicts the complete Nemlog-in protocol as a UML Communication diagram. This will serve as reference for the rest of this chapter. The figure contains five principals; the user through the browser denoted as U, the service providers 1 and 2 denoted SP1 and SP2, the identity provider denoted IdP and lastly the authenticator DanId, simply denoted DanId.

The representation of message exchange in figure 3.1 does not follow any specific formalism, but obeys following syntax. Messages are named in accordance with their HTTP protocol verb e.g., GET instead of HTTP-GET. REDIRECT can refer to either HTTP-GET or HTTP-POST redirects, but for the purpose of this description, that bares no significant importance. Since the destination is not directly implied by the arrow in the diagram, the first parameter for REDIRECT messages carries the destination. RESPONSE messages refer to simple HTTP response messages and might be associated with a HTTP status code if not 200 (status OK) or 302 (Redirect), thus indicating an error. The message items are enclosed in parentheses and separated by comma. These items might map to different HTTP protocol artifacts like cookies, query parameters or form fields, but for the purpose of formalizing the protocol that bares no significance.

3.3 Message processing

The diagram in figure 3.1 describes the messages exchanged between the principals but it does not reveal anything about how the messages should be interpreted and processed. This could be done by annotating the diagram with comments containing the processing rules in pseudo code. This would however, clutter the diagram too much. Instead I will describe the processing rules as algorithm parts, referring back to the diagram by the in-going message number. For instance the processing done by SP1 in response to message 1, I will refer to as Process 1. Since the role of the browser is to route messages from the SP to the IdP and back no formal description of message processing will be described here. However in section 4.6 I will discuss the use of the browser role for modelling adversaries. Process 1 and Process 13 are the same since they are both request for a protected resource at a service provider, where the user does not already have a session. Processing for both are described in algorithm 1.

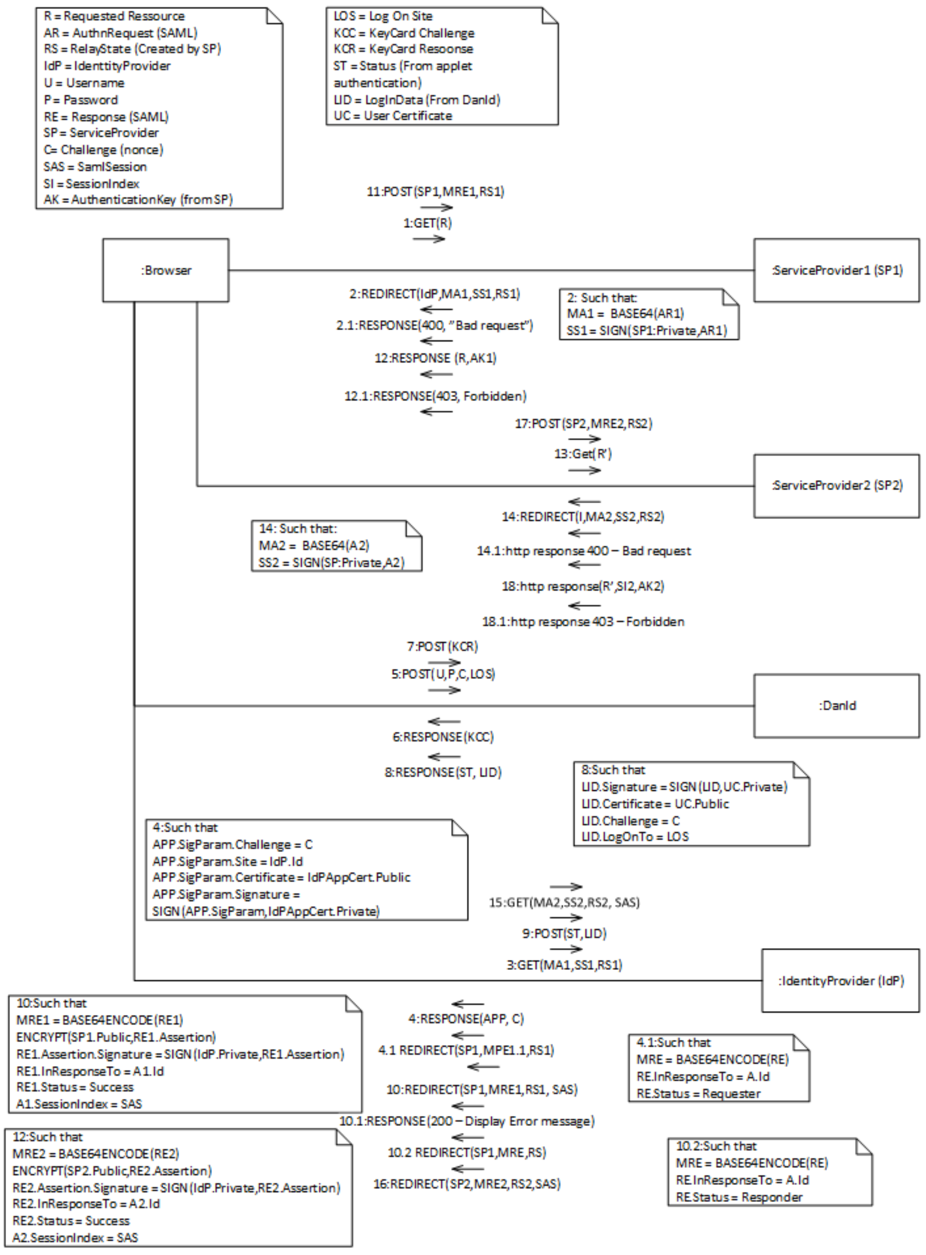


Figure 3.1: Communication diagram of the complete protocol

Algorithm 1 Process 1 and Process 13

```
Require: $GET$ is well-formed and IdP.Public and SP.Private
  if $R$ exists then
    $AR \leftarrow CreateAuthnRequest()$
    $SS \leftarrow SIGN(M, SP.Private)$
    $MA \leftarrow UrlEnc(Base64Enc(DEFLATE(AR)))$
    $RS \leftarrow UrlEnc(Base64Enc(R))$
    return $REDIRECT(IdP, MA, SS, RS)$
  else
    return $RESPONSE(400, Bad - Request)$
  end if
```

Process 3 describes the processing at the identity provider when an *AuthnRequest* is received. This message will either result in a response where the user is challenged for credentials or a SAML response for the identity provider indicating an error.

Algorithm 2 Process 3

```
Require: $GET$ is well-formed and IdP.Private and SP.Public and IdPAppCert.Private
  $AR \leftarrow UrlDec(Base64Dec(DEFLATE - D(MA)))$
  if $VERIFY(AR, SS, SP.Public)$ then
    $C \leftarrow GenChallenge()$
    $APP \leftarrow GenApplet()$
    $APP.SigParam.Challenge \leftarrow C$
    $APP.SigParam.Site \leftarrow IdP.Id$
    $APP.SigParam.Certificate \leftarrow IdpAppCert.Public$
    $APP.SigParam.Signature \leftarrow SIGN(APP.SigParam, IdpAppCert.Public)$
    $ARC \leftarrow CreateCookie(MA, SS, RS)$
    return $RESPONSE(APP, ARC)$
  else
    $RE \leftarrow CreateResponse()$
    $RE.InResponseTo \leftarrow AR.Id$
    $RE.Status \leftarrow "Requester"$
    $MRE \leftarrow Base64Enc(RE)$
    return $REDIRECT(SP, MRE, RS)$
  end if
```

For this project, the criteria for what will cause an error response is kept to a valid signature and that the signer is known i.e., that metadata has been exchanged. But in reality SAML and OIOSALM dictates a lot of very specific processing rules regarding more or less every possible element of the possible combination of message elements. This will not be included in this description for the sake of communication. Another certificate will be introduced here, called IdPAppCert. This certificate is a certificate issued by DanId and registered for use with the applet. Algorithm 2 shows how the certificate is used for signing the parameters to the applet. Lastly the original AuthnRequest including signature is included in the response in order for the IdP to remain more stateless.

Algorithm 3 Process 4

```
Require: $U$ and $P$ and Browser supports Java
  $SigParams \leftarrow APP.SigParams$
  if $VERIFY(SigParams, SigParams.Signature, SigParams.Certificate)$
  then
    $C \leftarrow SigParams.Challenge$
    $LOS \leftarrow SigParams.Site$
    return $POST(U, P, C, LOS)$
  else
    print $ERROR$
  end if
```

Even though stated earlier, that the browser processing will not be modeled, process 4 (algorithm 3) takes place at the browser, but through the DanID applet. Since the documentation of the design of the interaction between the applet and the DanId servers are not publicly available, I have based the process description in algorithm 3 on what can be deducted by analyzing and debugging the source code examples and integration descriptions provided at the DanId service provider support page[3]. This means that algorithm 3 will be a best guess. However the parameters in message 4 are correct, only the exact processing and the exact message exchange between the applet and DanId can not be verified. Further the applet expects user input in the form of user name U and password P.

In the algorithm it is expected that these are already provided. This is not the correct behavior but for this exercise it will suffice. Lastly the applet is responsible for negotiating a secure tunnel between the applet and DanId, this will not be modeled here but rather assumed to be a secure transport like the assumption on HTTPS for the rest of the protocol.

Algorithm 4 Process 5

```
Require: Secure connection to applet
  if $U$ is known and $P$ matches then
    $KCC \leftarrow GetNextChallenge(U, P)$
    return $POST(KCC)$
  else
    return $POST(ERROR)$
  end if
```

As for algorithm 3, algorithm 4 is a best guess, and describes the expected behavior at DanId. Therefore this algorithm does not describe error conditions in detail since that would be too much guessing, therefore not part of further modelling.

Algorithm 5 is constructed on the basis of what the *LogInData or LID* (see appendix D for example) looks like and how it is processed in the OOAPI[1]. The key point here is that the response contains the public part of the users OCES

[1]Open source implementation of functionality necessary to use NemId for authentication, maintained bu DanID

certificate, the challenge generated by the requester - in this case the IdP - and the *LogOnTosite* - the identifier for the IdP. All of these response parameters are then signed with the users private key from the certificate. Note that the algorithm is written so that it looks like the private part of the users certificate will be in the hands of the DanId web server, where in reality it will be securely stored in a Hardware Security Module *HSM*. The signing will be done by that particular box and the private key will never be released outside that box.

Algorithm 5 Process 7

```
Require: Secure connection to applet
  if $VERIFY(KCC, KCR)$ then
    $UC \leftarrow RetrieveUserCert(U)$
    $LID \leftarrow CreateLogInData()$
    $LID.Certificate \leftarrow LID.Certificate$
    $LID.Challenge \leftarrow C$
    $LID.Site \leftarrow LOS$
    $LID.Signature \leftarrow SIGN(LID, UC.Private)$
    $ST \leftarrow "OK"$
    return $POST(ST, LID)$
  else
    $ST \leftarrow "ERROR"$
    return $POST(ST)$
  end if
```

Algorithm 6 describes the processing of the response from DanId routed to the IdP via the applet and the users browser. For this report the content of the assertion - the attributes - will be properties from the users certificate denoted *UC*. However in the context of Nemlog-In attributes can be retrieved from other sources such as the CPR-number[2] register as well. But for the modelling of this protocol it is not significant. Note that some properties of the assertion is left out to enhance readability.

The service providers' processing of response from the IdP will follow the same rule no matter which service provider it is and whether it is a result of the first login to the federation or a subsequent login resulting in a SSO. Thus the algorithm 7 describes processing of both message 11 and message 17. The result of the processing is, if the authentication is successful, that the service provider creates a local session with the user i.e., setting a session cookie and returning the original requested resource. If the authentication is not successful the user is denied access to the resource. For illustrative purposes check for assertion validity (they are usually only valid for a few minutes) are left out (it was also left out of algorithm 6).

Algorithm 8 describes the issuing of assertions in a SSO scenario. This algorithm could have been described as a part of algorithms 2 and 6, but to make it easier to follow, it is split into three descriptions. This will however create a little redundancy in the three algorithms. Issuing assertions in a SSO scenario requires that the IdP is able to obtain the public part of the users certificate. This can be done in 2 ways; either by the IdP storing the users

[2]CPR is personal identifier and every citizen in Denmark has one. The CPR number is considered confidential

Algorithm 6 Process 9

```
Require: $POST$ is well-formed and IdP.Private and SP.Public
  if $ST = "OK$ and VERIFY(LID,LID.Signature,LID.Certificate) then
    $MA \leftarrow ARC.AR$
    $SS \leftarrow ARC.SS$
    $RS \leftarrow= ARC.RS$
    $AR \leftarrow UrlDec(Base64Dec(DEFLATE - D(MA)))$
    if $VERIFY(AR, SS, SP.Public)$ then
      $A \leftarrow BuildAssertion(LID.Certificate)$
      $SI \leftarrow GenerateSessionIndex()$
      $A.InResponseTo \leftarrow AR.Id$
      $A.Issuer \leftarrow IdP.Id$
      $A.Audience \leftarrow SP.Id$
      $A.SessionIndex \leftarrow SI$
      $A.Signature \leftarrow SIGN(A, IdP.Private)$
      $EA \leftarrow ENCRYPT(A, SP.Public)$
      $RE \leftarrow CreateResponse()$
      $RE.Assertion \leftarrow EA$
      $RE.InResponseTo \leftarrow AR.Id$
      $RE.Status \leftarrow "Success"$
      $MRE \leftarrow Base64Enc(RE)$
      $SAS \leftarrow CreateSAMLSession(SI, SP.Id, LID.Certificate.Subject)$
      return $REDIRECT(SP, MRE, RS, SAS)$
    else
      $RE \leftarrow CreateResponse()$
      $RE.InResponseTo \leftarrow AR.Id$
      $RE.Status \leftarrow "Requester"$
      $MRE \leftarrow Base64Enc(RE)$
      return $REDIRECT(SP, MRE, RS)$
    else
      return $RESPONSE(ST)$
    end if
  end if
```

Algorithm 7 Process 11 and 17

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

Algorithm 8 Process 15

```
Require: $GET$ is well-formed and IdP.Public and SP.Private and IdPAppCert.Private
  $AR \leftarrow UrlDec(Base64Dec(DEFLATE - D(MA)))$
  if $VERIFY(AR, SS, SP.Public)$ and $IsValidSession(SAS)$ then
    $UC \leftarrow GetUserCert(SAS.Subject)$
    $A \leftarrow BuildAssertion(UC)$
    $SI \leftarrow GenerateSessionIndex()$
    $A.InResponseTo \leftarrow AR.Id$
    $A.Issuer \leftarrow IdP.Id$
    $A.Audience \leftarrow SP.Id$
    $A.SessionIndex \leftarrow SI$
    $A.Signature \leftarrow SIGN(A, IdP.Private)$
    $EA \leftarrow ENCRYPT(A, SP.Public)$
    $RE \leftarrow CreateResponse()$
    $RE.Assertion \leftarrow EA$
    $RE.InResponseTo \leftarrow AR.Id$
    $RE.Status \leftarrow "Success"$
    $MRE \leftarrow Base64Enc(RE)$
    $SAS \leftarrow AddSAMLSession(SI, SP.Id)$
    return $REDIRECT(SP, MRE, RS, SAS)$
  else
    $RE \leftarrow CreateResponse()$
    $RE.InResponseTo \leftarrow AR.Id$
    $RE.Status \leftarrow "Requester"$
    $MRE \leftarrow Base64Enc(RE)$
    return $REDIRECT(SP, MRE, RS)$
  end if
```

certificates locally through the duration of the session or by the IdP looking up the certificate at the DanId LDAP certificate service. For this algorithm it is assumed that the certificate is persisted locally and can be looked up using the SAMLSession key.

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

Chapter 4

Modelling with F*

In this chapter I will introduces the language F* and domonstrate how it can be used to model a security protocol. F* can be considered a formal specification language that is also executable. The F* project describes it as a *Verifying compiler for distributed programming.* This chapter describes how F* is used to start building a formal specification of the Nemlog-in protocol and how it can be executed.

4.1 Introducing F*

F* is a research language developed at Microsoft Research, henceforth MSR, and is at the time of writing considered to be an α-release. The F* project primarily subsumes two earlier projects from MSR, F7[1] and Fine[2], both working on providing refinement types for F#. F* is as F7, Fine and F# for that matter, a dialect of ML and compiles to .NET bytecode, thus enabling interop with especially F# code. It also provides access to libraries for networking, cryptography and concurrency through the .NET framework. The .NET byte-code generated by the F* compiler preserves types, which means that types defined in F* can be used by other .NET languages without loosing type information. Compiling and type-checking F* code utilizes the Z3[3] SMT solver for proving typing assumptions made with refinement types described in section 4.3. F* has it self been formalized and verified in Coq[4].

4.2 Syntax and semantics

As described F* is a dialect of ML and inherits syntax and semantics from it. This makes F* a functional language with things like immutability by default, polymorphic types and type inference. Listing 4.1 shows the classic *Hello world* example in F*. It could have been written a bit simpler, but this example also demonstrates how to explicitly specify types with the *colon* operator. Given

[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, often used as benchmark - http://coq.inria.fr/

type inference type annotation can in most cases be omitted. The example also demonstrates the F* syntax for a *main* method or program entry-point. This is done by defining a function _ (underscore), at the end of a module. This will instruct the compiler to create an .exe file instead of a dll.

```
module hello

let _ =
  let x:string = "hello world" in
  print_string x
```

Listing 4.1: Hello world example.

The example in listing 4.2, demonstrates the *val* declaration for defining function signatures. In the example a simple method for summing 2 integers are defined with the *val* declaration, and implemented with the subsequent *let* binding. F* uses the currying syntax for signatures, thus tupled arguments are not supported for input parameters. Note here that omitting the *let* binding and implementation will not make the compilation fail, but rather give the following warning. *Warning: Admitting value declaration Sum.sum as an axiom without a corresponding definition.* In other words the signature promised by the val binding was valid, but there was no concrete implementation of the *sum* method to support the claim.

```
module Sum

val sum: x:int -> y:int -> z:int
let sum x y = x+y

let tree = sum 1 2
```

Listing 4.2: Sum example demonstrating function signatures.

4.3 Refinement types

One of the features of F* that is derived from F7 and Fine is that of *Refinement Types*. Refinement types are type safe refinements of existing types. This makes it possible to refine or restrict values more than their original type. The code example in listing 4.3 illustrates this by creating a new type *pos* that is a refinement of *int*. The states that *pos* will always be larger than 0. The example also demonstrates an attempt to assign 0 to a type of *pos* will make the program fail type checking.

```
(* Declare a type pos of positive numbers *)
type pos = i:int{i>0}

let x:pos = 1
let y:pos = 0 (* Will not type-check*)
```

Listing 4.3: Simple refinement types.

Refinement types are created by decorating an existing type with an expression enclosed in curly brackets. In this simple example the refinement expression is a simple boolean expression. But refinements are not limited to boolean expressions, F* extends this through it's kind-system. Without going into detail about kind theory[5], kinds can be seen as an abstraction over types. This means that types are *of* a kind or that types *have* a kind. F* uses the *-kind to denote 'regular' types, hence the name F*. The term 'regular types', here cover all the types possible to create in a regular type system for a programming language like C#, Java or ML for that matter. Refinement types are not of the *-kind but rather the E-kind. E-Kinds or **E**rasable kinds bares no runtime significance, thus only have effect during type checking an compilation. Further use of the E-Kinds will be explained in the following sections describing how is is used to formalize the Nemlog-in protocol. F* has several other kinds that could be used with the implementation of Nemlog-in. But since time did not allow use of these in this project, they will not be described in this section.

4.4 Formalizing Nemlog-in by example

The F* website[14] and the interactive guide[13], has an example of a simple authentication protocol implemented with F*. This example builds on work done with F7 explained in a technical report by Fournet et al.[6]. This example has been used as reference for implementing the first sample fragment of the Nemlog-in protocol. The code fragment in listing 4.4 is the formalization of the algorithm 1 in section3.3.

The algorithm 1 is implemented in the code as the function *serviceprovider* in the bottom of the listing. The function is an implementation of the signature declared with the **val** binding just above it. The function takes 6 arguments and returns **unit** - which in most dialects of ML means the same as **void** in **C#** or **Java** like languages. The arguments declared in the signature, specifies 1) an abstract type principal for identifying the service provider, 2) a public key for the service provider, 3) a private key for the service provider, 4) a identifier for the client, 5) and identifier for the IdP and lastly 6) the public key for the IdP.

The *pubkey* and *privkey* types are declared using F*'s syntax for constructing dependent types (the double colon), or more specifically types of a specific kind - in this case the *-kind. The type declaration should be read as; a type of *pubkey* will have a constructor that takes a *principal* and returns a type of *-kind. Note that for this example the type is still abstract and has no actual constructor.

[5]Joseph Kiniry's Ph.D Dissertation on Kind Theory - `http://www.itu.dk/~josr/final_draft.pdf`

```
module Serviceprovider

type principal
type pubkey :: principal => *
type privkey :: principal => *
type dsig
type uri
type SamlMessage =
  | Login: uri -> SamlMessage
  | AuthnRequest: issuer:principal ->  destination:principal
      -> message:string -> dsig -> SamlMessage
  | Failed: int -> SamlMessage

type Log :: principal => string => E

val Send: principal
        -> SamlMessage
        -> unit
val Receive: principal
        -> SamlMessage
val CreateAuthnRequestMessage: issuer:principal
        -> destination:principal -> string
val Sign:  p:principal
        -> privkey p
        -> msg:string{Log p msg}
        -> dsig
val VerifySignature: p:principal
        -> pubkey p
        -> msg:string
        -> dsig
        -> b:bool{b=true ==> Log p msg}


val serviceprovider:  me:principal -> pubkey me -> privkey me
    ->
                      client:principal -> idp:principal ->
                          pubkey idp -> unit
let serviceprovider me pubk privk client idp pubkidp =
 let req = Receive client in (*1*)
 match req with
  | Login (url) ->
    let authnReq = CreateAuthnRequestMessage me idp in
    assume(Log me authnReq);
    let sigSP = Sign me privk authnReq in
    let resp = AuthnRequest me idp authnReq sigSP in
    Send client resp (*2*)
  | _ -> Send client (Failed 400)(*2.1*)
```

Listing 4.4: Simple implementation of the service provider.

Parameters to these type constructors can have more arguments than just one as in this example. In fact they can also be nested. Parameters will be separated by the double arrow ⇒ operator.

Looking at the implementation of the algorithm 1, the service provider starts by receiving a message from the client. This is also a signature promising that given a principal a *SamlMessage* will be returned. Compared to the algorithm, the example does not check if the requested resource exists, but rather matches the request with a *SamlMessage.Login* type and if it matches creates an *AuthnRequest*. For simplicity most arguments to the creation method has been omitted.

Now the signing of the *AuthnRequest* utilizes a few of the features of F* to ensure that the service provider is actually the one signing the message. The signature for signing messages takes 4 arguments, the principal - the signer, the private key of the principal - declared dependent on the principal, and the message to sign. The message is annotated with a refinement type **{**Log p msg**}** as can be seen from the type declaration is an E-kinded type. It takes a principal and a string as constructor elements. Now the sign methods requires that the message to sign is related to the principal signing it i.e., the predicate **{Log p msg}** to be true before the function can type-check. In order to secure this, the **assume** construct is used. By calling **assume(Log me authnRequest)** before signing the message the predicate is thereby "validated". Even though the **validatesignature** function is not used in the example, it makes sense to explain the function signature here. The function takes a principal, its public key, a message and a signature. It the returns a boolean indicating if the check passed. But the return type has a refinement type that, if the validation passes relates the message to the principal (note that ==> is to be read as implication) thus stating that the predicate is valid.

After the *AuthnRequest* has been signed, the signature, SP and the IdP is wrapped in an *AuthnRequestMessage* and sent back to the browser.

4.5 Nemlog-in specified in F*

The code in this section represents the state of the project now, thus not in any way a complete implementation of the protocol. Implementation was carried out in an incremental manner, focusing on solving core challenges first and then adding more functionality later. Not all code written during the project has made it into the main part of the report. These code examples will all be included in the enclosed CD. All source code produces as part of this project can also be found on the source code sharing community Github[6].

The F* code for the prototype is organized in 5 modules:

1. The SamlProtocol module
2. The Crypto module
3. The Certificate Store module
4. The Service Provider module
5. The Identity Provider module

The modelling follows the principles for cryptographic protocol modelling outlined by Dolev & Yao [5], in that cryptographic primitives are assumed to be perfect and cyphers cannot be decrypted without being in procession of the proper decryption key i.e., the matching private key. Each module is listed on the following pages. In the following I will explain key principles for each module and for the service provider and the relation to the algorithms in chapter 2.

[6]The source code repository for this project can be found at `https://github.com/hgaard/FStar-saml-sso`

4.5.1 Specification of the SAML Protocol

```
module SamlProtocol

open Crypto

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

type AuthnRequest =
  | MkAuthnRequest: id:string -> 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 Assertion =
  | SignedAssertion: assertiontoken -> dsig -> Assertion
  | EncryptedAssertion: cypher -> Assertion

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

type SamlMessage =
  | Login: uri -> SamlMessage
  | LoginResponse: string -> SamlMessage
  | AuthnRequestMessage: issuer:prin ->  destination:endpoint
     -> message:string -> dsig -> SamlMessage
  | AuthResponseMessage: issuer:prin -> destination:endpoint
     -> Assertion -> SamlMessage
  | UserAuthenticated: status:string -> loginData:LoginData ->
      authnRequest:AuthnRequest -> SamlMessage
  | UserCredRequest: challenge:nonce -> 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:id -> assertiontoken
val AddSignatureToAssertion: assertiontoken -> dsig ->
    signedtoken
val EncryptAssertion: receiver:prin -> pubkey receiver ->
    signedtoken -> Assertion
val DecryptAssertion: receiver:prin -> privkey receiver ->
    Assertion -> (signedtoken * dsig)
```

Listing 4.5: Formalization of protocol elements.

The *SamlProtocol* module is responsible for 2 things, the specification of messages and to provide functions for sending and receiving these messages. The functions for sending and receiving are specified only by signature, thus having no runtime implementation. However the intention is that these functions should handle mapping protocol elements to the network. My experiment showns that this is indeed possible.

4.5.2 Specification of cryptographic elements

```
module Crypto

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

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

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

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}

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 4.6: Formalization of crypto elements.

The *crypto* module is responsible for providing the cryptographic functions necessary to sign, verify, encrypt and decrypt messages. The *crypto* module utilized the refinement type to ensure that signed messages have typed dependency to the signing principal. It does not have a concrete implementation, but tests have successful confirmed it possible.

4.5.3 Specification of certificate store module

```
module CertStore

open Crypto

val GetPublicKey: p:prin -> pubkey p

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

Listing 4.7: Abstract certificate store.

The *CertStore* module provides two abstract functions for retrieving certificates from a certificate store. The two functions uses the value-dependent syntax for relating the principal to the certificate keys. Note that this in the case of the private key is a somewhat naive specification since it basically only requires the name of the principal to obtain the private key. For this to be more robust the principal type could be extended with a credential type.

4.5.4 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
  | Login (url) -> (*Create AuthnRequest and send back*)
    let authnReq = CreateAuthnRequestMessage me idp in
    assume(Log me authnReq) (*Protocol event*);
    let myprivk = CertStore.GetPrivateKey me in
    let sigSP = Sign me myprivk authnReq in
    let resp = AuthnRequestMessage me idp authnReq sigSP in
    SendSaml client resp; (*2*)
    serviceprovider me client idp (*Start over*)

  | 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) (*12*)
      else SendSaml client (DisplayError 403);(*12.1*)
      serviceprovider me client idp (*Start over*)

  | _ -> SendSaml client (DisplayError 400);(*2.1*)
        serviceprovider me client idp (*Start over*)
```

Listing 4.8: Formalization of service provider.

The service provider implements the algorithms 1 and 7 in section 3.3. The module is constructed to accept messages of type *Login* and *AuthnResponseMessage*. Any other type of message will make the service provider return a HTTP error. Compared to the algorithms 1 and 7, the service provider does not directly implement encoding and decoding, since this is expected to be handled by the *SamlProtocol* module.

4.5.5 Specification of the Identity Provider

```
module Identityprovider

open SamlProtocol
open Crypto

let handleUserAudhenticated 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);
        handleUserAudhenticated 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 4.9: Formalization of identity provider.

The identity provider implements the algorithms 2 and 6. As the service provider, the identity provider implements the two algorithms as the two match branches, *AuthnRequestMessage* and *UserAuthenticated*. The *AuthnRequestMessage* branch verifies the *AuthnRequest* and creates a challenge, a nonce, to be used for relating the login at the IdP to the authentication at DanId. But the challenge is also used for ensuring that the authentication is not reused i.e., a challenge can only be used once. This part is not yet enforced by the IdP, but I have discussed with the F* team and a prototype is under way, but did not make it in time for this report. The *UserAuthenticated* branch of the IdP module, verifies the authentication data received from the DanId applet. If accepted an assertion is issued. I have tested this verification using the OOAPI. The function for issuing the assertion is abstract i.e., not executable, but my experiments have shown it possible to use the OIOSAML.NET[7] to generate assertions for F* programs. The assertion is then signed and encrypted before it is sent to the service provider through the users browser.

4.6 Introducing adversaries

In the formalization described in F* in the previous section I have been focused on implementing the protocol according to specification (or as part of). Things however get a lot more interesting when introducing an adversary in the protocol verification. In this project I did not, within the duration of the project, manage to incorporate dedicated adversaries into the protocol verification. On the F* tutorial website, the example authentication protocol an adversary is introduced through an abstract program and a main function to execute a protocol run. An adoption of this way of introducing an adversary is applied the Nemlog-in protocol as showed in listing 4.10. The abstract *attacker* function is a parameter to the *main* function. It is able to use any function provided by the modules, but will not be able to call any assume command, thus every assertion will in the Sp and IdP succeed.

```
module Main

open SamlProtocol
open Crypto
open Serviceprovider
open Identityprovider

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

let main attacker =
  Fork [  attacker;
    (fun () -> serviceprovider "serviceprovider.org" "browser"
         "identityprovider.org");
    (fun () -> identityprovider "identityprovider.org" "
        browser")]
```

Listing 4.10: Main module for introducing adversaries.

Given that attack on a protocol like this most often is done through the users browser, a more sophisticated introduction of adversaries would be to model the

[7]Open source implementation of the OIOSAML protocol in .NET

browser as part of the protocol model. If an attacker has gained control of a users browser, security measures like encrypting network traffic over HTTPS also becomes less valuable. Modelling the browser specifically will enable to model and mitigate known attacks on the protocol like *Man In the Middle* and it's real-time variant, authentication replay and session hijacking.

4.7 State of the implementation

The implementation of the Nemlog-in protocol as described in the previous section only covers a core subset of the entire protocol. The identity and service providers have abstract implementation, but are both missing session handling. I have not incorporated the DanId and the applet parts yet and these would, thus be obvious modules to pick for continuing my work. As described in section 4.6, modelling of adversaries is not part of the present implementation either.

All in all about 20% of the protocol can be considered to be implemented, since the most of the implementation is abstract signatures only this might be on the high side. But the crypto and networking experiments (attached in appendix B), will make up for the seemingly missing implementation, the time-frame has just not allowed me to incorporate it into the model itself.

Chapter 5

Evaluation

In hhis chapter i will is focused on describing what have been achieved during the project. What can be derived from the work done and where research into this area would make the most sense. It will also discuss what related work have been done and how that can contribute to further research.

5.1 Project evaluation

In this report I have outlined the core work of this project, a prototype specification of the Nemlog-in protocol including a service provider and identity provider in written in F*.

Working with a language and compiler like F* can some time be quite challenging, even though the project is well documented compared to many other research projects. Though the F*-project has an interactive compiler on the project website with demo code and explanation, the compiler can still be rather hard to interpret. Here it should be mentioned that I did not have any experience with ML or any other functional oriented language, but a strong background in C# and .NET. Being a research language claiming to be a dialect of ML or ML-like, it can be somewhat difficult to figure out which ML constructs are implemented, which constructs are implemented as SML and which like OCaml. Having the compiler source code available, in the end made things possible. I will also note that some of the publications claimed that F* provided access for libraries for cryptography and networking. This did not hold true for the 2 versions of the compiler I used, where this functionality seemed to have been commented out of the compiler source code. This meant that I had to conduct some experiments in order to interact with the network and to sign messages. This is of course a result of F* being a research project, thus serving as vehicle primarily for their own research, thus not (for good reasons) being able to maintain all aspects of the language and compiler.

Evaluating the project with a general software development perspective in mind, the model presented here can be evaluated against common software development and software architecture principles. The interesting part here is not so much how the modules have been designed as much as what kind of style a language like F* imposes on the developer. F* enforces the principle of programming to an interface immortalized by Gamma et al.[9]. It even takes it further to *design by contract* known from other languages like Eiffel. Functional languages like ML where immutability is the default makes it easier for the developer to design components which holds no state, which in turn makes scaling system across multiple processors or even computers a lot easier.

As a final remark I will point out the fact that architecting secure software and security systems in general, should not focus solely on solving security issues with technology. As described by Schneier in *Liars and Outliers*[15], security in the end boils down to trust and trust relations. Users should be able to trust the systems they use and they should be enabled to do so.

5.2 Related work

In *Using static analysis to validate the SAML single sign-on protocol* by Hansen and Skriver in [10] and [16] the SAML login protocol is thoroughly analyzed and formalized. Their analysis goes towards the older version 1.1 protocol, but their recommendation of mandating HTTPS network transport has been incorporated into the OIOSAML specification. Armando et al.[1], have also analyzed the SAML protocol in connection with Google Apps. This analysis pointed out errors in the actual implementation of the protocol, thus further justifying the approach of this project to implement by formal specification. In *Modelling session types using contracts*[2] by Bernardi and Hennessy, the authors explore the use of contracts to model session types. Though a rather different and more complicated approach to protocol modelling, session types might prove interesting for future Nemlog-in researchers .

5.3 Threats to validity

In a project like this there will be lots of threats to validity. Given that the project navigates in the area of security, scrutinizing the work of this report is extremely important.

First of all it should be noted that I have not worked with neither formal specification nor functional programming before commencing this project. Thus there is a fair chance that the full power and capabilities of F* has not been utilized to its fullest potential.

Further the F* compiler, though formalized on Coq and having verified more than 20.000 lines of code[18] is still to be considered an alpha-release. This means that results still needs to be treated with some degree of vigilance. However the concepts on which F* builds have been throughly explored in previous project like F7 and Fine[7].

In this report I have touched very briefly on the subject of modelling of adversaries. This might leave room for some degree of uncertainty, and therefore this would be a very natural path to pursue when continuing the research.

Another point to mention is that the parts of the protocol description related to authentication with DanId, is the result of some degree of "black-box" analysis. Thus it should be clarified that this might not be 100 % true to the actual Nemlog-in protocol, but it will suffice for building a formal specification of the authentication that *would* work in the Nemlog-in protocol.

5.4 Further research

As the prototype implementation presented in this report only covers a subset of the entire protocol it would of course make sense to finish the work and implement the remainder of the protocol. Making a more robust framework for protocol verification and implementation would require a structured modelling of adversaries. In section 4.6, I discuss how adversaries can be modeled as part of modelling the browser. Research by Swamy et al.[19] and Fournet et al.[8] suggests the feasibility of using F* for compilation to JavaScript. In the context of Nemlog-in this will be interesting for at least two reasons. First of all this would make it possible to build an executable specification for the applet part of the protocol and secondly present a solution to an inherent problem with todays NemId. The fact that authentication with NemId is done through a Java applet, forces all users of NemId to install a Java browser plug-in. Given the many security issues reported lately regarding the Java browser plug-in, relying on JavaScript instead will thereby eliminate a large security risk for the users.

5.5 Conclusion

In a world where digital identity is getting increasingly important, secure systems that protect these identities are even more important. Securing such systems not only depend on secure protocol. Research into security protocol suggest that it is often the case that the implementation of a seemingly secure protocol introduces vulnerabilities. In this report I show how the use of a special programming language, F* for implementing security protocols, can help proving the security properties of the protocol.

In this project I have used Danish governmental Single Sign-On federation - Nemlog-in as an example. Nemlog-in is a protocol based on the SAML standard for exchanging security tokens, but the authentication part is done by another governmental initiative NemId (Easy Id), using a non-public protocol. The Nemlog-in protocol have been thoroughly investigated as a whole and specified to the extend possible, given that fact that the authentication part of the protocol is not publicly available. Specification of the authentication parts have been done by studying public information, implementation guides and analysis of network traffic.

In this report I have formalized key parts of the Nemlog-in protocol using the F* programming language, and used it to secure key security properties like signing of the messages exchanged over the network.

Since only key security properties have been formalized, a way forward for securing other properties have been described. For further securing the protocol I have suggested to use F* to model know attacks as part of the protocol test suite.

To finish the implementation of the protocol I have suggested to utilize the work done by Swamy et al.[19] to formalize the browser aided part of the protocol in a dialectal of F* that compiles to JavaScript.

To conclude it has proven feasible that a programming language like F* can be used to create an executable model of a security protocol thereby ensuring security features mandated by the protocol.

Bibliography

[1] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 1–10. ACM, 2008.

[2] Giovanni Bernardi and Matthew Hennessy. Modelling session types using contracts. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1941–1946, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2231936.2232097. URL `http://doi.acm.org/10.1145/2231936.2232097`.

[3] DanId. Nemid tjenesteudbyder support, 2013. URL `https://www.nets-danid.dk/produkter/for_tjenesteudbydere/nemid_tjenesteudbyder/nemid_tjenesteudbyder_support/`.

[4] Digitaliser.dk. Oio web sso profile v2.0.9, 2012. URL `http://digitaliser.dk/resource/2377872/artefact/OIO+Web+SSO+Profile+V2+09.pdf`.

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

[6] Cédric Fournet, Karthikeyan Bhargavan, and Andrew D. Gordon. Cryptographic verification by typing for a sample protocol implementation. In *FOSAD*, pages 66–100, 2011.

[7] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *ACM Conference on Computer and Communications Security*, pages 341–350, 2011.

[8] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Ben Livshits. Fully abstract compilation to javascript. In *In Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (To appear)*, 2013.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[10] Steffen M. Hansen, Jakob Skriver, and Hanne Riis Nielson. Using static analysis to validate the saml single sign-on protocol. In *Proceedings of*

the 2005 workshop on Issues in the theory of security, WITS '05, pages 27–40, New York, NY, USA, 2005. ACM. ISBN 1-58113-980-2. doi: 10.1145/1045405.1045409. URL `http://doi.acm.org/10.1145/1045405.1045409`.

[11] Nets-DanId. Nemid guide, 2013. URL `https://www.nets-danid.dk/produkter/for_tjenesteudbydere/nemid_tjenesteudbyder/`.

[12] OASIS-Open. Saml 2.0-core, 2005. URL `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`.

[13] Microsoft Research. F* - guide, 2012. URL `http://rise4fun.com/FStar/tutorial/guide`.

[14] Microsoft Research. F*: A verifying ml compiler for distributed programming, 2012. URL `http://research.microsoft.com/en-us/projects/fstar/`.

[15] B. Schneier. *Liars and Outliers: Enabling the Trust that Society Needs to Thrive*. Wiley, 2012. ISBN 9781118239018. URL `http://books.google.dk/books?id=lPsbhIUexo0C`.

[16] J. Skriver and S. M. Hansen. Using static analysis to validate SAML protocols. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2004. URL `http://www2.imm.dtu.dk/pubdb/p.php?3396`. Supervised by Prof. Hanne Riis Nielson.

[17] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking saml: be whoever you want to be. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362814`.

[18] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *SIGPLAN Not.*, 46(9):266–278, September 2011. ISSN 0362-1340. doi: 10.1145/2034574.2034811. URL `http://doi.acm.org/10.1145/2034574.2034811`.

[19] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Ben Livshits. Towards javascript verification with the dijkstra state monad. Technical report, MSR, 2012.

Appendix A

SAML messages

Examples of SAML messages and assertion.

```xml
<samlp:AuthnRequest ID="id84be1b160a0b44d893402f993f080679"
  Version="2.0" IssueInstant="2012-10-23T18:36:14.5304032Z"
  Destination="https://login.idp.dk"
  IsPassive="false" xmlns:samlp="urn:oasis:names:tc:SAML:2.0
      :protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https:
      //saml.serviceprovider.dk</Issuer>
  <Conditions xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
    <AudienceRestriction>
      <Audience>https://saml.serviceprovider.dk</Audience>
    </AudienceRestriction>
  </Conditions>
</samlp:AuthnRequest>
```

Listing A.1: Sample SAML AuthnRequest message.

```xml
<samlp:Response ID="_e0ef82f8-f8d9-4bcc-a4da-894a165a04c9"
  Version="2.0" IssueInstant="2012-10-23T18:36:37.997Z"
  Destination="https://login.serviceprovider.dk"
  Consent="urn:oasis:names:tc:SAML:2.0:consent:unspecified"
  InResponseTo="id84be1b160a0b44d893402f993f080679"
      xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https:
      //saml.idp.dk</Issuer>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0
        :status:Success" />
  </samlp:Status>
  <Assertion ID="_b3e42065-401b-4fe6-954b-2f485828e68b"
      IssueInstant="2012-10-23T18:36:37.997Z" Version="2.0"
      xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
  ...
  </Assertion>
</samlp:Response>
```

Listing A.2: Sample SAML Response message without the content of the Assertion.

```
<Assertion ID="_b3e42065-401b-4fe6-954b-2f485828e68b" IssueInstant="2012-10-23T18:36:37.997Z" Version="2.0" xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
    <Issuer>https://saml.idp.dk</Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      ...
    </ds:Signature>
    <Subject>
      <NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">The Subject</NameID>
      <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
        <SubjectConfirmationData InResponseTo="id84be1b160a0b44d893402f993f080679" NotOnOrAfter="2012-10-23T18:41:37.997Z" Recipient="https://saml.serviceprovider.dk" />
      </SubjectConfirmation>
    </Subject>
    <Conditions NotBefore="2012-10-23T18:36:37.997Z" NotOnOrAfter="2012-10-23T19:36:37.997Z">
      <AudienceRestriction>
        <Audience>https://saml.serviceprovider.dk</Audience>
      </AudienceRestriction>
    </Conditions>
    <AttributeStatement>
      <Attribute Name="urn:oid:2.5.4.3" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
        <AttributeValue>Frode Bo</AttributeValue>
      </Attribute>
      <Attribute Name="urn:oid:0.9.2342.19200300.100.1.3" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
        <AttributeValue>fbh@mail.dk</AttributeValue>
      </Attribute>
    </AttributeStatement>
    <AuthnStatement AuthnInstant="2012-10-23T18:36:36.622Z" SessionIndex="_b3e42065-401b-4fe6-954b-2f485828e68b">
      <AuthnContext>
        <AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:X509</AuthnContextClassRef>
      </AuthnContext>
    </AuthnStatement>
  </Assertion>
```

Listing A.3: Sample SAML Assertion.

Appendix B

Source code examples

This chapter displays some of the experiments done in order to inter-operate with the .NET framework.

B.1 Interop declarations

This example shows to import .NET into the F* environment

```
module Interop

open Protocol

extern reference Crypto {language="F#";
            dll="RuntimeExtention";
            namespace="";
            classname="Crypto"}

extern Crypto val KeyGenExt: p:prin
          -> (pubkey p * privkey p)

extern reference Saml {language="F#";
            dll="RuntimeExtention";
            namespace="";
            classname="Saml"}

extern Saml val CreateAuthnRequest: issuer:prin -> destination
    :prin -> samlmessage
extern Saml val CreateChallenge: prin -> nonce
extern Saml val CreateSamlAssertion: user:string -> issuer:
    prin -> destination:prin -> assertion
extern Saml val CreateSamlResponse: issuer:prin -> destination
    :prin -> assertion -> samlmessage
extern Saml val CreateSamlFailedResponse: issuer:prin ->
    destination:prin -> SamlStatus -> samlmessage

extern reference Network {language="F#";
            dll="RuntimeExtention";
            namespace="";
            classname="Network"}
extern Network val SendX: prin -> message -> bool
extern Network val RecieveX: prin -> message
```

Listing B.1: Example of how to interop with .NET

B.2 Using .NET functions

A simple example of using the imported functions in F*

```
module X

open Protocol
open Interop

val GenKeys: prin -> string
let GenKeys principal =
  println "starting keygen";
  let (privk, pubk) = KeyGenExt principal in

  print_int 2;
  "super"

val GetReq: prin -> prin -> samlmessage
let GetReq sp idp =
  CreateAuthnRequest sp idp


val GetSamlMessages: prin -> prin -> string
let GetSamlMessages sp idp =
  print_string "Getting AuthReq\n";
  let req = CreateAuthnRequest sp idp in
  print_string req;
  (*print_string "\nGetting Challenge\n";
  let challenge = CreateChallenge sp in
  print_string challenge;*)
  print_string "\nGetting Assertion\n";
  let assertion = CreateSamlAssertion "me" idp sp in
  print_string assertion;
  print_string "\nGetting Response\n";
  let response = CreateSamlResponse idp sp assertion in
  print_string response;
  "Done"

val CreateAndSend: prin -> prin ->unit
let CreateAndSend sp idp =
  let authReq = CreateAuthnRequest sp idp in
  let req = SamlProtocolMessage sp authReq "sig" in
  let status = SendX sp req in
  match status with
  | true -> print_string "great!";()
  | _ -> print_string "crap!";()

val CreateSendAndRecieve: prin -> prin ->unit
let CreateSendAndRecieve sp idp =
  let authReq = CreateAuthnRequest sp idp in
  let req = SamlProtocolMessage sp authReq "sig" in
  let status = SendX sp req in
  match status with
  | true -> print_string "great!";
    let msg = RecieveX sp in
    match msg with
    | HttpGet (url) ->
      print_string url;()
  | _ -> print_string "crap!";()
```

Listing B.2: Example of use types from .NET

B.3 Crypto wrapper implemented in F#

```
module Crypto

open System.Security.Cryptography
open System.Text
open System
open System.IO

let GetEncoder = new ASCIIEncoding()

let GetHashProvicer = new SHA1CryptoServiceProvider()

let GetCryptoProvider = new RSACryptoServiceProvider()

let WriteKeyToFile (filename:string) (key:string) =
    // save to file
    let fn = filename.Replace("://", "-")
    let file = File.CreateText(fn)
    file.Write key |>  file.Close

let KeyGenExt prin : Prims.DepTuple<Protocol.pubkey,Protocol.
    privkey> =
    let cspParams = new CspParameters()
    cspParams.ProviderType <- 1
    cspParams.Flags <- CspProviderFlags.UseArchivableKey
    cspParams.KeyNumber <- int KeyNumber.Exchange
    let rsaProvicer = GetCryptoProvider

    // Export Public Key
    let pubkey = Protocol.pubkey(0,rsaProvicer.ToXmlString
        false)

    // Export private key
    let privkey = Protocol.privkey(0,rsaProvicer.ToXmlString
        true)

    new Prims.DconDepTuple<Protocol.pubkey,Protocol.privkey>(
        pubkey, privkey) :> Prims.DepTuple<Protocol.pubkey,
        Protocol.privkey>

let SignMessage (prin:string) (key:string) (msg:string) :
    string =
    let ByteConverter = GetEncoder
    let msgData = ByteConverter.GetBytes(msg)
    let alg = GetCryptoProvider
    alg.FromXmlString key
    let signedData = alg.SignData(msgData, GetHashProvicer)

    Convert.ToBase64String(signedData)

let ValidateMessage (prin:string) (key:string) (msg:string) (
    signature:string) : bool =
    let ByteConverter = GetEncoder
    let dataToVerify = ByteConverter.GetBytes msg
    let sign = Convert.FromBase64String signature
    let alg = GetCryptoProvider
    alg.FromXmlString key

    alg.VerifyData(dataToVerify, GetHashProvicer, sign)
```

Listing B.3: Example of implementation of crypto functions in F#

B.4 SAML wrapper implemented in F#

```
module Saml

open System
open System.Security.Cryptography

let CreateChallenge prin =
    let provider = new RNGCryptoServiceProvider()
    let randBytes = Array.create 10 (byte 0)
    let _ = provider.GetNonZeroBytes randBytes
    BitConverter.ToString (randBytes)

let CreateAuthnRequest sp idp =
    let authnReq = new dk.nita.saml20.Saml20AuthnRequest()
    authnReq.Destination <- idp
    authnReq.Issuer <- sp
    let doc = authnReq.GetXml()
    doc.InnerXml

let CreateSamlAssertion user issuer destination =
    let assertion = new dk.nita.saml20.Schema.Core.Assertion()
    assertion.Issuer <- issuer
    "assertion"

let CreateSamlResponse user issuer destination status =
    let response = new dk.nita.saml20.Schema.Protocol.Response
        ()
    response.Issuer <- issuer
    response.IssueInstant <- Nullable(DateTime.UtcNow)
    response.Destination <- destination
    response
```

Listing B.4: Example of implementation of SAML functions in F#

B.5 Network wrapper implemented in F#

```fsharp
module Network

open System
open System.Net
open System.Text

let listener = new HttpListener()

let GetMsg (msg:Protocol.message) =
    match msg with
    | :? Protocol.SamlProtocolMessage -> (msg :?> Protocol.
        SamlProtocolMessage).field_2
    | _ -> "<html><head><title>Server</title></head><body><h1>
        Message from: %s not valid</h1></body></html>"

let StartListen  =
    listener.Prefixes.Add("http://localhost:8080/")
    listener.Start()
    printfn "listening...."

let GetContext = listener.GetContext()

let SendX (prin:string) (msg:Protocol.message) =
    let resp = GetContext.Response
    let cookie = new Cookie("AuthnTicket", "12 02 002 020 22")
    resp.AppendCookie(cookie)
    let respmsg = GetMsg msg
    let buffer = System.Text.Encoding.UTF8.GetBytes(respmsg)
    resp.ContentLength64 <- int64 buffer.Length
    let output = resp.OutputStream;
    output.Write(buffer,0,buffer.Length)
    output.Close();
    true

let RecieveX (prin:string) : (Protocol.message) =
    let req = GetContext.Request
    match prin with
    | "sp" ->  Protocol.HttpGet ("Request to sp: " + req.
        RawUrl) :> Protocol.message
    | "idp" -> Protocol.HttpGet "This is a idp reply" :>
        Protocol.message
    | "browser" -> Protocol.HttpGet "This is a browserreply"
        :> Protocol.message
    | _ -> Protocol.Failed 400 :> Protocol.message
```

Listing B.5: Example of implementation of network functions in F#

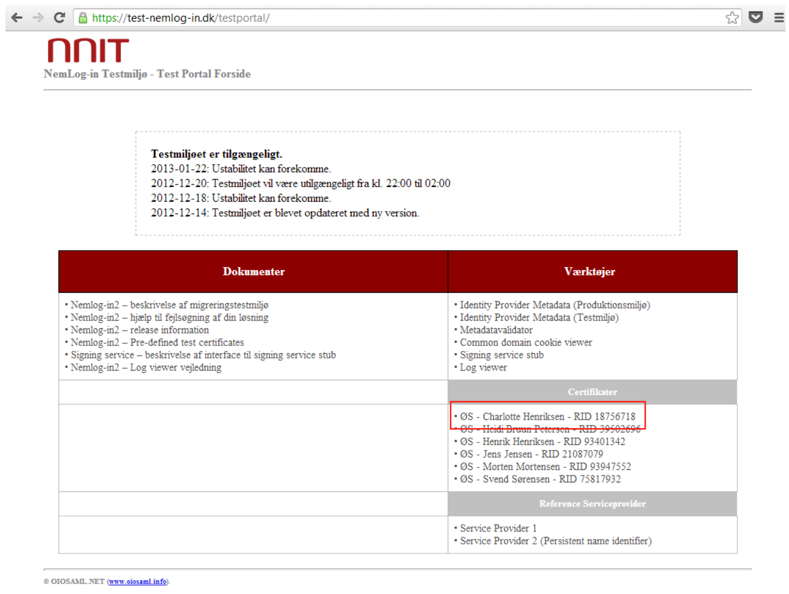
Appendix C

Detailed fiddler trace

This appendix will describe how to do a trace of a login using Fiddler 2 and the Nemlog-in test environment provided for Nemlog-in. The trace described has been recorded in Fiddler 2 and is available from the attached CD. First section will describe how to login through the test portal. The second section will describe how to use understand the trace recorded in Fiddler 2.

C.1 Performing a login

The Nemlog-in test portal can be found at `https://test-nemlog-in.dk/testportal/`. In order actually login it is necessary to obtain a test identity. This can be done from the test portal. Download the certificate from the link in the red rectangle.


https://test-nemlog-in.dk/testportal/

NemLog-in Testmiljø - Test Portal Forside

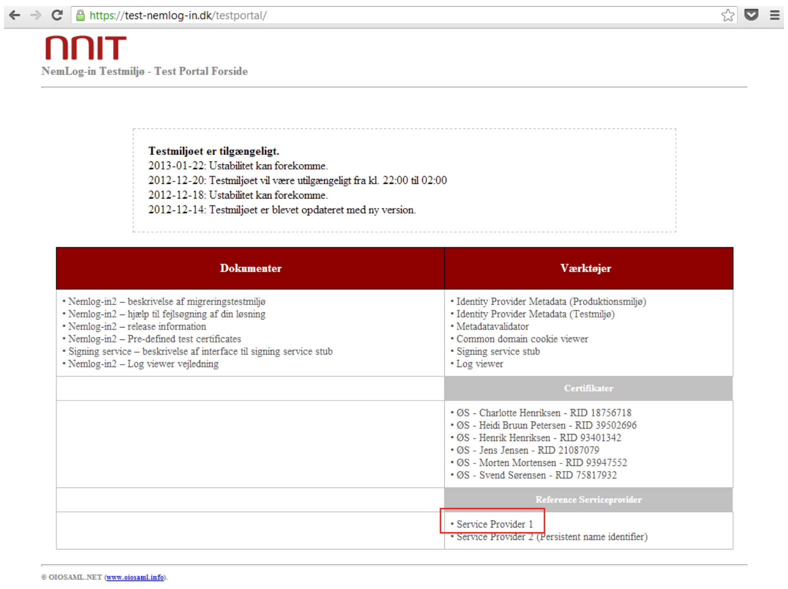
Testmiljøet er tilgængeligt.
2013-01-22: Ustabilitet kan forekomme.
2012-12-20: Testmiljøet vil være utilgængeligt fra kl. 22:00 til 02:00
2012-12-18: Ustabilitet kan forekomme.
2012-12-14: Testmiljøet er blevet opdateret med ny version.

Dokumenter	Værktøjer
• Nemlog-in2 – beskrivelse af migreringstestmiljø • Nemlog-in2 – hjælp til fejlsøgning af din løsning • Nemlog-in2 – release information • Nemlog-in2 – Pre-defined test certificates • Signing service – beskrivelse af interface til signing service stub • Nemlog-in2 – Log viewer vejledning	• Identity Provider Metadata (Produktionsmiljø) • Identity Provider Metadata (Testmiljø) • Metadatavalidator • Common domain cookie viewer • Signing service stub • Log viewer
	Certifikater
	• ØS - Charlotte Henriksen - RID 18756718 • ~~ØS - Heidi Bruun Petersen - RID 39502696~~ • ØS - Henrik Henriksen - RID 93401342 • ØS - Jens Jensen - RID 21087079 • ØS - Morten Mortensen - RID 93947552 • ØS - Svend Sørensen - RID 75817932
	Reference Serviceprovider
	• Service Provider 1 • Service Provider 2 (Persistent name identifier)

© OIOSAML.NET (www.oiosaml.info).

Figure C.1: Nemlog-in test portal - Download test certificate

Click on the test service provider (the link in the red rectangle) to be able to perform a login.



https://test-nemlog-in.dk/testportal/

NemLog-in Testmiljø - Test Portal Forside

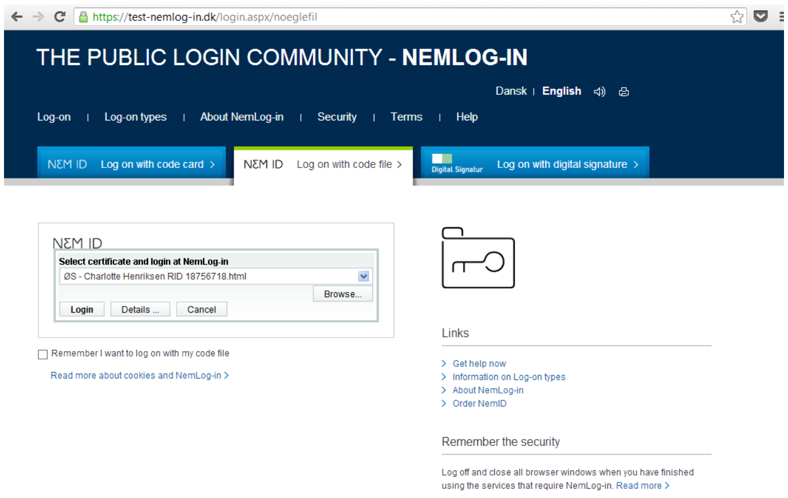
Testmiljøet er tilgængeligt.
2013-01-22: Ustabilitet kan forekomme.
2012-12-20: Testmiljøet vil være utilgængeligt fra kl. 22:00 til 02:00
2012-12-18: Ustabilitet kan forekomme.
2012-12-14: Testmiljøet er blevet opdateret med ny version.

Dokumenter	Værktøjer
• Nemlog-in2 – beskrivelse af migreringstestmiljø • Nemlog-in2 – hjælp til fejlsøgning af din løsning • Nemlog-in2 – release information • Nemlog-in2 – Pre-defined test certificates • Signing service – beskrivelse af interface til signing service stub • Nemlog-in2 – Log viewer vejledning	• Identity Provider Metadata (Produktionsmiljø) • Identity Provider Metadata (Testmiljø) • Metadatavalidator • Common domain cookie viewer • Signing service stub • Log viewer
	Certifikater
	• ØS - Charlotte Henriksen - RID 18756718 • ØS - Heidi Bruun Petersen - RID 39502696 • ØS - Henrik Henriksen - RID 93401342 • ØS - Jens Jensen - RID 21087079 • ØS - Morten Mortensen - RID 93947552 • ØS - Svend Sørensen - RID 75817932
	Reference Serviceprovider
	• Service Provider 1 • Service Provider 2 (Persistent name identifier)

© OIOSAML.NET (www.oiosaml.info).

Figure C.2: Nemlog-in test portal - Go to test service provider

Choose login with code file and select the downloaded test certificate. Password for the certificate is "Test1234".



https://test-nemlog-in.dk/login.aspx/noeglefil

THE PUBLIC LOGIN COMMUNITY - **NEMLOG-IN**

Dansk | **English**

Log-on | Log-on types | About NemLog-in | Security | Terms | Help

NEM ID Log on with code card > | NEM ID Log on with code file > | Digital Signatur Log on with digital signature >

NEM ID
Select certificate and login at NemLog-in
ØS - Charlotte Henriksen RID 18756718.html
Browse...
Login | Details ... | Cancel

☐ Remember I want to log on with my code file

Read more about cookies and NemLog-in >

Links
- Get help now
- Information on Log-on types
- About NemLog-in
- Order NemID

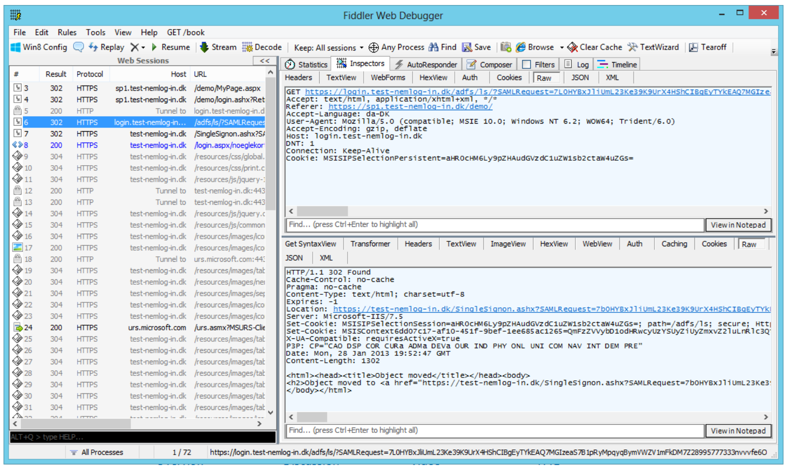
Remember the security

Log off and close all browser windows when you have finished using the services that require NemLog-in. Read more >

Figure C.3: Nemlog-in test portal - login page

C.2 Fiddler trace

The login sequence described previously is recorded with Fiddler. The figure below show an overview of the recorded trace. To load the trace from CD, choose File-¿Load Archive.


Fiddler Web Debugger
File Edit Rules Tools View Help GET /book
Win8 Config Replay Resume Stream Decode Keep: All sessions Any Process Find Save Browse Clear Cache TextWizard Tearoff
Web Sessions
Statistics Inspectors AutoResponder Composer Filters Log Timeline
Headers TextView WebForms HexView Auth Cookies Raw JSON XML
Get SyntaxView Transformer Headers TextView ImageView HexView WebView Auth Caching Cookies Raw JSON XML
Find... (press Ctrl+Enter to highlight all)
View in Notepad
ALT+Q > type HELP...
All Processes

#	Result	Protocol	Host	URL
3	302	HTTPS	sp1.test-nemlog-in.dk	/demo/MyPage.aspx
4	302	HTTPS	sp1.test-nemlog-in.dk	/demo/login.ashx?Ret
5	200	HTTP	Tunnel to	login.test-nemlog-in.d
6	302	HTTPS	login.test-nemlog-in...	/adfs/ls/?SAMLReques
7	302	HTTPS	test-nemlog-in.dk	/SingleSignon.ashx?SA
8	200	HTTPS	test-nemlog-in.dk	/login.aspx/noeglekor
9	304	HTTPS	test-nemlog-in.dk	/resources/css/global.
10	304	HTTPS	test-nemlog-in.dk	/resources/css/print.c
11	304	HTTPS	test-nemlog-in.dk	/resources/js/jquery-1
12	200	HTTP	Tunnel to	test-nemlog-in.dk:443
13	200	HTTP	Tunnel to	test-nemlog-in.dk:443
14	304	HTTPS	test-nemlog-in.dk	/resources/js/jquery.c
15	304	HTTPS	test-nemlog-in.dk	/resources/js/common
16	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
17	200	HTTPS	test-nemlog-in.dk	/resources/images/ico
18	200	HTTP	Tunnel to	urs.microsoft.com:443
19	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
20	304	HTTPS	test-nemlog-in.dk	/resources/images/ne
21	304	HTTPS	test-nemlog-in.dk	/resources/images/sep
22	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
23	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
24	200	HTTPS	urs.microsoft.com	/urs.asmx?MSURS-Clie
25	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
26	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
27	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
28	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
29	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
30	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
31	304	HTTPS	test-nemlog-in.dk	/resources/images/tab

```
GET https://login.test-nemlog-in.dk/adfs/ls/?SAMLRequest=7L0HYBxJliUmL23Ke39K9UrX4HShCIBgEyTYkEAQ7MGIze
Accept: text/html, application/xhtml+xml, */*
Referer: https://sp1.test-nemlog-in.dk/demo/
Accept-Language: da-DK
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Accept-Encoding: gzip, deflate
Host: login.test-nemlog-in.dk
DNT: 1
Connection: Keep-Alive
Cookie: MSISIPSelectionPersistent=aHR0cHM6Ly9pZHAudGVzdC1uZW1sb2ctaW4uZGs=
```

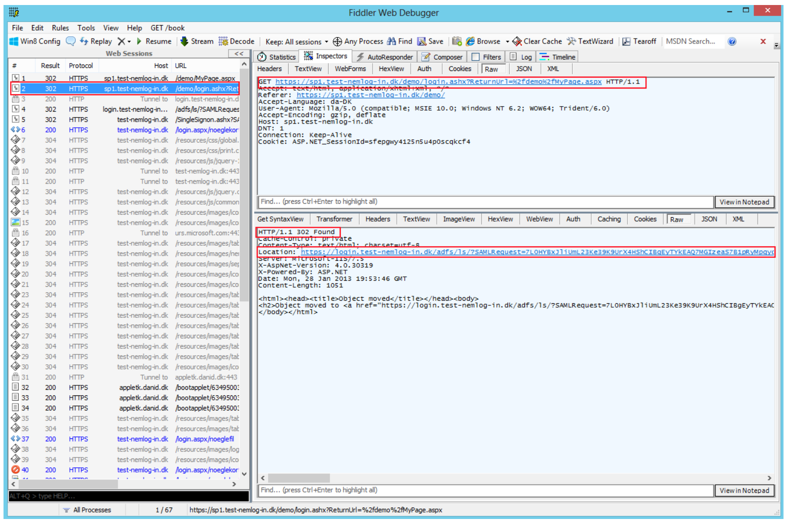
```
HTTP/1.1 302 Found
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Location: https://test-nemlog-in.dk/SingleSignon.ashx?SAMLRequest=7b0HYBxJliUmL23Ke39K9UrX4HShCIBgEyTYk
Server: Microsoft-IIS/7.5
Set-Cookie: MSISIPSelectionSession=aHR0cHM6Ly9pZHAudGVzdC1uZW1sb2ctaW4uZGs=; path=/adfs/ls; secure; Htt
Set-Cookie: MSISContext6dd07c17-af10-451f-9bef-1ee685ac1265=QmFzZVVybD1odHRwcyUzYSUyZiUyZmxvZ2luLnRlc3Q
X-UA-Compatible: requiresActiveX=true
P3P: CP="CAO DSP COR CURa ADMa DEVa OUR IND PHY ONL UNI COM NAV INT DEM PRE"
Date: Mon, 28 Jan 2013 19:52:47 GMT
Content-Length: 1302

<html><head><title>Object moved</title></head><body>
<h2>Object moved to <a href="https://test-nemlog-in.dk/SingleSignon.ashx?SAMLRequest=7b0HYBxJliUmL23Ke3
</body></html>
```

1 / 72 https://login.test-nemlog-in.dk/adfs/ls/?SAMLRequest=7L0HYBxJliUmL23Ke39K9UrX4HShCIBgEyTYkEAQ7MGIzeaS7B1pRyMpqyqBymVWZV1mFkDM7Z28995777333nvvvfe6O

Figure C.4: Overview fiddler trace

In order to show the details, click in one of the item in the list to the left. In the figure below, the request for login on the service provider has been selected. On the right can be seen the request and response messages respectively. As can be seen the response is a redirect (status code 302) and the destination for the redirect is Nemlog-in. It can also be seen that there is a query parameter called *SAMLRequest*.


Fiddler Web Debugger
File Edit Rules Tools View Help GET /book
Win8 Config Replay Resume Stream Decode Keep: All sessions Any Process Find Save Browse Clear Cache TextWizard Tearoff MSDN Search...
Web Sessions
Statistics Inspectors AutoResponder Composer Filters Log Timeline
Headers TextView WebForms HexView Auth Cookies Raw JSON XML
Get SyntaxView Transformer Headers TextView ImageView HexView WebView Auth Caching Cookies Raw JSON XML
Find... (press Ctrl+Enter to highlight all)
View in Notepad
ALT+Q > type HELP...
All Processes

#	Result	Protocol	Host	URL
1	302	HTTPS	sp1.test-nemlog-in.dk	/demo/MyPage.aspx
2	302	HTTPS	sp1.test-nemlog-in.dk	/demo/login.ashx?Ret
3	200	HTTP	Tunnel to	login.test-nemlog-in.d
4	302	HTTPS	login.test-nemlog-in...	/adfs/ls/?SAMLReques
5	302	HTTPS	test-nemlog-in.dk	/SingleSignon.ashx?SA
6	200	HTTPS	test-nemlog-in.dk	/login.aspx/noeglekor
7	304	HTTPS	test-nemlog-in.dk	/resources/css/global.
8	304	HTTPS	test-nemlog-in.dk	/resources/css/print.c
9	304	HTTPS	test-nemlog-in.dk	/resources/js/jquery-1
10	200	HTTP	Tunnel to	test-nemlog-in.dk:443
11	200	HTTP	Tunnel to	test-nemlog-in.dk:443
12	304	HTTPS	test-nemlog-in.dk	/resources/js/jquery.c
13	304	HTTPS	test-nemlog-in.dk	/resources/js/common
14	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
15	200	HTTPS	test-nemlog-in.dk	/resources/images/ico
16	200	HTTP	Tunnel to	urs.microsoft.com:443
17	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
18	304	HTTPS	test-nemlog-in.dk	/resources/images/ne
19	304	HTTPS	test-nemlog-in.dk	/resources/images/sep
20	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
21	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
23	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
24	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
25	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
26	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
27	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
28	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
29	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
30	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
31	200	HTTP	Tunnel to	appletk.danid.dk:443
32	200	HTTPS	appletk.danid.dk	/bootapplet/6349500
33	200	HTTPS	appletk.danid.dk	/bootapplet/6349500
34	200	HTTPS	appletk.danid.dk	/bootapplet/6349500
35	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
36	304	HTTPS	test-nemlog-in.dk	/resources/images/tab
37	200	HTTPS	test-nemlog-in.dk	/login.aspx/noeglefil
38	304	HTTPS	test-nemlog-in.dk	/resources/images/log
39	304	HTTPS	test-nemlog-in.dk	/resources/images/ico
40	200	HTTPS	test-nemlog-in.dk	/login.aspx/noeglekor

```
GET https://sp1.test-nemlog-in.dk/demo/login.ashx?ReturnUrl=%2fdemo%2fMyPage.aspx HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Referer: https://sp1.test-nemlog-in.dk/demo/
Accept-Language: da-DK
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Accept-Encoding: gzip, deflate
Host: sp1.test-nemlog-in.dk
DNT: 1
Connection: Keep-Alive
Cookie: ASP.NET_SessionId=sfepgwy4125n5u4p0scqkcf4
```

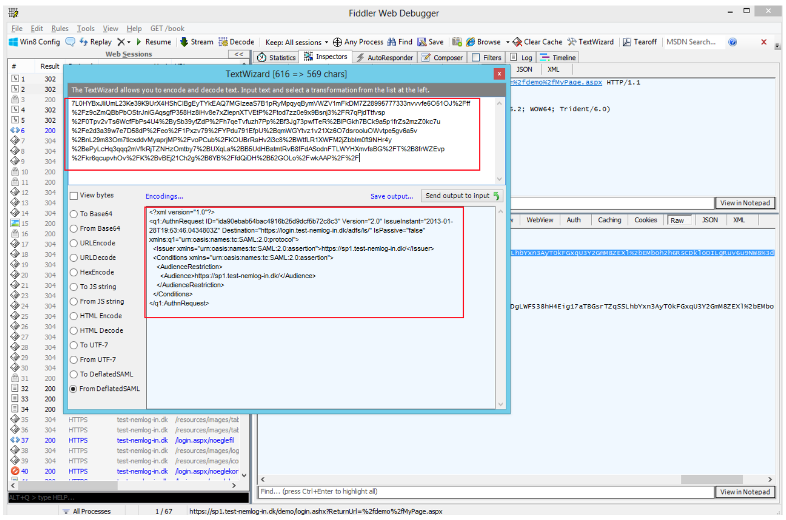
```
HTTP/1.1 302 Found
Cache-Control: private
Content-Type: text/html; charset=utf-8
Location: https://login.test-nemlog-in.dk/adfs/ls/?SAMLRequest=7L0HYBxJliUmL23Ke39K9UrX4HShCIBgEyTYkEAQ7MGIzeaS7B1pRyMpqy
Server: Microsoft-IIS/7.5
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
Date: Mon, 28 Jan 2013 19:53:46 GMT
Content-Length: 1051

<html><head><title>Object moved</title></head><body>
<h2>Object moved to <a href="https://login.test-nemlog-in.dk/adfs/ls/?SAMLRequest=7L0HYBxJliUmL23Ke39K9UrX4HShCIBgEyTYkEAC
</body></html>
```

1 / 67 https://sp1.test-nemlog-in.dk/demo/login.ashx?ReturnUrl=%2fdemo%2fMyPage.aspx

Figure C.5: Fiddler - Redirect to IdP

The *SAMLRequest* query parameter can be decoded using the Text-wizard in Fiddler. Simply copy the content of the query parameter and click the Text-wizard. Choosing DeflateFromSAML will reveal the *AuthnRequest*.


TextWizard [616 => 569 chars]

The TextWizard allows you to encode and decode text. Input text and select a transformation from the list at the left.

```
7L0HYBxJliUmL23Ke39K9UrX4HShCIBgEyTYkEAQ7MGIzeaS7B1pRyMpqyqBymVWZV1mFkDM7Z28995777333nvvvfe6O51OJ%2Fff
%2Fz9cZmQBbPbOStrJniGAqsgfP358Hz8iHv8e7xZlepnXTVEtP%2Ftod7zz0e9x9Bsnj3%2FR7qPjdTtfvsp
%2F0Tpv2vTs6WcfFbPs4U4%2BySb39yfZdP%2Fh7qeTvfuzh7Pp%2Bf3Jg73pwfTeR%2BlPGkh7BCk9a5p1frZs2mzZ0kc7u
%2Fe2d3a39w7e7D58dP%2Feo%2F1Pxzv79%2FYPdu791EfpU%2BqmWGYtvz1v21Xz6O7dsrooluOWvtpe5gv6a5v
%2BnL29m83Om7tlcxddvMyaprjMP%2FvoPCub%2FKOUBrRsHv2i3c8%2BWtfLR1XWFM2jZbblm0ft9NHr4y
%2BePyLcHq3qqq2mVfkRjTZNHzOmtby7%2BUXqLa%2BB5UdHBstmtRvB8fFdASodnFTLWYHXmvfsBG%2FT%2B8frWZEvp
%2Fkr6qcupvhOv%2FK%2BvBEj21Ch2g%2B6YB%2FfdQiDH%2B52GOLo%2FwkAAP%2F%2F
```

```xml
<?xml version="1.0"?>
<q1:AuthnRequest ID="ida90ebab54bac4916b25d9dcf5b72c8c3" Version="2.0" IssueInstant="2013-01-
28T19:53:46.0434803Z" Destination="https://login.test-nemlog-in.dk/adfs/ls/" IsPassive="false"
xmlns:q1="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">https://sp1.test-nemlog-in.dk/</Issuer>
  <Conditions xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
    <AudienceRestriction>
      <Audience>https://sp1.test-nemlog-in.dk/</Audience>
    </AudienceRestriction>
  </Conditions>
</q1:AuthnRequest>
```

Figure C.6: Fiddler - Decode *AuthnRequest*

Appendix D

Login Data

Example login data as received from DanId after successful authentication

```xml
<?xml version="1.0" encoding="UTF-8"?><openoces:signature
    xmlns:openoces="http://www.openoces.org/2006/07/signature#
    " version="0.1">
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig
        #" xmlns:openoces="http://www.openoces.org/2006/07/
        signature#">
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/
          TR/2001/REC-xml-c14n-20010315\"></
          ds:CanonicalizationMethod>
      <ds:SignatureMethod Algorithm="http://www.w3.org
          /2001/04/xmldsig-more#rsa-sha256\"></
          ds:SignatureMethod>
      <ds:Reference URI="#ToBeSigned">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/
            xmlenc#sha256\"></ds:DigestMethod>
        <ds:DigestValue>rlkD1arq3Z3apuPPz5YgSmQYDF5t4ie6+
            tn2qDrp0aQ=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>nEDrHw+oLQAbp9/q3ANAn53f5Q/
        mQ0rj4q1c6NKTS0BbDCwlpAx8dYCdnDIJMJfbxpeUHtgcXn09F
      6Rr/LDw1oGNa0pserddIuR8fRnTMkgtsNvnE+/g9hJ+
          kV9QdpMg5SDCIcHb51DLGrw2p0j66310t
      ChRCsMuUhEo8xbxuHn/+
          sxW3dCfB5QpavakoQpayFGgjviE9Gxly00QE6DFjESwfRVgWr8FV0je
          /\
          nuabVa4CmUoKZtqZGXWlJxtB8SvW6RDHZanK29bb1fUnZQv1teVmMXNhnAC
          +eCcF6+cC5gyynmqj3
      PvA27zwRJKd60kY/qvDalWTanaFtCg9rz2zY0Q==
    </ds:SignatureValue>
    <ds:KeyInfo>
      <ds:X509Data>
        <ds:X509Certificate>cert data</ds:X509Certificate>
      </ds:X509Data>
      <ds:X509Data>
        <ds:X509Certificate>cert data</ds:X509Certificate>
      </ds:X509Data><ds:X509Data>
        <ds:X509Certificate>cert data</ds:X509Certificate>
      </ds:X509Data>
    </ds:KeyInfo>
    <ds:Object xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        xmlns:openoces="http://www.openoces.org/2006/07/
        signature#" Id="ToBeSigned">
      <ds:SignatureProperties>
        <ds:SignatureProperty>
          <openoces:Name>host</openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              yes">TmVtTG9nLWlu</openoces:Value>
        </ds:SignatureProperty>
        <ds:SignatureProperty>
          <openoces:Name>logonto</openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              yes">TmVtTG9nLWlu</openoces:Value>
        </ds:SignatureProperty>
        <ds:SignatureProperty>
          <openoces:Name>openoces_opensign_layout_size_width</
              openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              no">NTY4</openoces:Value>
        </ds:SignatureProperty>
        <ds:SignatureProperty>
          <openoces:Name>openoces_opensign_layout_size_height<
              /openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              no">MTI5</openoces:Value>
        </ds:SignatureProperty>
        <ds:SignatureProperty>
          <openoces:Name>challenge</openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              no">
            OTYtREYtMDktRjUtNzAtRTAtOTgtRUQtMjUtOUMtMTEtOT
            gtNzgtNDctNzctQjQtMjEtOTItN0MtMEI=
          </openoces:Value>
        </ds:SignatureProperty>
        <ds:SignatureProperty>
          <openoces:Name>action</openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              no">bG9nb24=</openoces:Value>
        </ds:SignatureProperty>
        <ds:SignatureProperty>
          <openoces:Name>signtext</openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              no">T3BlbkxvZ29uIDE5LTAyLTEzIDE2OjU2</
              openoces:Value>
        </ds:SignatureProperty>
        <ds:SignatureProperty>
          <openoces:Name>
              openoces_opensign_layout_color_background</
              openoces:Name>
          <openoces:Value Encoding="base64" VisibleToSigner="
              no">MjU1LDI1NSwyNTU=</openoces:Value>
        </ds:SignatureProperty>
      </ds:SignatureProperties>
    </ds:Object>
  </ds:Signature>
</openoces:signature>
```

Listing D.1: Sample SAML AuthnRequest message.

Appendix E

CD content

The enclosed CD will include the following content

1. Source code
   - F* Source code and examples
   - F* Compiler used in this project
   - OOAPI
   - OIOSAML.NET
2. Fiddler trace archive