UNIVERSITÉ DU LUXEMBOURG

FACULTY OF SCIENCE, TECHNOLOGY AND COMMUNICATION

Verifying Multi-Party Authentication of Solid OIDC and VC Protocols

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Gilles SCHAUS

Supervisor:
Prof. Sjouke MAUW

Reviewer:
Prof. Dr. Peter Y. A. RYAN

Advisor:
Ross HORNE

July 2023

Abstract

This paper aims at analysing and verifying Solid Open ID Connect, a recent addition to the third party authentication protocols. During the analysis, it becomes clear that some aspects of this new protocol are not working as intended, giving attackers the opportunity to execute different forms of mixup attacks, resulting in possible access of private resources. After giving a potential fix and therefore mitigating the threat, this paper is going a step further and develops a new protocol providing third party authentication by relying on the concept of verifiable credentials. At the end, the key concepts of the new protocol are verified and constitute a solid base for further development.

Contents

1 Introduction **1**

2 ProVerif **3**

2.1 General Model Structure . . . 4

2.1.1 Definitions . . . 4

2.1.2 Agent Macros . . . 4

2.1.3 Open Network . . . 4

2.1.4 Assumptions . . . 5

2.1.5 Main . . . 5

2.2 Encountered Problems . . . 6

2.2.1 Bloating . . . 6

2.2.2 Identical Messages . . . 7

2.2.3 Non-Bitstring Messages . . . 7

3 Solid OpenID Connect **9**

3.1 HTTPS Handshake . . . 9

3.1.1 Simple vs Diffie Hellman . . . 9

3.1.2 Timing Results . . . 11

3.2 Authorization Code Grant Protocol . . . 12

3.2.1 Mixup Attack for OpenID Provider . . . 14

3.2.2 Cross-Site Request Forgery . . . 15

3.2.3 Providing a Fix . . . 17

3.2.4 Proposed Version Of SOIDC . . . 19

4 New VC-Based Third Party Authentication Protocol **21**

4.1 STS Handshake . . . 21

4.2 VC Issuance and Use . . . 22

5 Limitations / Further Work **25**

6 Summary **27**

Appendix: ProVerif Models **31**

.1 General Definitions . . . 31

.2 SOIDC - Issuer Mixup Attack . . . 32

.3 SOIDC - CSRF . . . 40

.4 VC-Based Third Party Authentication Protocol . . . 48

List of Figures

1.1 Solid Framework [1] 1

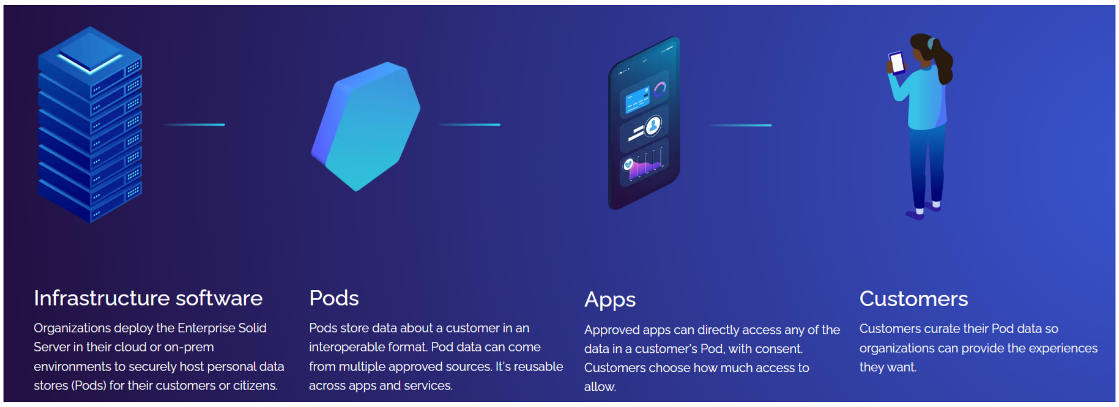
3.1 HTTPS Connect 10
3.2 HTTPS Connect Using Diffie Hellman 10
3.3 SOIDC 13
3.4 SOIDC Attack On Issuer Authentication 14
3.5 CSRF [2] 15
3.6 SOIDC Late Handshake 16
3.7 SOIDC CSRF Attack On Late Handshake 17
3.8 SOIDC Proposed Fix 18
3.9 SOIDC Proposed Fixed Version 19

4.1 STS Connect 22
4.2 VC Issuance and Use 23

Chapter 1

Introduction

In 1989, Tim Berners-Lee proposed an information management system and implemented the first successful communication between an HTTP client and server, making him the inventor of the World Wide Web (WWW). Now working as director of the World Wide Web Consortium (W3C), he is still shaping the web by creating and improving standards on an international scale. One of his projects is the Solid project, which aims at extending the web to include identity management and thus better access control for resources. The method used to achieve this feat is clear data separation by introducing Solid Pods like depicted in image 1.1.



Organizations deploy the Enterprise Solid Server in their cloud or on-prem environments to securely host personal data stores (Pods) for their customers or citizens.

Pods store data about a customer in an interoperable format. Pod data can come from multiple approved sources. It's reusable across apps and services.

Approved apps can directly access any of the data in a customer's Pod, with consent. Customers choose how much access to allow.

Customers curate their Pod data so organizations can provide the experiences they want.

Figure 1.1: Solid Framework [1]

These pods are personal data stores that provide a place for both user and service provider to access and update data. This data can also be shared by the user with third party clients by granting them the necessary permission using the proposed third party authentication protocol called Solid OpenID Connect [3]. In short, this protocol uses the users consent to allow the service provider the distribution of a dpop token to a third party client, which grants access to private resources on the pod. However, a previous paper by Fett et al [4] already discovered a potential mixup attack on OpenID Connect. As Solid OpenID Connect is based on OpenID Connect, the first goal of this paper is the search for a similar or even the same vulnerability in Solid OpenID Connect by creating a tool verifiable model.

1. The creation and verification of a model depicting Solid Open ID Connect. The criterion of the model was it to be sufficiently rich to demonstrate the presence of a suspected mixup attack.

The second goal of the paper follows-up on the first goal: during the development of the model and its verification, two different attacks have been discovered. To prevent the attacks from being exploitable, a potential fix is modelled and verified. The resulting protocol is then proposed as a new version of SOIDC.

2. Proposition for a fixed version of SOIDC, nullifying the previously discovered attacks.

The third and last goal of the paper addresses a common problem shared by third party authentication protocols: based on their particular architecture, the identity provider needs to learn about the resource being accessed before generating an access token. This allows them to track user behaviour and create profiles, which may be used to generate commercial profit. With the aim to increase the privacy of the user, this paper proposes a new form of protocol using verifiable credentials, leading to the identity provider losing the ability to track user behaviour and therefore increasing user privacy.

3. The creation and verification of a new protocol providing third party authentication using verifiable credentials.

This paper is divided into the following parts: chapter 2 presents the tool used in this paper to acquire proof for its security statements. Chapter 3 takes a closer look at Solid Open ID Connect (SOIDC), a specification allowing users to grant access to third party services to private information stored on supervised servers. Chapter 4 is taking the main idea of SOIDC and mixing it with the idea of verifiable credentials to create a new protocol that provides more privacy for the user and protects their information from being tracked by the identity provider. Chapter 5 addresses the limitations of this paper and gives some guidance for potential further work. Chapter 6 summarizes the most important aspects of the the paper. The appendix contains the models used to verify authentication properties, which were explained in the paper. These models may be used to recreate the verification proofs or as a starting point for future work.

Chapter 2

ProVerif

In order to achieve the goals mentioned above, it is important to go beyond intuition and use a tool providing the capabilities needed to formally proof and disprove statements. The tool chosen for this paper is called ProVerif and was created by Bruno Blanchet et al. [5]. It is a cryptographic protocol verifier using applied π-calculus to find solid evidence for different statements. It is capable of modelling many cryptographic primitives like both symmetric and asymmetric encryption, signatures and hash functions. Additionally, due to using well-chosen approximations, it is capable of handling an infinite number of sessions of the protocol, which may even run in parallel. These feats combined allow the verification of secrecy and authentication properties, which constitute the goal of this paper.

In regard to its trustworthiness, ProVerif states in its manual that it is sound, but not complete. In other words, in case a property is satisfied in the eyes of ProVerif, then the model does guarantee the property in question. In contrast, in case ProVerif is not finding a proof for a given property, the model may nonetheless provide that property. Being sound but not complete makes ProVerif an adequate choice for achieving the goal of this paper, as each verification result can be trusted [6].

ProVerif relies on the symbolic, Dolev-Yao model of cryptography. This means that is goes beyond a passive, eavesdropping attacker and assumes that the adversary is able to also read, modify, delay, delete and inject messages according to their current knowledge. Moreover, all models created in this paper are using an open network architecture, which makes it possible for the attacker to pose as any of the honest actors in each protocol. This mirrors the capabilities of attackers on the web, strengthening the models and their verification results even further. In the given models, each actor both has an identity, a secret and a matching public key, which may be used for asymmetric encryption and signatures. Identities and public keys are globally known, while the secret keys remain private. It is assumed that the attacker is not able to break any underlying cryptographic primitives like encryptions, reverse hashes or forge signatures.

2.1 General Model Structure

The syntax of ProVerif is broad and provides a whole set of various operations, which may be used in many different ways. This makes it important to keep the models well structured to make grasping the underlying meaning as easy as possible. Therefore, all models in this paper are using a simple structure, which is briefly explained in the following. Complete models are also accessible in the appendix 6.

2.1.1 Definitions

Firstly, the model starts out with general definitions. As general cryptographic primitives are not immediately part of the syntax, they need to defined using functions and reductions. The following example illustrates the definition of symmetric encryption and decryption:

```
fun enc(bitstring, SymmetricKey):bitstring.
reduc forall msg:bitstring,symk:SymmetricKey; dec(enc(msg,symk),symk)=msg.
```

After declaring all relevant cryptographic primitives, the model needs variables. Constants are global variables, which may be used by both the attacker and the honest actors. Free variables are used to check the attackers access on different values, which are subsequently used in secrecy claim evaluation. Events are used to save message sets, which are subsequently used in authentication claim evaluation:

```
const tag1:bitstring.
free free_code: bitstring [private].
event appSendsLastMessageToIssuer(bitstring, bitstring, bitstring).
```

2.1.2 Agent Macros

Right after the definitions come the actor declarations. Each actor participating in the protocol is listed here. This part may be seen as the core of the model, as it defines all messages being send from one actor to another. This implies that each message is appearing twice, as it must both be send as output by someone and read as input by someone else. The following example illustrates the message passing process between browser and app:

```
(* Browser *)
new n:bitstring;
new sskI:SecretKey;
let m_1 = aenc((tag1, n, pk(sskI)), pka) in
out(c, m_1);

(* App *)
in(c, m_1:bitstring);
let (tag1':bitstring, n:bitstring, spkI:PublicKey) = adec(m_1, ska) in
if (tag1 = tag1') then
```

2.1.3 Open Network

At the current state, the model is only checking properties in regard to an attacker, who may easily be distinguished from the honest actors. In order to give the attacker more control, the open network part of the model allows them to pose as any actor. This mimics the capabilities of a real attacker on the internet and gives the model additional importance when verifying its properties. The open network of for instance an app may be modelled like this:

```
let open_app(c:channel, a:bitstring, ska:SecretKey, u:bitstring,
pku:PublicKey, i:bitstring, pki:PublicKey) =
  in(c, e1:bitstring);
  in(c, e2:bitstring);
  if (e1 = u) && (e2 = i) then
    app(true, c, a, ska, u, pku, i, pki)
  else
  if (e1 <> u) && (e2 = i) then
    in(c, pke1:PublicKey);
    app(false, c, a, ska, e1, pke1, i, pki)
  else
  if (e1 = u) && (e2 <> i) then
    in(c, pke2:PublicKey);
    app(false, c, a, ska, u, pku, e2, pke2)
  else
  if (e1 <> u) && (e2 <> i) then
    in(c, pke1:PublicKey);
    in(c, pke2:PublicKey);
    app(false, c, a, ska, e1, pke1, e2, pke2)
  .
```

2.1.4 Assumptions

This part of the model defines the properties, which are checked by ProVerif. The simplest property to verify is secrecy of different free variables. It is used to establish if a protocol managed to successfully complete or if some important variable like a password remains secret:

```
query attacker(check_reach_app).
query attacker(free_password).
```

Furthermore, ProVerif is able to verify authentication claims using the previously explained event statements. In the following example, the claim states that an app may only complete its protocol when all messages it received match the messages send by the browser and issuer:

```
query m_1:bitstring, m_2:bitstring, m_3:bitstring, m_8:bitstring, m_9:bitstring,
m_10:bitstring, m_11:bitstring, m_12:bitstring;
    inj-event(appCompletesProtocol(m_1, m_2, m_3, m_8, m_9, m_10, m_11, m_12))
    ==>
    inj-event(browserSendsLastMessageToApp(m_1, m_2, m_3, m_8))
    &&
    inj-event(issuerSendsLastMessageToApp(m_9, m_10, m_11, m_12))
    .
```

2.1.5 Main

The main section of the model is the starting point of the ProVerif verification process. In this part, secret keys as well as public identities and keys are being created, before they are distributed to the different processes. The processes themselves are running in parallel and may be replicated any number of times. This mimics the attackers ability to start as many sessions as needed to find and exploit a vulnerability in the protocol:

```
(* create identities *)
new a:bitstring;
new u:bitstring;
new i:bitstring;

(*advertise identities*)
out(c,a);
out(c,u);
out(c,i);

(*declare secret keys*)
new ska:SecretKey;
new sku:SecretKey;
new ski:SecretKey;

(*advertise public keys*)
out(c,pk(ska));
out(c,pk(sku));
out(c,pk(ski));

(*generate variables*)
new password:bitstring;

(*let agents dance*)
(
    ! open_app      (c, a, ska, u, pk(sku), i, pk(ski))
  | ! open_browser  (c, u, sku, a, pk(ska), i, pk(ski), password)
  | ! open_issuer   (c, i, ski, a, pk(ska), u, pk(sku), password)
)
```

2.2 Encountered Problems

This part of the paper mentions some of the most dominant problems slowing down the progress of the model creation and verification. For each iterated problem, a possible solution is given to facilitate further development.

2.2.1 Bloating

Unfortunately during the development of the model, an inherent weakness of ProVerif made the verification process slow and tedious. When verifying a protocol, it may occur that the existence of infinite loops prevent the process form finishing. These infinite loops may result from message structures of the following form:

$$B \to A : senc(N_b, k)$$
$$A \to B : senc(f(N_b), k)$$

In order to find out the value of the nonce N_b, the attacker could take the second message $f(N_b)$ and use it as an input to restart the sequence again, obtaining $f(f(Nb))$. This could again be sent as the input of the first message, resulting in an infinite loop. Preventing this behaviour can be done by adding tags to the use of each cryptograhic primitive, like so:

$$B \to A : senc((c_0, N_b), k)$$
$$A \to B : senc((c_1, f(N_b)), k)$$

With tags, the messages can't be mixed anymore, which enables the verification process to finish. However, when using tags in the verification process, they should also be used in the implementation of the protocol. The security of the tagged protocol does not imply the security of the untagged version [6].

2.2.2 Identical Messages

Another common problem which leads to unnecessary bloating during the verification process is the appearance of exactly the same messages in the same protocol. In this case, ProVerif is not able to reliably match the outputs with their corresponding inputs. Therefore, when having the requirement to use identical messages at different places in the model, it is advised to artificially change some part of them to support ProVerif in its verification process. This may be achieved by adding some constant or using some function, which is not changing anything essential to the protocol. Alternatively, if sending a tuple containing different types of terms, reordering them may also work and is good practice, as it prevents potential replay attacks.

2.2.3 Non-Bitstring Messages

During the development of the protocol, it appeared that using messages of any type other than bitstring leads to unforeseen behaviour, often resulting in ProVerif not being able to match outputs with corresponding inputs and therefore not being able to finish the verification of the protocol. One example of such an occurrence is the transmission of the first message in a Diffie Hellman key exchange. This message is an element of the group G raised by some exponent, resulting in another element of group G. Without first converting it to a bitstring using a custom function, ProVerif was not able to complete the protocol.

Chapter 3

Solid OpenID Connect

Since the release of the OAuth 2.0 [RFC6749] and OpenID Connect Core 1.0 [OIDC.Core] web standards, they have quickly been adopted across the industry in various applications. These extended practical uses offered a lot of insight on their corresponding strengths through extensive real-world data and experience. However, with the growing importance of privacy and digital identities in our modernising world, an improvement was proposed. SOIDC aims at eliminating the previously necessary trust relationship between identity providers and resource servers including their corresponding authorisation servers [7]. This part of the paper analyses the message flow used to accomplish that goal by providing a verifiable model supporting the initial claims.

When taking a look at the SOIDC message flow described in their primer [8], it is clearly divided in two different parts: the first part called 'Authorization Code Grant' and takes care of the access token distribution, while the second part is called 'Request Flow' and takes care of the token and resource exchange. After finishing the modelling and verification process of the first part, it became apparent that it already showed signs of major vulnerabilities. Therefore, this paper focuses its analysis on the first part, gaining in-depth knowledge about these vulnerabilities and providing a possible fix. The second part of SOIDC is not included in this process and may be analysed in future work.

3.1 HTTPS Handshake

3.1.1 Simple vs Diffie Hellman

When beginning to model Solid Open ID Connect, the first step is to search for a way to model a secure connection on the web using HTTPS. This kind of connection is characterized through its uni-directional authentication, usually granted to important servers using certificates from trusted third parties to acknowledge their specific public keys. These certificates may then be verified by the connecting browser to guarantee communication with the chosen party and not some imposter. The main goal of such a connection process is the establishment of a symmetric session key, which may be used for any further communication. This results in forward secrecy, which means that even the compromise of the long-term keys cannot result in a loss of secrecy of older communication sessions. The connecting party usually authenticates later in the process through some sort of shared secret, like for instance a password. One possibility of modelling a protocol providing these properties is depicted in the msc 3.1. It intentionally forgoes authentication for the initiator, highlighting the fact that anyone is able to connect anonymously to a certified web server. This protocol is verified using ProVerif and provides exactly the anticipated security factors: both parties are able to finish execution of the handshake and sharing a symmetric key. Even though only the initiator is able to authenticate the responder, no attacker is able to compromise the symmetric session key.

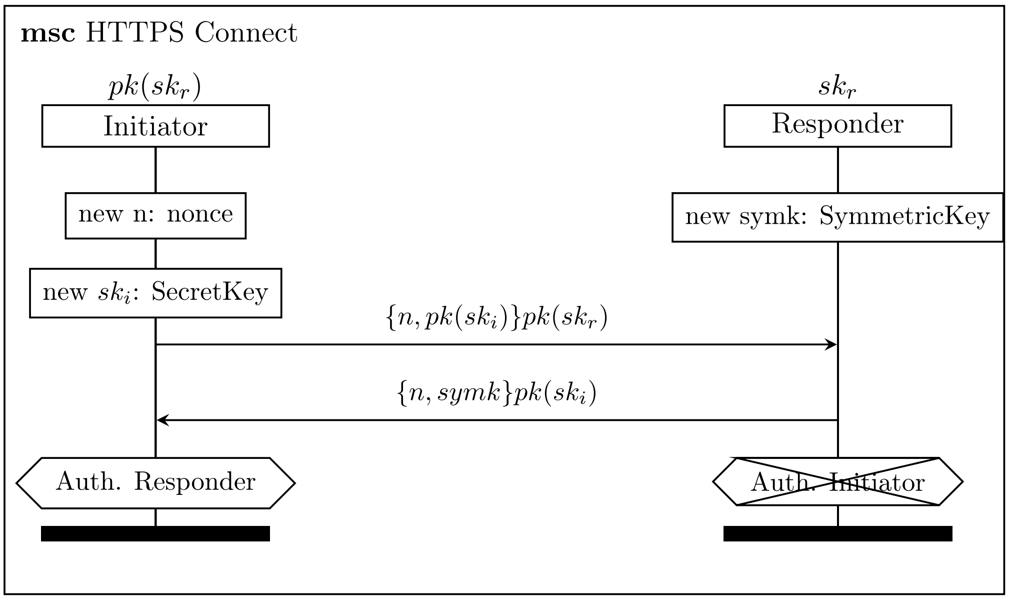


Figure 3.1: HTTPS Connect

Alternatively, a much stronger handshake mimicing HTTPS may be modelled by using a variation of the Diffie Hellman Key Exchange (DHKE) as depicted in the msc 3.2.

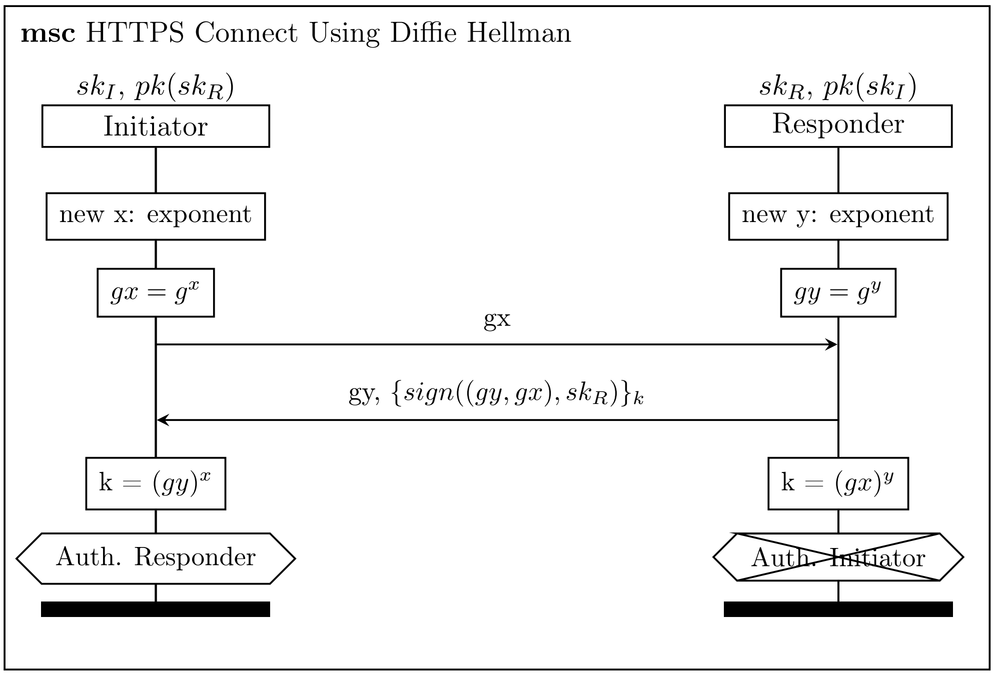


Figure 3.2: HTTPS Connect Using Diffie Hellman

As a standard DHKE is vulnerable to man in the middle attacks, the variation is using a signature to ensure the presence of the honest party and agreement on the same key material, preventing any attacker of inserting their own. Its strength comes from the fact that, in contrast to the previous protocol, both parties add partial information to a combined symmetric session key. This denies the capability of an attacker to pose as an honest party and to decide on the session key themselves. This protocol is verified using ProVerif and provides the exact same qualities as the previous handshake proposal. However, even tough this version is stronger, it puts a lot more strain on ProVerif when verifying the entire authorization code grant explained in the next section.

3.1.2 Timing Results

Verifying the simple HTTPS variant is fast with verification times generally below 5 minutes. In contrast, verifying the Diffie Hellman variant harbours some bloating problems as explained in section 2.2.1. This leads to long verification times, even with the use of tags already mitigating the problem. The following tables show in detail the verification results of SOIDC using the simple version in comparison with the Diffie Hellman version:

Solid Open ID Connect			
Property	Holds in Simple	Holds in DH	Verification Time DH
Auth. App	Yes	Yes	1h15
Auth. Browser	Yes	Yes	1h15
Auth. Issuer	No	OOM	2h15 (2.553 rules)

Solid Open ID Connect With Fix			
Property	Holds in Simple	Holds in DH	Verification Time DH
Auth. App	Yes	Yes	0h30
Auth. Browser	Yes	Yes	0h30
Auth. Issuer	Yes	Yes	1h30

As seen above, ProVerif is not capable of finding the issuer mixup attack when using the DH handshake, as it runs out of memory with 2.553 rules in queue. Even though this amount of rules is quite low, it can always happen that it bloats up again in the course of verification. The DH version with fix finished as expected. The following tables show in detail the verification results of SOIDC with a late handshake using the simple version in comparison with the Diffie Hellman version:

Solid Open ID Connect Late HS			
Property	Holds in Simple	Holds in DH	Verification Time DH
Auth. App	No	OOM	0h45 (7.060 rules)
Auth. Browser	Yes	OOM	1h00 (29.496 rules)
Auth. Issuer	No	OOM	1h15 (12.763 rules)

Solid Open ID Connect Late HS With Fix			
Property	Holds in Simple	Holds in DH	Verification Time DH
Auth. App	Yes	Yes	0h10
Auth. Browser	Yes	Yes	0h10
Auth. Issuer	Yes	Yes	0h10

As seen above, Proverif is again not capable of finding the attack when using the DH handshake, as it runs out of memory with thousands of rules in queue. However, the fixed version is verified and running fast with a verification time of 10 minutes. Without ProVerif being able to identify the attacks on SOIDC using the Diffie Hellman handshake, this paper uses the previously explained simpler but weaker version in the following analysis.

3.2 Authorization Code Grant Protocol

As mentioned before, the authorization code grant protocol is taking care of app authentication and the following access token distribution. The model of the protocol in question is based on the message flow provided in the official Solid OIDC Primer [8] and is depicted in the msc 3.3. Some of the major model design choices are explained in the following, starting with its actors:

App - The App constitutes the relying party (RP) in the process, needing permission of the Resource Owner (RO) to access private information.

Browser - The Browser of the RO is used to connect both App and Issuer, allowing them to talk via HTTPS requests. It also allows the RO to provide the needed password for the OpenID Provider (OP).

Issuer - The Issuer is the OP, providing the RP with the needed access token for the private resource if authentication succeeds.

After having declared the actors and their corresponding roles, the model skips the first few steps enumerated in the primer and supposes them to already be done. These steps include the 'Retrieve Profile' and the 'Retrieve OP Configuration' step, which are mainly used to setup communication between RP and OP. Additionally, the 'Fetch Client ID Document' step is skipped, as we suppose the OP already knows everything it needs without an attacker being able to interfere in this request.

Before starting communication and sending any message, the two actors in question are always using an HTTPS handshake to establish a connection. As previously mentioned, these handshakes are fashioned in a special way to only grant uni-lateral authentication. The little arrow in the msc 3.3 indicates which party is actually authenticated. Using the HTTPS handshake before every communication between two parties also mimics the use of callbacks to communicate, as the generated session keys tightly bind messages together.

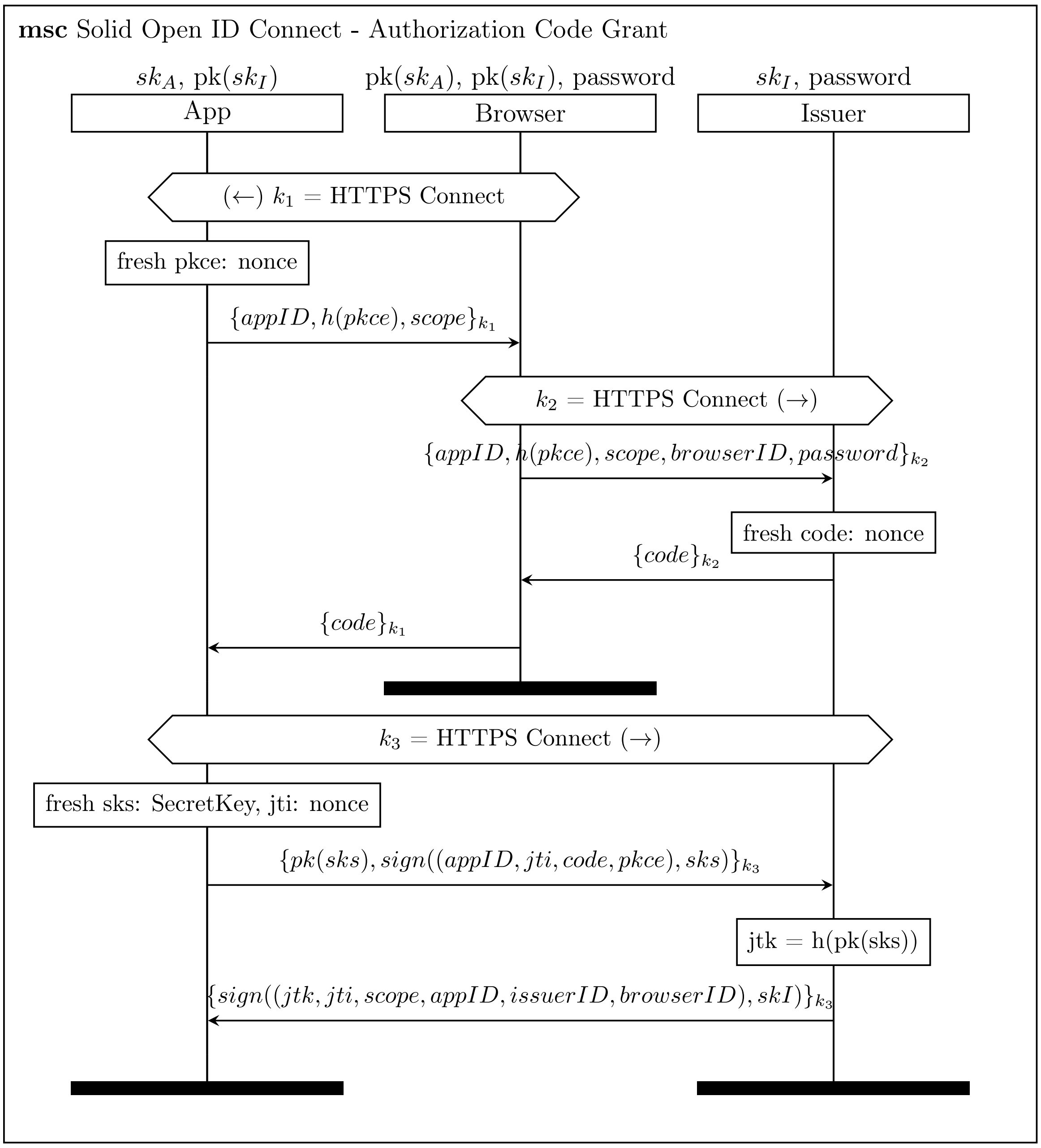

msc Solid Open ID Connect - Authorization Code Grant
sk_A, pk(sk_I)
pk(sk_A), pk(sk_I), password
sk_I, password
App
Browser
Issuer
(←) k_1 = HTTPS Connect
fresh pkce: nonce
$\{appID, h(pkce), scope\}_{k_1}$
k_2 = HTTPS Connect (→)
$\{appID, h(pkce), scope, browserID, password\}_{k_2}$
fresh code: nonce
$\{code\}_{k_2}$
$\{code\}_{k_1}$
k_3 = HTTPS Connect (→)
fresh sks: SecretKey, jti: nonce
$\{pk(sks), sign((appID, jti, code, pkce), sks)\}_{k_3}$
jtk = h(pk(sks))
$\{sign((jtk, jti, scope, appID, issuerID, browserID), skI)\}_{k_3}$

Figure 3.3: SOIDC

The first message actually being send is the authorisation request containing the appID, hashed pkce and scope. The appID is like the two other IDs a unique identifier for the app, the browser and the issuer respectively. Pkce is a generated code used by the app, which has the purpose to later guarantee the issuer that it is talking to the right app. The scope is related to the resource which is to be requested. Following the path over the browser, the RO is able to check on the values provided and login in case everything is valid. After receiving the values and checking them, the OP is generating a code which is send via the browser to the RP. This mimics SOIDCs send code to redirect uri step. Afterwards, the RP is using the code to request a dpop token, which can later be used for resource access. The request is mainly secured by a signature with a fresh generated key, as well as the pkce and the code from before. After checking the given information, the OP is issuing the requested token to the RP.

Verifying the model in ProVerif revealed that not every aspect of the model is secure. While the password stays secret throughout the entire protocol, the code is leaked to the attacker at some point. Further investigations revealed that not all parties are being authenticated correctly. The issuer in this case is not able to authenticate the app and the browser, meaning that there is a way the issuer finishes the protocol while talking to some entity pretending to either be app and / or browser. As this constitutes a serious vulnerability, the attack is explored in-depth in the next section of the paper. This completes the first goal of the paper as described in chapter 1.

3.2.1 Mixup Attack for OpenID Provider

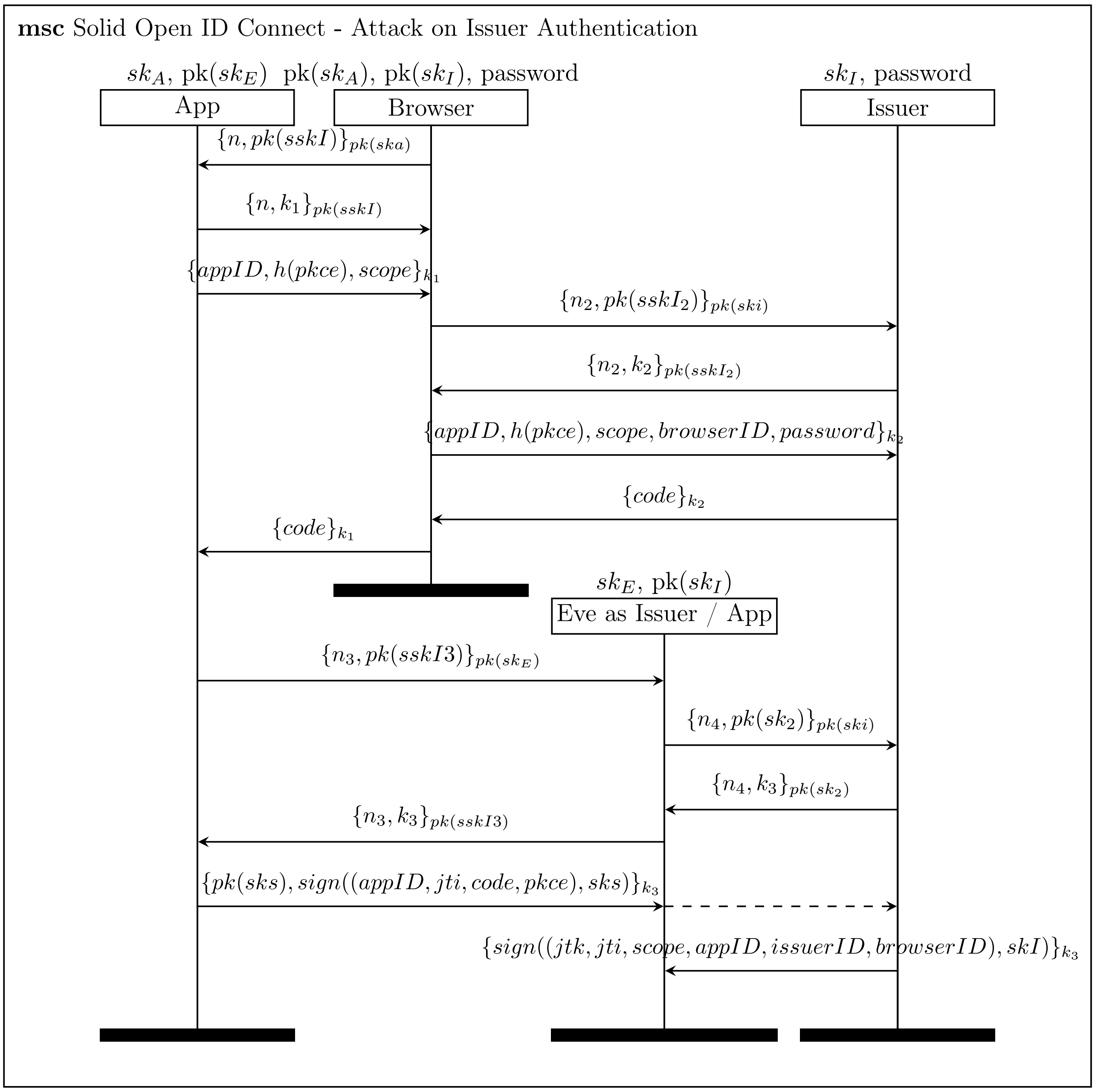

msc Solid Open ID Connect - Attack on Issuer Authentication
sk_A, pk(sk_E)
pk(sk_A), pk(sk_I), password
sk_I, password
App
Browser
Issuer
$\{n, pk(sskI)\}_{pk(ska)}$
$\{n, k_1\}_{pk(sskI)}$
$\{appID, h(pkce), scope\}_{k_1}$
$\{n_2, pk(sskI_2)\}_{pk(ski)}$
$\{n_2, k_2\}_{pk(sskI_2)}$
$\{appID, h(pkce), scope, browserID, password\}_{k_2}$
$\{code\}_{k_2}$
$\{code\}_{k_1}$
sk_E, pk(sk_I)
Eve as Issuer / App
$\{n_3, pk(sskI3)\}_{pk(sk_E)}$
$\{n_4, pk(sk_2)\}_{pk(ski)}$
$\{n_4, k_3\}_{pk(sk_2)}$
$\{n_3, k_3\}_{pk(sskI3)}$
$\{pk(sks), sign((appID, jti, code, pkce), sks)\}_{k_3}$
$\{sign((jtk, jti, scope, appID, issuerID, browserID), skI)\}_{k_3}$

Figure 3.4: SOIDC Attack On Issuer Authentication

In general, a mixup attack is a form of attack on a protocol, where the attacker Eve poses as one or more different agents to trick an honest party in sending sensitive information to them. This information may subsequently be used by Eve to perform usually restricted actions or get access to private information. Searching for such an attack in the model illustrating the authorisation code grant message flow, ProVerif revealed the attack depicted in the msc 3.4. The complete model of this protocol is available in the appendix .2.

In the first steps of the attack, Eve is not acting at all and lets the app receive the the issuer generated code following the usual protocol. However, Eve then poses as a fake issuer in such a way, that the app starts performing the HTTPS handshake with them. Halfway through the handshake, Eve then connects to the real issuer, posing as the honest app. By cleverly mixing the handshakes, Eve is able to generate the same session key k_3 for both communication channels. After receiving the token request with code and code verifier from the honest app, Eve proceeds to forward this message to the honest issuer. This leads to the issuer assuming that Eve must be the honest app and proceeding to send them the access token, giving her access to private resources. The occurrence of such an attack may be tracked down to the simple fact that the app is not aware of the issuerID corresponding to the honest issuer. This problem is later addressed in the section 3.2.3.

3.2.2 Cross-Site Request Forgery

In general, Cross-Site Request Forgeries (CSRF) aim at inducing users to perform actions they don't intend to perform. This includes actions like changing the password, changing their email address or sending money to somebody else. CSRF attacks only work if there is a relevant action to perform by the victim (like a password change), matching messages to the right session is handled by cookies and the hyperlink in question includes no parameter which is unknown to the attacker [9]. Figure 3.5 depicts the general structure of such an attack.

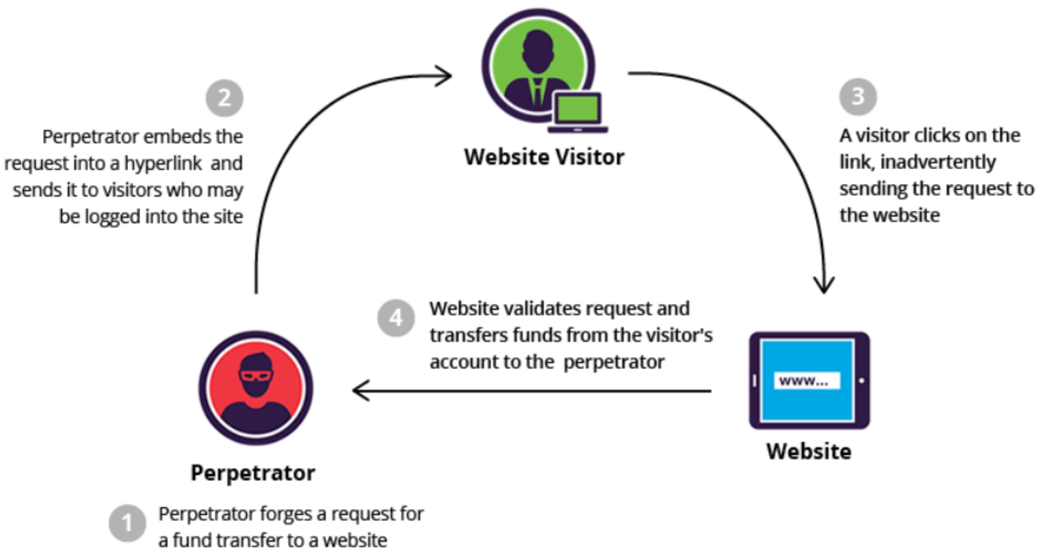

2
Perpetrator embeds the request into a hyperlink and sends it to visitors who may be logged into the site
Website Visitor
3
A visitor clicks on the link, inadvertently sending the request to the website
4
Website validates request and transfers funds from the visitor's account to the perpetrator
www...
Website
Perpetrator
1
Perpetrator forges a request for a fund transfer to a website

Figure 3.5: CSRF [2]

Even though the original model illustrating the authorisation code grant message flow does not allow for this kind of attack, it is important to note that implementation errors may occur. Such errors may lead to delayed authentication, which may be modelled by shifting the handshakes like depicted in msc 3.6, giving the attacker control over the initial authorisation request and allowing them to freely choose the values appID, pkce and scope. The complete model of this protocol is available in the appendix .3.

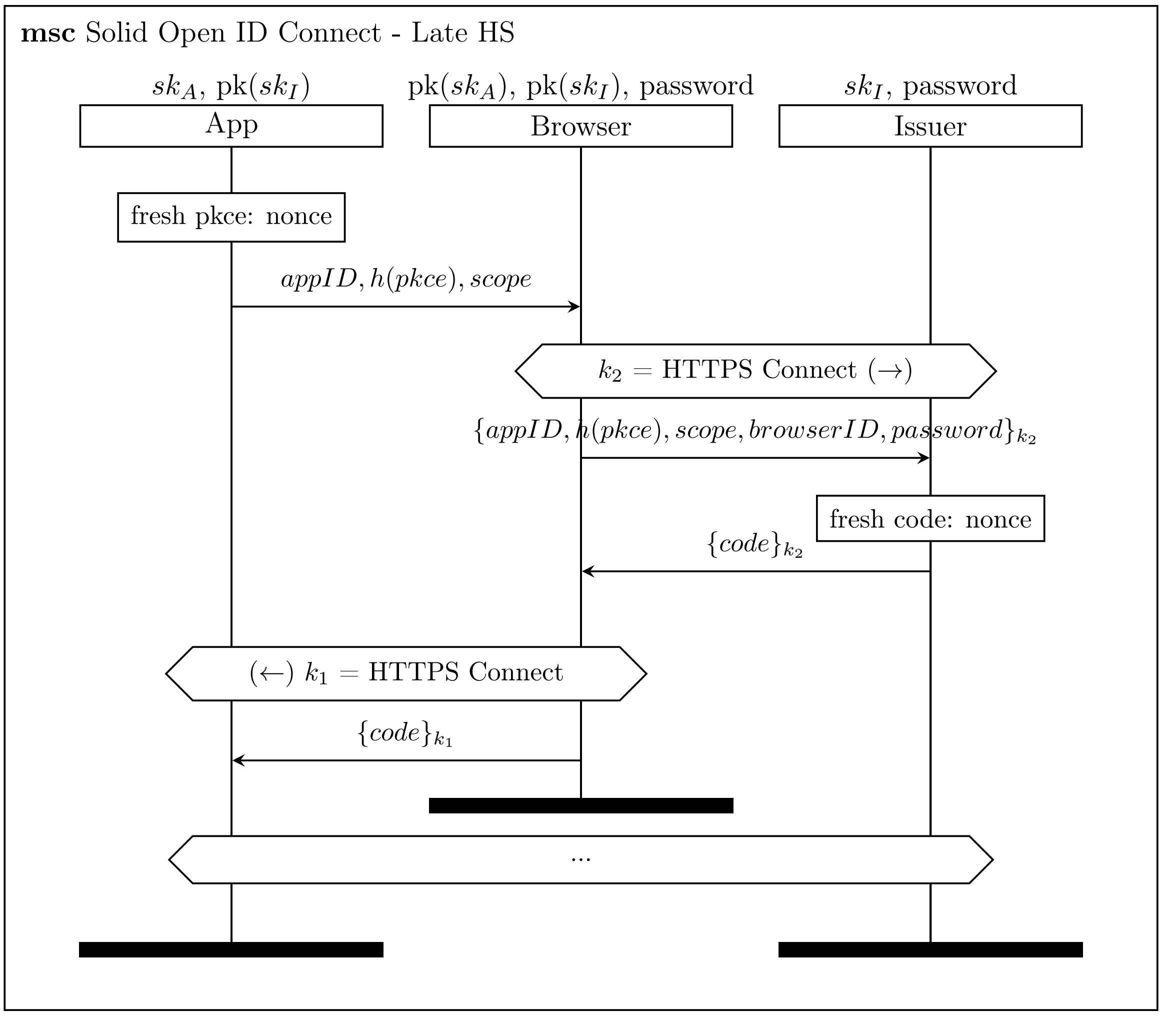

msc Solid Open ID Connect - Late HS
sk_A, pk(sk_I)
pk(sk_A), pk(sk_I), password
sk_I, password
App
Browser
Issuer
fresh pkce: nonce
$appID, h(pkce), scope$
k_2 = HTTPS Connect (→)
$\{appID, h(pkce), scope, browserID, password\}_{k_2}$
fresh code: nonce
$\{code\}_{k_2}$
(←) k_1 = HTTPS Connect
$\{code\}_{k_1}$
...

Figure 3.6: SOIDC Late Handshake

Running a ProVerif verification on this model reveals the now existing CRSF attack, which emerged as the app is no longer authenticating browser and issuer. This leads again to the code to be leaked to the attacker, which in combination with the ability of choosing the authorisation request themselves, gives them free hand in using it with an honest issuer and acquiring an access token. This attack is depicted in msc 3.7.

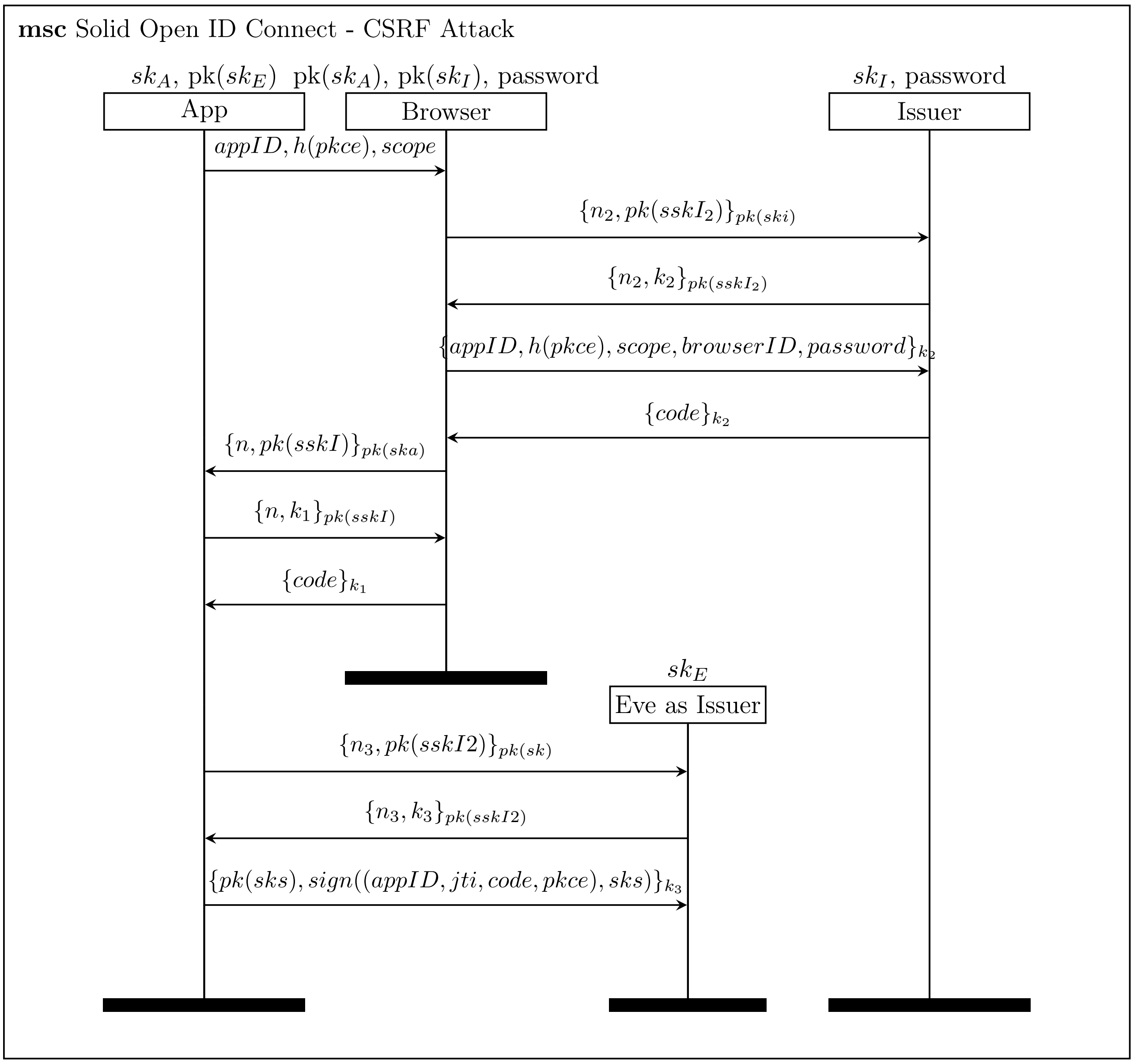

msc Solid Open ID Connect - CSRF Attack
sk_A, pk(sk_E)
pk(sk_A), pk(sk_I), password
sk_I, password
App
Browser
Issuer
$appID, h(pkce), scope$
$\{n_2, pk(sskI_2)\}_{pk(ski)}$
$\{n_2, k_2\}_{pk(sskI_2)}$
$\{appID, h(pkce), scope, browserID, password\}_{k_2}$
$\{code\}_{k_2}$
$\{n, pk(sskI)\}_{pk(ska)}$
$\{n, k_1\}_{pk(sskI)}$
$\{code\}_{k_1}$
sk_E
Eve as Issuer
$\{n_3, pk(sskI2)\}_{pk(sk)}$
$\{n_3, k_3\}_{pk(sskI2)}$
$\{pk(sks), sign((appID, jti, code, pkce), sks)\}_{k_3}$

Figure 3.7: SOIDC CSRF Attack On Late Handshake

3.2.3 Providing a Fix

Being prone to such devastating attacks is indeed troublesome, but luckily both attacks rely on the same vulnerability and may be fixed easily. While the browser is talking to the issuer, the app has no way of knowing which issuer the browser is actually talking to. This may lead to misunderstandings, which may be exploited by malicious parties. However, by adding the issuerID to the code like shown in the msc 3.8, the app learns about the identity of the conversing issuer. In case the identity is not matching with the apps expectation, the protocol can now be stopped, preventing sensitive information from leaking.

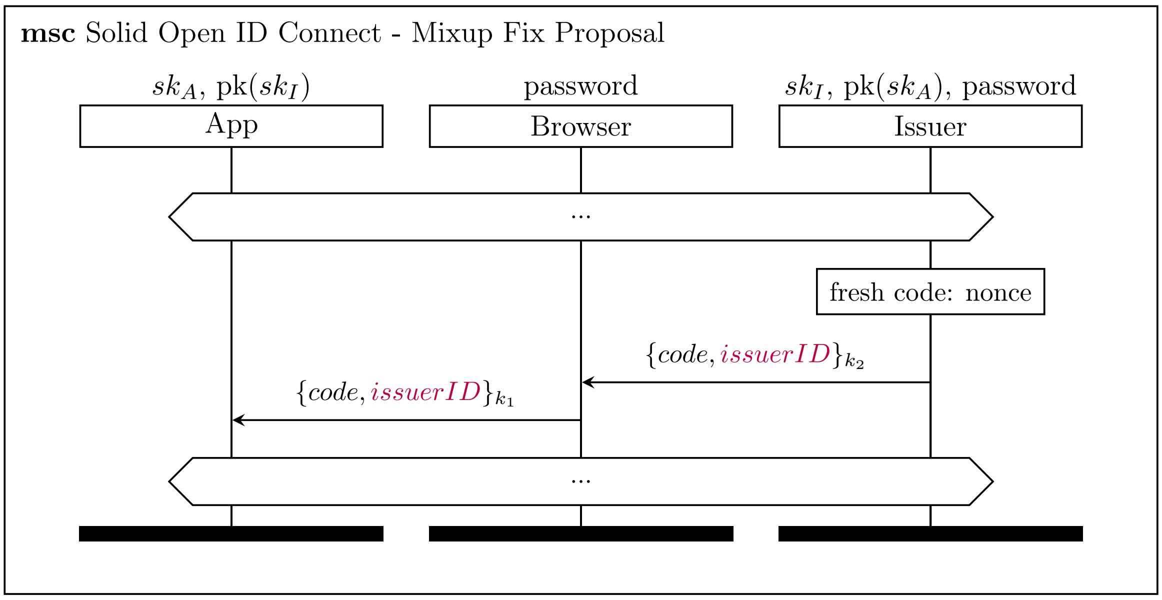

msc Solid Open ID Connect - Mixup Fix Proposal
sk_A, pk(sk_I)
App
password
Browser
sk_I, pk(sk_A), password
Issuer
...
fresh code: nonce
$\{code, issuerID\}_{k_2}$
$\{code, issuerID\}_{k_1}$
...

Figure 3.8: SOIDC Proposed Fix

Using ProVerif to verify the fix, it is proven that authentication now holds for all parties. This was tested for both the normal version and the version including the implementation error with the late handshake. Additionally, the code stays now secret throughout the entire run of the protocol. This denies both of the attacks mentioned beforehand. The following short snippet of ProVerif code illustrates the simplicity of the fix, as only two messages needed to be changed:

```
(* ---------- Issuer ---------- *)
(* Message 7 *)
new code:bitstring;
let m_7 = enc((tag7, code, i), k2) in
event issuerSendsLastMessageToUser(m_4, m_5, m_6, m_7);
out(c, m_7);

(* ---------- User ---------- *)
(* Message 7 *)
in(c, m_7:bitstring);
let (tag7':bitstring, code:bitstring, i':bitstring) = dec(m_7, k2) in
if (tag7 = tag7') then
if (i = i') then

(* Message 8 *)
let m_8 = enc((tag8, code, i), k1) in
event browserSendsLastMessageToApp(m_1, m_2, m_3, m_8);
out(c, m_8);

(* ---------- App ---------- *)
(* Message 8 *)
in(c, m_8:bitstring);
let (tag8':bitstring, code:bitstring, i':bitstring) = dec(m_8, k1) in
if (tag8 = tag8') then
if (i = i') then
```

Furthermore, applying the fix also allows for the pkce to be completely be omitted without losing security, as it is no longer the sole responsible for issuer authentication.

3.2.4 Proposed Version Of SOIDC

A new and improved version of the SOIDC authorisation code grant protocol is proposed in the following and depicted in the msc 3.9. It utilizes the fix proposed in section 3.2.3 and has had the code verifier pkce removed as it became obsolete in the process. Running the verification process of ProVerif on this model proved that authentication now holds for each party. Additionally, the code stays secret throughout the entire protocol. This completes the second goal of the paper as described in chapter 1.

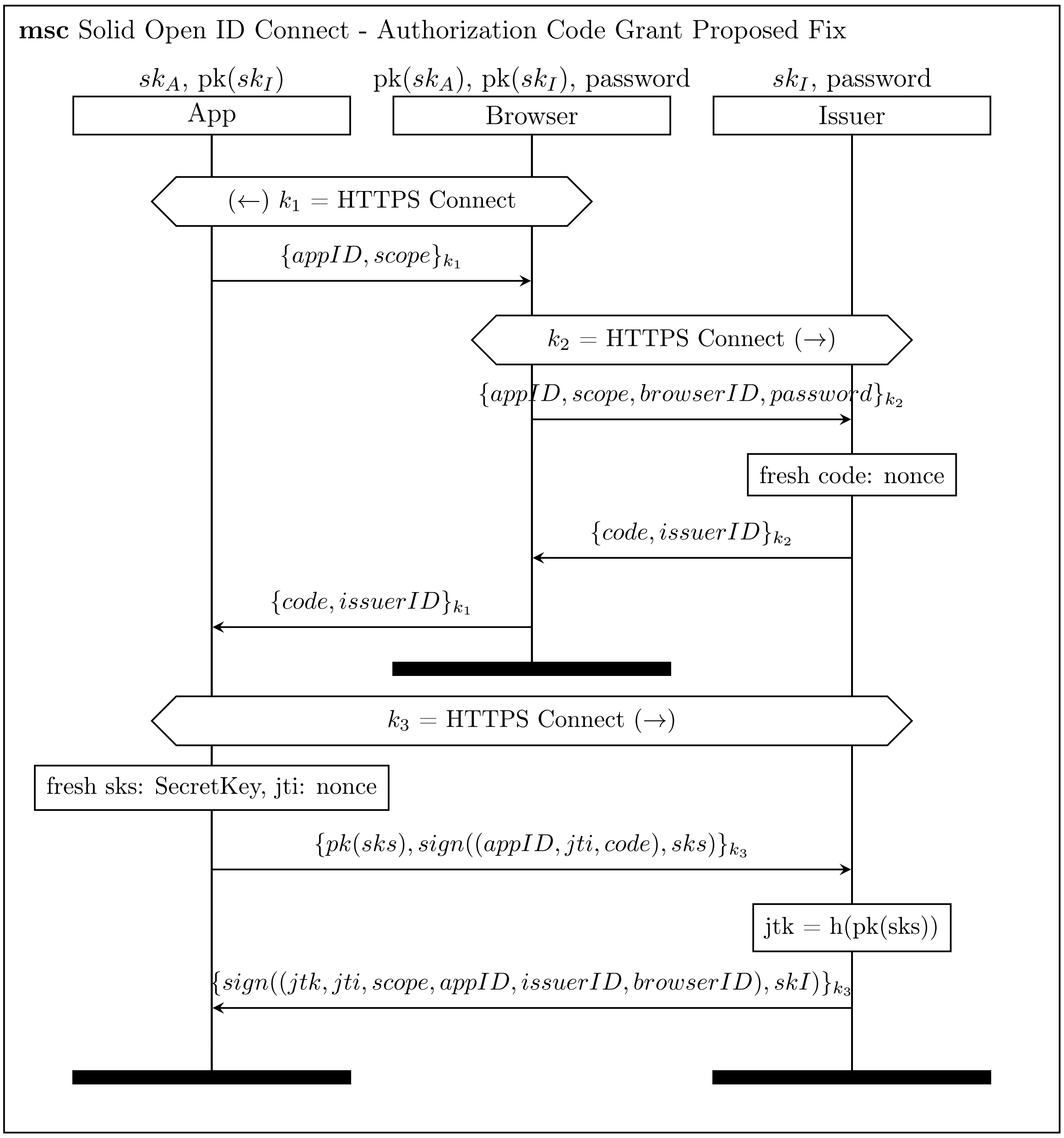

msc Solid Open ID Connect - Authorization Code Grant Proposed Fix
sk_A, pk(sk_I)
pk(sk_A), pk(sk_I), password
sk_I, password
App
Browser
Issuer
(←) k_1 = HTTPS Connect
$\{appID, scope\}_{k_1}$
k_2 = HTTPS Connect (→)
$\{appID, scope, browserID, password\}_{k_2}$
fresh code: nonce
$\{code, issuerID\}_{k_2}$
$\{code, issuerID\}_{k_1}$
k_3 = HTTPS Connect (→)
fresh sks: SecretKey, jti: nonce
$\{pk(sks), sign((appID, jti, code), sks)\}_{k_3}$
jtk = h(pk(sks))
$\{sign((jtk, jti, scope, appID, issuerID, browserID), skI)\}_{k_3}$

Figure 3.9: SOIDC Proposed Fixed Version

Chapter 4

New VC-Based Third Party Authentication Protocol

After having analysed and modelled the inner workings of Solid Open ID Connect, it becomes clear that relying on HTTPS connection protocols is not enough for the creation of secure communication channels. The existence of the previously discovered mixup and CSRF attacks prove the lack of control over the participants of the protocol. In order to regain control, the use of a bi-directional authenticating protocol is therefore proposed and modelled in the following. This should order the chaotic message flow jumping between all actors and provide less opportunities for a malicious party to infiltrate the system.

Additionally, the protocol flow of Solid Open ID Connect implies that the issuer contacted every time a user grants a third party access to private resources. The third party in question is therefore transmitted to the issuer, allowing them to create profiles and track user behaviour based on that information. Reclaiming that privacy may be achieved by circumventing the need for the issuer to know the third party. To achieve this feat, the concept of Verifiable Credentials (VC) is introduced in the following. In general, VCs are official documents issued by an authority, proving that an entity is indeed fulfilling some sort of requirement. They are the digital counterpart of for instance a passport or identity card, which are issued by the government and proving the holder's name, age etc. By transferring this system to a third party authentication protocol, it becomes possible for any user to provide trustworthy evidence to a third party, which allows them to access sensitive information on their behalf. By adding an expiry date, these credentials may be used multiple times before they need to be refreshed by issuer [10].

4.1 STS Handshake

Similar to the HTTPS connection protocol mentioned in chapter 3, the Station To Station (STS) Protocol [11] aims at establishing a symmetric session key, which may be used for communication while providing forward secrecy. It is based on a Diffie Hellman Key Exchange operating in a previously fixed group G. Both parties start by exchanging a group element, which is raised by individually created secret exponents. As only the result is transferred, no attacker is able to identify the secret exponent. By raising the received group element again by their individual exponents, both parties now obtain the same group element, which may be used as a symmetric key for future communication.

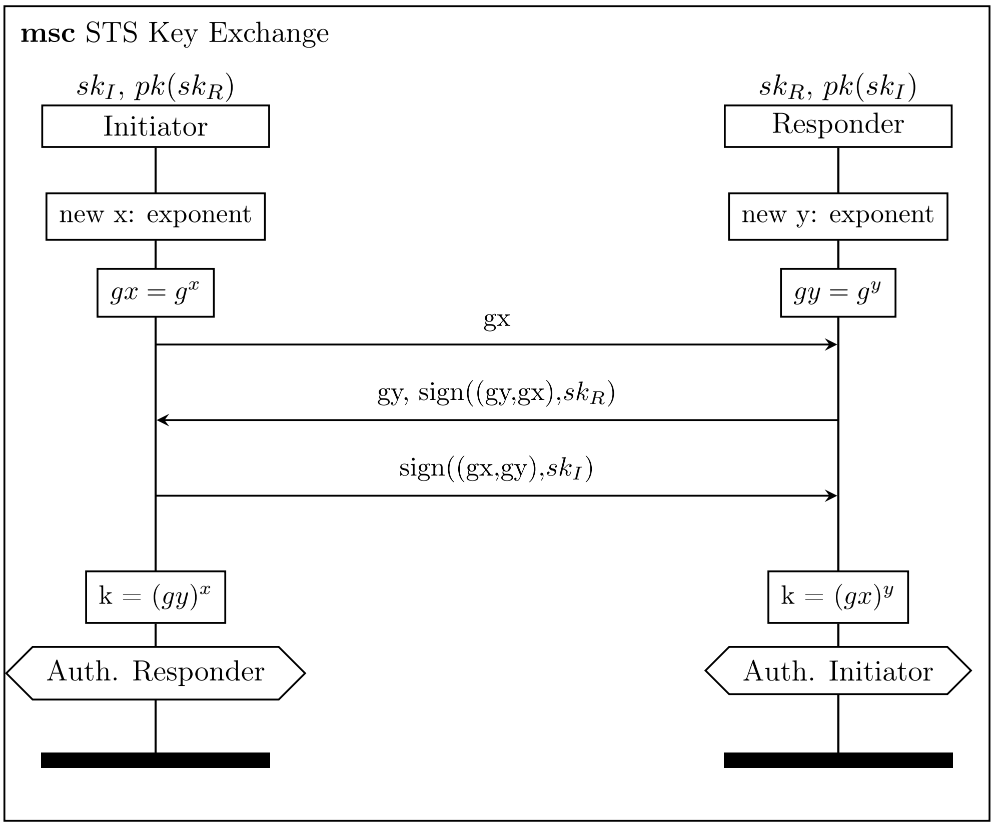


Figure 4.1: STS Connect

In general, this process is prone to a man in the middle attack. However, this is prevented by using signatures from both communicating parties to agree on the key material, also binding the generated symmetric key to their session. This process is illustrated in msc 4.1. Verifying the protocol with ProVerif reveals that both parties are authenticating each other like anticipated. Additionally, the generated symmetric key stays secret throughout the entire protocol.

4.2 VC Issuance and Use

By cooperating with Christoph Braun from the Karlsruhe Institute of Technology (KIT), a first version of a VC-based third party authentication protocol has been created. All of his work on this topic is accessible in the corresponding GitHub repository [12], while this paper shows a slightly modified version based on the result of our collaboration. The main difference is the use of expanded attacker capabilities by using the open network model as explained in chapter 2, as it allows the attacker to pose as any party involved in the communication process. The protocol is divided into two different parts: The issuance of the VC by the issuer and its use to access private resources in a pod. Both parts are divided by a fictive sovereignty line, depicting the breaking point at which the issuer loses sight of the users targeted pod, thus increasing their privacy. The entire protocol is depicted in msc 4.2, while both individual parts are explained in further detail in the following. The complete model of this protocol is available in the appendix .4.

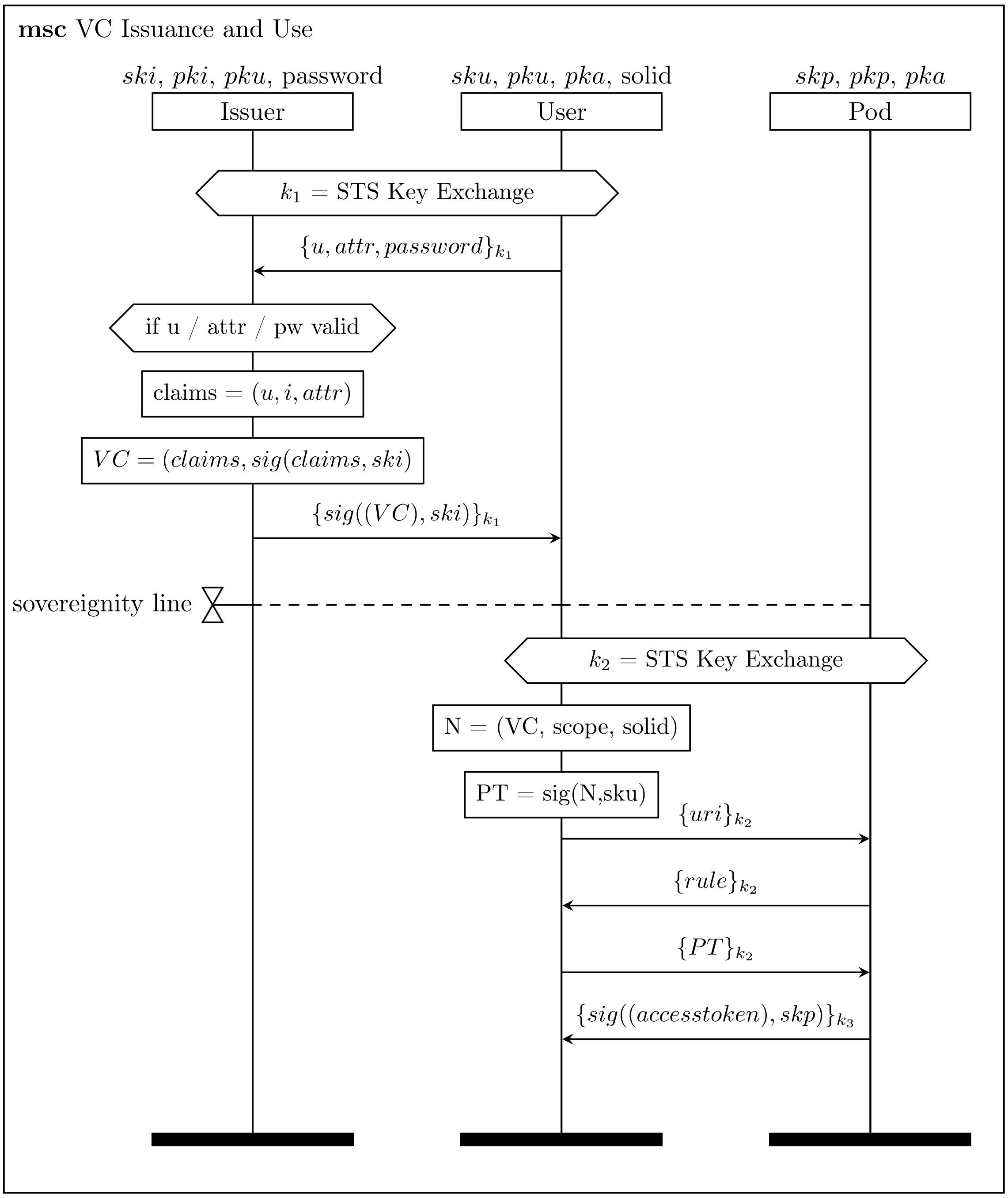


Figure 4.2: VC Issuance and Use

The first part of the proposed protocol consists in acquiring a verifiable credential from an official issuer. After connecting to the issuer in question, the user is sending their identifier u, some sort of attribute they want to be certified and a previously established password. The methods used to agree an a password are not included in this model. Each of the previously send terms fulfills a different role: The user identifier determines whose attribute should be approved by the issuer. The attribute determines what should be approved, like for instance a university degree or similar. The password is authenticating the user. After verifying the given information, the issuer creates a verifiable credential (VC) by signing the user identifier, their own identifier and the attribute in question.

This provides solid proof that the claim has been seen, verified and accorded by the issuer.

The second part of the proposed protocol starts below the sovereignty line and consists in the creation and use of the presentation token (PT) to get access to a specific resource. Transforming the VC to a PT by signing it is only possible for the user and is an important step as the holder of a VC is able to grant third party access to resources. Transferring the raw VC to any other party allows them to distribute access to private resources and thus should be avoided at all cost. In contrast, the possession of the PT only grants access to the resource specified in the scope and does not allow further distribution as it is already signed by the user. The next step consists in sending the resource access request uri to the pod. After receiving the request, the permission server is exposing the rule needed to fulfill in order to access the resource in question. The rule could for instance be the requirement of being registered on a specific website. With the matching PT, the app is able to provide proof of respecting the rule. Afterwards, the app receives an access token, which may be used to access the resource initially requested, completing the protocol.

The protocol was modelled and verified in ProVerif. The verification confirms that each party is able to complete the protocol and that both symmetric communication keys and the password stay secret during that time. Additionally, each actor is able to authenticate each other actor, meaning that no mixup attacks are possible in this scenario. This completes the third goal of the paper as described in chapter 1.

Chapter 5

Limitations / Further Work

Even though the models provided in this paper are as detailed as possible, it is important to make a note on the natural limitations of a model. These limitations emerge a model is in general not able to represent reality in its entirety. They usually use a set of assumptions, which aim at reducing the cheer amount of possibilities occurring in real life. If not chosen carefully, it may happen that some important aspects are abstracted away, hiding the potential existence of other attacks. That said, models are still a strong tool and should be used as much as possible during the creation process of any protocol. Each time a model is finding an attack, the discovered attack may be prevented before any malicious party is capable of exploiting it. This already protects user data even during development of a new protocol. Additionally, providing models and verification as proof for attacks or security properties is a guarantee for their existence.

When continuing the work on verifying Solid OpenID Connect, the next step would be to create a model illustrating its second part called 'Request Flow' as described in the official Solid Primer [8], which is describing the usage of the generated access token to receive the requested resource. Such a model could use the same general structure and HTTPS handshake as described in this paper, which would facilitate its development. Furthermore, checking for other properties than only secrecy and authentication could reveal new knowledge of potential vulnerabilities in either part.

Chapter 6

Summary

This paper has created its own ProVerif model depicting authorisation grant flow of Solid OpenID Connect. It uses a simple protocol mimicking the properties of a standard HTTPS structure to allow the different actors to connect. The resulting session key is then used to tightly bind all messages passed between two communicating agents, creating the notion of hyperlink requests and callbacks. The messages transfer the same content as described in the SOIDC Primer [8], which makes the model an accurate replication of the message flow in question. The discovery of two potential attacks even confirm the initial suspicions. This completes the first goal of the paper as stated in chapter 1.

Verifying the previously described model resulted in the discovery of a mixup and a cross-site request forgery attack. These attacks make it possible for a malicious party to access private resources, resulting in consequences of unforeseen severity. The simple solution of adding the issuer identity to the code transfer binds the generated code to the issuer and hereby prevents both attacks from being executed, as the attacker is no longer able use the code in sessions with different actors. A model containing the fix has been created and verified using ProVerif. This completes the second goal of the paper as stated in chapter 1.

Afterwards, this paper has presented a new protocol, which could be used to grant third party access to private resources. In contrast to already existing protocols, the new version uses verifiable credentials to limit the possibility for any issuer to track user behaviour. This increases user privacy by preventing the issuer to create user profiles with detailed information. The protocol has been modeled and verified with ProVerif. The results show that secrecy and authentication hold for the involved parties, which constitutes a solid starting point for further work on this protocol. This completes the third goal and last of the paper as stated in chapter 1.

Bibliography

[1] "Solid pods." https://www.inrupt.com/products/enterprise-solid-server. Accessed: 2023-07-08.

[2] "Cross site request forgery (csrf) attack." https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/?utm_source=google&utm_medium=cpc&utm_campaign=sw-waf-nord&utm_content=&utm_term=cross-site%20request%20forgery&gad=1&gclid=Cj0KCQjwnf-kBhCnARIsAFlg490xjzyAGD5QseM7KR0bh6FaJ7Bec8EIn4D_dQPVo7BwehSX_8YK-DwaArX6EALw_wcB. Accessed: 2023-07-01.

[3] "Solid." https://www.inrupt.com/solid. Accessed: 2023-07-08.

[4] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1204–1215, 2016.

[5] B. Blanchet, V. Cheval, X. Allamigeon, B. Smyth, and M. Sylvestre, "Proverif: cryptographic protocol verifier in the formal model." https://bblanche.gitlabpages.inria.fr/proverif/. Accessed: 2023-05-22.

[6] B. Blanchet, V. Cheval, B. Smyth, and M. Sylvestre, "Proverif 2.04: Automatic cryptographic protocol verifier, user manual and tutorial." https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf. Accessed: 2023-06-06.

[7] A. Coburn, E. Pavlik, and D. Zagidulin, "Solid-oidc." https://solidproject.org/TR/oidc. Accessed: 2023-06-05.

[8] J. Morgan, A. Coburn, and M. Bosquet, "Solid-oidc primer." https://solidproject.org/TR/oidc-primer. Accessed: 2023-06-28.

[9] "Cross-site request forgery (csrf)." https://portswigger.net/web-security/csrf. Accessed: 2023-07-01.

[10] M. Sporny, G. Noble, D. Longley, D. C. Burnett, B. Zundel, and K. D. Hartog, "Verifiable credentials data model v1.1." https://www.w3.org/TR/vc-data-model/#cryptography-suites-and-libraries. Accessed: 2023-06-05.

[11] B. O'Higgins, W. Diffie, L. Strawczynski, and R. De Hoog, "Encryption and isdn-a natural fit," in *Proceedings of the 1987 International Switching Symposium*, pp. 863–869, 1987.

[12] "Github containing sissi authentication and authorization protocols." https://github.com/uvdsl/proverif-sissi. Accessed: 2023-07-03.

[13] "Github containing soidc and vc proverif models." `https://github.com/g-schaus/proverif-soidc-and-vc`. Accessed: 2023-07-03.

Appendix: ProVerif Models

The appendix contains ProVerif models of the most important protocols presented in this paper. These protocols include the two different attacks on SOIDC as well as the newly proposed VC-based third party authentication protocol. All of the models presented here, together with their log files containing attack descriptions and traces, are also available at the following GitHub repository [13].

.1 General Definitions

The definitions of the cryptographic primitives are part of each ProVerif model of this paper and are therefore only listed once to save space. Forgetting to include them when copying the models will result in various error messages, preventing ProVerif from running the verifiaction process.

```
(* START keys *)
type SymmetricKey.
type SecretKey.
type PublicKey.
fun pk(SecretKey):PublicKey.
(* END keys *)

(* START group *)
type G.
type exponent.
fun exp(G, exponent): G.
fun g(exponent): G.
equation forall x: exponent, y: exponent; exp(g(x),y) = exp(g(y),x).
fun GtoSymK(G): SymmetricKey.
fun bitG(G):bitstring.
reduc forall elem:G; unbitG(bitG(elem))=elem.
(* END group *)

(* START hash *)
fun hash(bitstring): bitstring.
fun hash_pk(PublicKey): bitstring.
(* End hash *)

(* START symmetric encryption *)
fun enc(bitstring(*the message*), SymmetricKey):bitstring (*symmetric encrypt input msg*).
reduc forall msg:bitstring,symk:SymmetricKey; dec(enc(msg,symk),symk)=msg (*symmetric decrypt*).

fun symk_to_bit(SymmetricKey): bitstring.
reduc forall symk:SymmetricKey; bit_to_symk(symk_to_bit(symk)) = symk.
(* END symmetric encryption *)
(* START asymmetric encryption *)
fun aenc(bitstring(*the message*), PublicKey):bitstring (*asymmetric encrypt input msg*).
reduc forall msg:bitstring,sk:SecretKey; adec(aenc(msg,pk(sk)),sk)=msg (*asymmetric decrypt*).
(* END Asymmetric encryption *)

(* START digital signature *)
fun sign(bitstring(*the message*), SecretKey):bitstring (*sign the input msg*).
reduc forall msg:bitstring,sk:SecretKey; check(sign(msg,sk),pk(sk))=msg (*check signature, and get
reduc forall msg:bitstring,sk:SecretKey; getMsg(sign(msg,sk))=msg (*just get msg*).
(* END digital signature *)

(* START tags *)
const tag1:bitstring.
const tag2:bitstring.
const tag3:bitstring.
(* ... *)
(* END tags *)
```

.2 SOIDC - Issuer Mixup Attack

This model depicts the mixup attack as described in the msc 3.4. Note that ProVerif recognizes the issuer's inability of authenticating the other actors, leading to the vulnerability.

```
(* ------------------------------------------------------------------------ *)
(* Solid Open ID Connect *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* Start Definitions *)
(* ------------------------------------------------------------------------ *)

(* Do not forget to insert the previously mentioned definitions as explained in appendix 1 *)

(* SECRECY *)

free check_reach_app:bitstring [private].
free check_reach_browser:bitstring [private].
free check_reach_issuer:bitstring [private].

free free_code: bitstring [private].
free free_password: bitstring [private].

(* AUTHENTICATION *)

(* App *)
event appSendsLastMessageToIssuer(bitstring, bitstring, bitstring).
event appSendsLastMessageToBrowser(bitstring, bitstring, bitstring).
event appCompletesProtocol(bitstring, bitstring, bitstring, bitstring, bitstring,
bitstring, bitstring, bitstring).

(* Browser *)
event browserSendsLastMessageToIssuer(bitstring, bitstring, bitstring).
event browserSendsLastMessageToApp(bitstring, bitstring, bitstring, bitstring).
event browserCompletesProtocol(bitstring, bitstring, bitstring, bitstring,
bitstring, bitstring, bitstring).

(* Issuer *)
event issuerSendsLastMessageToApp(bitstring, bitstring, bitstring, bitstring).
event issuerSendsLastMessageToBrowser(bitstring, bitstring, bitstring, bitstring).
event issuerCompletesProtocol(bitstring, bitstring, bitstring, bitstring,
bitstring, bitstring).

(* ---------------------------------------------------------------------------- *)
(* End Definitions *)
(* Start Agent Macros *)
(* ---------------------------------------------------------------------------- *)

let app(talksOnlyToHonest:bool, c:channel, a:bitstring, ska:SecretKey, b:bitstring,
pkb:PublicKey, i:bitstring, pki:PublicKey) =

    (* !Connecting to Browser *)

    (* Message 1 *)
    in(c, m_1:bitstring);
    let (tag1':bitstring, n:bitstring, spkI:PublicKey) = adec(m_1, ska) in
    if (tag1 = tag1') then

    (* Message 2 *)
    new k1: SymmetricKey;
    let m_2 = aenc((tag2, n, k1), spkI) in
    out(c, m_2);

    (* !Talking to Browser*)

    (* Message 3 *)
    new pkce:bitstring;
    new scope:bitstring;
    let m_3 = enc((tag3, a, hash(pkce), scope), k1) in
    event appSendsLastMessageToBrowser(m_1, m_2, m_3);
    out(c, m_3);

    (* ... Waiting For Issuer Response While It Talks To The Browser ... *)

    (* Message 8 *)
    in(c, m_8:bitstring);
    let (tag8':bitstring, code:bitstring) = dec(m_8, k1) in
    if (tag8 = tag8') then

    (* !Connecting to Issuer *)

    (* Message 9 *)
    new n2:bitstring;
    new sskI2:SecretKey;
    let m_9 = aenc((tag9, n2, pk(sskI2)), pki) in
    out(c, m_9);

    (* Message 10 *)
    in(c, m_10:bitstring);
    let (tag10':bitstring, n2':bitstring, k3:SymmetricKey) = adec(m_10, sskI2) in
    if (tag10 = tag10') then
    if (n2 = n2') then

    (* !Talking to Issuer*)

    (* Message 11 *)
    new sks:SecretKey;
    new jti:bitstring;
    let T2 = (a, jti, code, pkce) in
    let m_11 = enc((tag11, pk(sks), sign(T2, sks)), k3) in
    event appSendsLastMessageToIssuer(m_9, m_10, m_11);
    out(c, m_11);

    (* Message 12 *)
    in(c, m_12:bitstring);
    out(c, check_reach_app);
    let (tag12':bitstring, signed_M:bitstring) = dec(m_12, k3) in
    if (tag12 = tag12') then
    let M = check(signed_M, pki) in
    let (jtk:bitstring, jti':bitstring, scope':bitstring, a':bitstring, i':bitstring,
    b':bitstring) = M in

    if (jtk = hash_pk(pk(sks))) then
    if (jti = jti') then
    if (scope = scope') then
    if (a = a') then
    if (i = i') then
    if (b = b') then

    (* ----------------------------------------------------------- *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_app);

    event appCompletesProtocol(m_1, m_2, m_3, m_8, m_9, m_10, m_11, m_12);

    out(ch(code), free_code);

    0.

let browser(talksOnlyToHonest:bool, c:channel, b:bitstring, skb:SecretKey, a:bitstring,
pka:PublicKey, i:bitstring, pki:PublicKey, password:bitstring) =

    (* !Connecting to App *)

    (* Message 1 *)
    new n:bitstring;
    new sskI:SecretKey;
    let m_1 = aenc((tag1, n, pk(sskI)), pka) in
    out(c, m_1);

    (* Message 2 *)
    in(c, m_2:bitstring);
    let (tag2':bitstring, n':bitstring, k1:SymmetricKey) = adec(m_2, sskI) in
```

```
if (tag2 = tag2') then
if (n = n') then

(* !Talking to App *)

(* Message 3 *)
in(c, m_3:bitstring);
let (tag3':bitstring, a':bitstring, h_pkce:bitstring, scope:bitstring) =
dec(m_3, k1) in
if (tag3 = tag3') then
if (a = a') then

(* !Connecting to Issuer *)

(* Message 4 *)
new n2:bitstring;
new sskI:SecretKey;
let m_4 = aenc((tag4, n2, pk(sskI)), pki) in
out(c, m_4);

(* Message 5 *)
in(c, m_5:bitstring);
let (tag5':bitstring, n2':bitstring, k2:SymmetricKey) = adec(m_5, sskI) in
if (tag5 = tag5') then
if (n2 = n2') then

(* !Talking to Issuer *)

(* Message 6 *)
let m_6 = enc((tag6, a, h_pkce, scope, b, password), k2) in
event browserSendsLastMessageToIssuer(m_4, m_5, m_6);
out(c, m_6);

(* Message 7 *)
in(c, m_7:bitstring);
let (tag7':bitstring, code:bitstring) = dec(m_7, k2) in
if (tag7 = tag7') then

(* !Talking to App *)

(* Message 8 *)
let m_8 = enc((tag8, code), k1) in
event browserSendsLastMessageToApp(m_1, m_2, m_3, m_8);
out(c, m_8);

(* ---------------------------------------------------------------- *)
(* Query Preparations *)

if talksOnlyToHonest then

out(c, check_reach_browser);

event browserCompletesProtocol(m_1, m_2, m_3, m_4, m_5, m_6, m_7);

out(ch(code), free_code);
out(ch(password), free_password);
```

```
    0.

let issuer(talksOnlyToHonest:bool, c:channel, i:bitstring, ski:SecretKey, a:bitstring,
pka:PublicKey, b:bitstring, pkb:PublicKey, password:bitstring) =

    (* !Connecting to Browser *)

    (* Message 4 *)
    in(c, m_4:bitstring);
    let (tag4':bitstring, n:bitstring, spkI:PublicKey) = adec(m_4, ski) in
    if (tag4 = tag4') then

    (* Message 5 *)
    new k2: SymmetricKey;
    let m_5 = aenc((tag5, n, k2), spkI) in
    out(c, m_5);

    (* !Talking to Browser *)

    (* Message 6 *)
    in(c, m_6:bitstring);
    let (tag6':bitstring, a':bitstring, h_pkce:bitstring, scope:bitstring, b':bitstring,
    password':bitstring) = dec(m_6, k2) in
    if (tag6 = tag6') then

    if (a = a') then
    if (b = b') then
    if (password = password') then

    (* Message 7 *)
    new code:bitstring;
    let m_7 = enc((tag7, code), k2) in
    event issuerSendsLastMessageToBrowser(m_4, m_5, m_6, m_7);
    out(c, m_7);

    (* Waiting for Browser to transfer code to App *)

    (* !Connecting to App *)

    (* Message 9 *)
    in(c, m_9:bitstring);
    let (tag9':bitstring, n2:bitstring, spkI2:PublicKey) = adec(m_9, ski) in
    if (tag9 = tag9') then

    (* Message 10 *)
    new k3: SymmetricKey;
    let m_10 = aenc((tag10, n2, k3), spkI2) in
    out(c, m_10);

    (* !Talking to App*)

    (* Message 11 *)
    in(c, m_11:bitstring);
    let (tag11':bitstring, pksks:PublicKey, signature:bitstring) = dec(m_11, k3) in
    if (tag11 = tag11') then
    let (a'':bitstring, jti:bitstring, code':bitstring, pkce:bitstring) =
    check(signature, pksks) in

    if (a = a'') then
    if (code = code') then
    if (h_pkce = hash(pkce)) then

    (* Message 12 *)
    let jtk = hash_pk(pksks) in
    let M = (jtk, jti, scope, a, i, b) in
    let m_12 = enc((tag12, sign(M, ski)), k3) in
    event issuerSendsLastMessageToApp(m_9, m_10, m_11, m_12);
    out(c, m_12);

    (* ------------------------------------------------------------ *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_issuer);

    event issuerCompletesProtocol(m_4, m_5, m_6, m_9, m_10, m_11);

    out(ch(code), free_code);
    out(ch(password), free_password);

    0.

(* ------------------------------------------------------------------------ *)
(* End Agent Macros *)
(* Start Open Network *)
(* ------------------------------------------------------------------------ *)

let open_app(c:channel, a:bitstring, ska:SecretKey, b:bitstring, pkb:PublicKey,
i:bitstring, pki:PublicKey) =
  in(c, e1:bitstring);
  in(c, e2:bitstring);
  if (e1 = b) && (e2 = i) then
    app(true, c, a, ska, b, pkb, i, pki)
  else
  if (e1 <> b) && (e2 = i) then
    in(c, pke1:PublicKey);
    app(false, c, a, ska, e1, pke1, i, pki)
  else
  if (e1 = b) && (e2 <> i) then
    in(c, pke2:PublicKey);
    app(false, c, a, ska, b, pkb, e2, pke2)
  else
  if (e1 <> b) && (e2 <> i) then
    in(c, pke1:PublicKey);
    in(c, pke2:PublicKey);
    app(false, c, a, ska, e1, pke1, e2, pke2)
  .

let open_browser(c:channel, b:bitstring, skb:SecretKey, a:bitstring, pka:PublicKey,
i:bitstring, pki:PublicKey, password:bitstring) =
  in(c, e1:bitstring);
  in(c, e2:bitstring);
  if (e1 = a) && (e2 = i) then
    browser(true, c, b, skb, a, pka, i, pki, password)
  else
  if (e1 <> a) && (e2 = i) then
    in(c, pke1:PublicKey);
    browser(false, c, b, skb, e1, pke1, i, pki, password)
  else
  if (e1 = a) && (e2 <> i) then
    in(c, pke2:PublicKey);
    in(c, wrong_pw:bitstring);
    browser(false, c, b, skb, a, pka, e2, pke2, wrong_pw)
  else
  if (e1 <> a) && (e2 <> i) then
    in(c, pke1:PublicKey);
    in(c, pke2:PublicKey);
    in(c, wrong_pw:bitstring);
    browser(false, c, b, skb, e1, pke1, e2, pke2, wrong_pw)
  .

let open_issuer(c:channel, i:bitstring, ski:SecretKey, a:bitstring, pka:PublicKey,
b:bitstring, pkb:PublicKey, password:bitstring) =
  in(c, e1:bitstring);
  in(c, e2:bitstring);
  if (e1 = a) && (e2 = b) then
    issuer(true, c, i, ski, a, pka, b, pkb, password)
  else
  if (e1 <> a) && (e2 = b) then
    in(c, pke1:PublicKey);
    issuer(false, c, i, ski, e1, pke1, b, pkb, password)
  else
  if (e1 = a) && (e2 <> b) then
    in(c, pke2:PublicKey);
    in(c, wrong_pw:bitstring);
    issuer(false, c, i, ski, a, pka, e2, pke2, wrong_pw)
  else
  if (e1 <> a) && (e2 <> b) then
    in(c, pke1:PublicKey);
    in(c, pke2:PublicKey);
    in(c, wrong_pw:bitstring);
    issuer(false, c, i, ski, e1, pke1, e2, pke2, wrong_pw)
  .


(* ------------------------------------------------------------------------ *)
(* End Open Network *)
(* Start Assumptions *)
(* ------------------------------------------------------------------------ *)

(* SECRECY *)

query attacker(check_reach_app).
query attacker(check_reach_browser).
query attacker(check_reach_issuer).
query attacker(free_code).
query attacker(free_password).

(* AUTHENTICATION *)

(* App Completes Protocol *)
query m_1:bitstring, m_2:bitstring, m_3:bitstring, m_8:bitstring, m_9:bitstring,
m_10:bitstring, m_11:bitstring, m_12:bitstring;
    inj-event(appCompletesProtocol(m_1, m_2, m_3, m_8, m_9, m_10, m_11, m_12))
    ==>
    inj-event(browserSendsLastMessageToApp(m_1, m_2, m_3, m_8))
    &&
    inj-event(issuerSendsLastMessageToApp(m_9, m_10, m_11, m_12))
    .

(* Browser Completes Protocol *)
query m_1:bitstring, m_2:bitstring, m_3:bitstring, m_4:bitstring, m_5:bitstring,
m_6:bitstring, m_7:bitstring;
    inj-event(browserCompletesProtocol(m_1, m_2, m_3, m_4, m_5, m_6, m_7))
    ==>
    inj-event(appSendsLastMessageToBrowser(m_1, m_2, m_3))
    &&
    inj-event(issuerSendsLastMessageToBrowser(m_4, m_5, m_6, m_7))
    .

(* Issuer Completes Protocol *)
query m_4:bitstring, m_5:bitstring, m_6:bitstring, m_9:bitstring, m_10:bitstring,
m_11:bitstring;
    inj-event(issuerCompletesProtocol(m_4, m_5, m_6, m_9, m_10, m_11))
    ==>
    inj-event(appSendsLastMessageToIssuer(m_9, m_10, m_11))
    &&
    inj-event(browserSendsLastMessageToIssuer(m_4, m_5, m_6))
    .

(* ------------------------------------------------------------------------- *)
(* End Assumptions *)
(* Start Main *)
(* ------------------------------------------------------------------------- *)

free c:channel.
process (*use root process for setup *)

    (* create identities *)
    new a:bitstring;
    new b:bitstring;
    new i:bitstring;

    (*advertise identities*)
    out(c,a);
    out(c,b);
    out(c,i);

    (*declare secret keys*)
    new ska:SecretKey;
    new skb:SecretKey;
    new ski:SecretKey;

    (*advertise public keys*)
    out(c,pk(ska));
    out(c,pk(skb));
    out(c,pk(ski));

    (*generate variables*)
    new password:bitstring;

    (*let agents dance*)
    (
        ! open_app      (c, a, ska, b, pk(skb), i, pk(ski))
      | ! open_browser  (c, b, skb, a, pk(ska), i, pk(ski), password)
      | ! open_issuer   (c, i, ski, a, pk(ska), b, pk(skb), password)
    )

(* ------------------------------------------------------------------------ *)
(* End Main *)
(* ------------------------------------------------------------------------ *)
```

.3 SOIDC - CSRF

This model depicts the mixup attack as described in the msc 3.7. Note that ProVerif recognizes the app's inability of authenticating the other actors, leading to the vulnerability.

```
(* ------------------------------------------------------------------------ *)
(* Solid Open ID Connect - Late Handshake Variant *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* Start Definitions *)
(* ------------------------------------------------------------------------ *)

(* Do not forget to insert the previously mentioned definitions as explained in appendix 1 *)

(* SECRECY *)

free check_reach_app:bitstring [private].
free check_reach_browser:bitstring [private].
free check_reach_issuer:bitstring [private].

free free_code: bitstring [private].
free free_password: bitstring [private].

(* AUTHENTICATION *)

(* App *)
event appSendsLastMessageToIssuer(bitstring, bitstring, bitstring).
event appSendsLastMessageToBrowser(bitstring, bitstring).
event appCompletesProtocol(bitstring, bitstring, bitstring, bitstring, bitstring,
bitstring, bitstring).

(* Browser *)
event browserSendsLastMessageToIssuer(bitstring, bitstring, bitstring).
event browserSendsLastMessageToApp(bitstring, bitstring, bitstring).
event browserCompletesProtocol(bitstring, bitstring, bitstring, bitstring, bitstring,
bitstring).

(* Issuer *)
event issuerSendsLastMessageToApp(bitstring, bitstring, bitstring, bitstring).
event issuerSendsLastMessageToBrowser(bitstring, bitstring, bitstring, bitstring).
event issuerCompletesProtocol(bitstring, bitstring, bitstring, bitstring, bitstring,
bitstring).

(* ------------------------------------------------------------------------- *)
(* End Definitions *)
(* Start Agent Macros *)
(* ------------------------------------------------------------------------- *)

let app(talksOnlyToHonest:bool, c:channel, a:bitstring, ska:SecretKey, b:bitstring,
pkb:PublicKey, i:bitstring, pki:PublicKey) =

    (* !Talking to Browser (Without Handshake) *)

    (* Message 1 *)
    new pkce:bitstring;
    new scope:bitstring;
    let m_1 = (tag1, a, hash(pkce), scope) in
    out(c, m_1);

    (* ... Waiting For Issuer Response While It Talks To The Browser ... *)

    (* !Connecting to Browser *)

    (* Message 6 *)
    in(c, m_6:bitstring);
    let (tag6':bitstring, n:bitstring, spkI:PublicKey) = adec(m_6, ska) in
    if (tag6 = tag6') then

    (* Message 7 *)
    new k1: SymmetricKey;
    let m_7 = aenc((tag7, n, k1), spkI) in
    event appSendsLastMessageToBrowser(m_6, m_7);
    out(c, m_7);

    (* !Talking to Browser *)

    (* Message 8 *)
    in(c, m_8:bitstring);
    let (tag8':bitstring, code:bitstring) = dec(m_8, k1) in
    if (tag8 = tag8') then

    (* !Connecting to Issuer *)

    (* Message 9 *)
    new n2:bitstring;
    new sskI2:SecretKey;
    let m_9 = aenc((tag9, n2, pk(sskI2)), pki) in
    out(c, m_9);
    (* Message 10 *)
    in(c, m_10:bitstring);
    let (tag10':bitstring, n2':bitstring, k3:SymmetricKey) = adec(m_10, sskI2) in
    if (tag10 = tag10') then
    if (n2 = n2') then

    (* !Talking to Issuer*)

    (* Message 11 *)
    new sks:SecretKey;
    new jti:bitstring;
    let T2 = (a, jti, code, pkce) in
    let m_11 = enc((tag11, pk(sks), sign(T2, sks)), k3) in
    event appSendsLastMessageToIssuer(m_9, m_10, m_11);
    out(c, m_11);

    (* Message 12 *)
    in(c, m_12:bitstring);
    let (tag12':bitstring, signed_M:bitstring) = dec(m_12, k3) in
    if (tag12 = tag12') then
    let M = check(signed_M, pki) in
    let (jtk:bitstring, jti':bitstring, scope':bitstring, a':bitstring, i':bitstring,
    b':bitstring) = M in

    if (jtk = hash_pk(pk(sks))) then
    if (jti = jti') then
    if (scope = scope') then
    if (a = a') then
    if (i = i') then
    if (b = b') then

    (* ---------------------------------------------------------- *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_app);

    event appCompletesProtocol(m_6, m_7, m_8, m_9, m_10, m_11, m_12);

    out(ch(code), free_code);

    0.

let browser(talksOnlyToHonest:bool, c:channel, b:bitstring, skb:SecretKey, a:bitstring,
pka:PublicKey, i:bitstring, pki:PublicKey, password:bitstring) =

    (* !Talking to App (Without Handshake) *)

    (* Message 1 *)
    in(c, m_1:bitstring);
    let (tag1':bitstring, a':bitstring, h_pkce:bitstring, scope:bitstring) = m_1 in
    if (tag1 = tag1') then
    if (a = a') then

    (* !Connecting to Issuer *)
    (* Message 2 *)
    new n2:bitstring;
    new sskI:SecretKey;
    let m_2 = aenc((tag2, n2, pk(sskI)), pki) in
    out(c, m_2);

    (* Message 3 *)
    in(c, m_3:bitstring);
    let (tag3':bitstring, n2':bitstring, k2:SymmetricKey) = adec(m_3, sskI) in
    if (tag3 = tag3') then
    if (n2 = n2') then

    (* !Talking to Issuer *)

    (* Message 4 *)
    let m_4 = enc((tag4, a, scope, h_pkce, b, password), k2) in
    event browserSendsLastMessageToIssuer(m_2, m_3, m_4);
    out(c, m_4);

    (* Message 5 *)
    in(c, m_5:bitstring);
    let (tag5':bitstring, code:bitstring) = dec(m_5, k2) in
    if (tag5 = tag5') then

    (* !Connecting to App *)

    (* Message 6 *)
    new n:bitstring;
    new sskI:SecretKey;
    let m_6 = aenc((tag6, n, pk(sskI)), pka) in
    out(c, m_6);

    (* Message 7 *)
    in(c, m_7:bitstring);
    let (tag7':bitstring, n':bitstring, k1:SymmetricKey) = adec(m_7, sskI) in
    if (tag7 = tag7') then
    if (n = n') then

    (* !Talking to App *)

    (* Message 8 *)
    let m_8 = enc((tag8, code), k1) in
    event browserSendsLastMessageToApp(m_6, m_7, m_8);
    out(c, m_8);

    (* ------------------------------------------------------------ *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_browser);

    event browserCompletesProtocol(m_2, m_3, m_4, m_5, m_6, m_7);

    out(ch(code), free_code);
```

```
    out(ch(password), free_password);

    0.

let issuer(talksOnlyToHonest:bool, c:channel, i:bitstring, ski:SecretKey, a:bitstring,
pka:PublicKey, b:bitstring, pkb:PublicKey, password:bitstring) =

    (* !Connecting to Browser *)

    (* Message 2 *)
    in(c, m_2:bitstring);
    let (tag2':bitstring, n:bitstring, spkI:PublicKey) = adec(m_2, ski) in
    if (tag2 = tag2') then

    (* Message 3 *)
    new k2: SymmetricKey;
    let m_3 = aenc((tag3, n, k2), spkI) in
    out(c, m_3);

    (* !Talking to Browser *)

    (* Message 4 *)
    in(c, m_4:bitstring);
    let (tag4':bitstring, a':bitstring, h_pkce:bitstring, scope:bitstring, b':bitstring,
    password':bitstring) = dec(m_4, k2) in
    if (tag4 = tag4') then

    if (a = a') then
    if (b = b') then
    if (password = password') then

    (* Message 5 *)
    new code:bitstring;
    let m_5 = enc((tag5, code), k2) in
    event issuerSendsLastMessageToBrowser(m_2, m_3, m_4, m_5);
    out(c, m_5);

    (* Waiting for Browser to transfer code to App *)

    (* !Connecting to App *)

    (* Message 9 *)
    in(c, m_9:bitstring);
    let (tag9':bitstring, n2:bitstring, spkI2:PublicKey) = adec(m_9, ski) in
    if (tag9 = tag9') then

    (* Message 10 *)
    new k3: SymmetricKey;
    let m_10 = aenc((tag10, n2, k3), spkI2) in
    out(c, m_10);

    (* !Talking to App*)

    (* Message 11 *)
    in(c, m_11:bitstring);
    let (tag11':bitstring, pksks:PublicKey, signature:bitstring) = dec(m_11, k3) in
    if (tag11 = tag11') then
    let (a'':bitstring, jti:bitstring, code':bitstring, pkce:bitstring) =
    check(signature, pksks) in

    if (a = a'') then
    if (code = code') then
    if (h_pkce = hash(pkce)) then

    (* Message 12 *)
    let jtk = hash_pk(pksks) in
    let M = (jtk, jti, scope, a, i, b) in
    let m_12 = enc((tag12, sign(M, ski)), k3) in
    event issuerSendsLastMessageToApp(m_9, m_10, m_11, m_12);
    out(c, m_12);

    (* ------------------------------------------------------------ *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_issuer);

    event issuerCompletesProtocol(m_2, m_3, m_4, m_9, m_10, m_11);

    out(ch(code), free_code);
    out(ch(password), free_password);

    0.

(* ---------------------------------------------------------------------- *)
(* End Agent Macros *)
(* Start Open Network *)
(* ---------------------------------------------------------------------- *)

let open_app(c:channel, a:bitstring, ska:SecretKey, b:bitstring, pkb:PublicKey,
i:bitstring, pki:PublicKey) =
  in(c, e1:bitstring);
  in(c, e2:bitstring);
  if (e1 = b) && (e2 = i) then
    app(true, c, a, ska, b, pkb, i, pki)
  else
  if (e1 <> b) && (e2 = i) then
    in(c, pke1:PublicKey);
    app(false, c, a, ska, e1, pke1, i, pki)
  else
  if (e1 = b) && (e2 <> i) then
    in(c, pke2:PublicKey);
    app(false, c, a, ska, b, pkb, e2, pke2)
  else
  if (e1 <> b) && (e2 <> i) then
    in(c, pke1:PublicKey);
    in(c, pke2:PublicKey);
    app(false, c, a, ska, e1, pke1, e2, pke2)
  .

let open_browser(c:channel, b:bitstring, skb:SecretKey, a:bitstring, pka:PublicKey,
i:bitstring, pki:PublicKey, password:bitstring) =
  in(c, e1:bitstring);
  in(c, e2:bitstring);
  if (e1 = a) && (e2 = i) then
    browser(true, c, b, skb, a, pka, i, pki, password)
  else
  if (e1 <> a) && (e2 = i) then
    in(c, pke1:PublicKey);
    browser(false, c, b, skb, e1, pke1, i, pki, password)
  else
  if (e1 = a) && (e2 <> i) then
    in(c, pke2:PublicKey);
    new wrong_pw:bitstring;
    browser(false, c, b, skb, a, pka, e2, pke2, wrong_pw)
  else
  if (e1 <> a) && (e2 <> i) then
    in(c, pke1:PublicKey);
    in(c, pke2:PublicKey);
    new wrong_pw:bitstring;
    browser(false, c, b, skb, e1, pke1, e2, pke2, wrong_pw)
  .

let open_issuer(c:channel, i:bitstring, ski:SecretKey, a:bitstring, pka:PublicKey,
b:bitstring, pkb:PublicKey, password:bitstring) =
  in(c, e1:bitstring);
  in(c, e2:bitstring);
  if (e1 = a) && (e2 = b) then
    issuer(true, c, i, ski, a, pka, b, pkb, password)
  else
  if (e1 <> a) && (e2 = b) then
    in(c, pke1:PublicKey);
    issuer(false, c, i, ski, e1, pke1, b, pkb, password)
  else
  if (e1 = a) && (e2 <> b) then
    new wrong_pw:bitstring;
    in(c, pke2:PublicKey);
    issuer(false, c, i, ski, a, pka, e2, pke2, wrong_pw)
  else
  if (e1 <> a) && (e2 <> b) then
    new wrong_pw:bitstring;
    in(c, pke1:PublicKey);
    in(c, pke2:PublicKey);
    issuer(false, c, i, ski, e1, pke1, e2, pke2, wrong_pw)
  .

(* ------------------------------------------------------------------------ *)
(* End Open Network *)
(* Start Assumptions *)
(* ------------------------------------------------------------------------ *)

(* SECRECY *)

query attacker(check_reach_app).
query attacker(check_reach_browser).
query attacker(check_reach_issuer).
query attacker(free_code).
query attacker(free_password).

(* AUTHENTICATION *)

(* App Completes Protocol *)
query m_6:bitstring, m_7:bitstring, m_8:bitstring, m_9:bitstring, m_10:bitstring,
m_11:bitstring, m_12:bitstring;
    inj-event(appCompletesProtocol(m_6, m_7, m_8, m_9, m_10, m_11, m_12))
    ==>
    inj-event(browserSendsLastMessageToApp(m_6, m_7, m_8))
    &&
    inj-event(issuerSendsLastMessageToApp(m_9, m_10, m_11, m_12))
    .

(* Browser Completes Protocol *)
query m_2:bitstring, m_3:bitstring, m_4:bitstring, m_5:bitstring, m_6:bitstring,
m_7:bitstring;
    inj-event(browserCompletesProtocol(m_2, m_3, m_4, m_5, m_6, m_7))
    ==>
    inj-event(appSendsLastMessageToBrowser(m_6, m_7))
    &&
    inj-event(issuerSendsLastMessageToBrowser(m_2, m_3, m_4, m_5))
    .

(* Issuer Completes Protocol *)
query m_2:bitstring, m_3:bitstring, m_4:bitstring, m_9:bitstring, m_10:bitstring,
m_11:bitstring;
    inj-event(issuerCompletesProtocol(m_2, m_3, m_4, m_9, m_10, m_11))
    ==>
    inj-event(appSendsLastMessageToIssuer(m_9, m_10, m_11))
    &&
    inj-event(browserSendsLastMessageToIssuer(m_2, m_3, m_4))
    .

(* ------------------------------------------------------------------------ *)
(* End Assumptions *)
(* Start Main *)
(* ------------------------------------------------------------------------ *)

free c:channel.
process (*use root process for setup *)

    (* create identities *)
    new a:bitstring;
    new b:bitstring;
    new i:bitstring;

    (*advertise identities*)
    out(c,a);
    out(c,b);
    out(c,i);

    (*declare secret keys*)
    new ska:SecretKey;
    new skb:SecretKey;
    new ski:SecretKey;

    (*advertise public keys*)
    out(c,pk(ska));
    out(c,pk(skb));
    out(c,pk(ski));

    (*generate variables*)
    new password:bitstring;

    (*let agents dance*)
    (
        ! open_app      (c, a, ska, b, pk(skb), i, pk(ski))
      | ! open_browser  (c, b, skb, a, pk(ska), i, pk(ski), password)
      | ! open_issuer   (c, i, ski, a, pk(ska), b, pk(skb), password)
    )

(* ------------------------------------------------------------------------- *)
(* End Main *)
(* ------------------------------------------------------------------------- *)
```

.4 VC-Based Third Party Authentication Protocol

This model depicts the newly proposed alternative granting third party authentication using verifiable credentials as described in the msc 4.2.

```
(* ------------------------------------------------------------------------- *)
(* VC-Based Third Party Authentication Protocol *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* Start Definitions *)
(* ------------------------------------------------------------------------- *)

(* Do not forget to insert the previously mentioned definitions as explained in appendix 1 *)

(*DEFS*)
fun ch(bitstring): channel.
fun chSymK(SymmetricKey): channel.

(*Const*)
(* const password:bitstring. *)
const attr:bitstring.
const uri:bitstring.
const scope:bitstring.
const solid:bitstring.

(* SECRECY *)

free check_reach_issuer:bitstring [private].
free check_reach_user:bitstring [private].
free check_reach_user_to_pod:bitstring [private].
free check_reach_pod:bitstring [private].

free symmetric_key_IU:bitstring [private].
free symmetric_key_UP:bitstring [private].
```

```
(* AUTHENTICATION *)

(* Issuer *)
event userSendsLastMessageToIssuer(bitstring, bitstring, bitstring, bitstring).
event issuerCompletesProtocol(bitstring, bitstring, bitstring, bitstring).

(* User *)
event issuerSendsLastMessageToUser(bitstring, bitstring, bitstring, bitstring,
bitstring).
event podSendsLastMessageToUser(bitstring, bitstring, bitstring, bitstring,
bitstring, bitstring, bitstring).
event userCompletesProtocol(bitstring, bitstring, bitstring, bitstring,
bitstring).
event userCompletesProtocolFull(bitstring, bitstring, bitstring, bitstring,
bitstring, bitstring, bitstring, bitstring, bitstring, bitstring, bitstring, bitstring).

(* Pod *)
event userSendsLastMessageToPod(bitstring, bitstring, bitstring, bitstring,
bitstring, bitstring).
event podCompletesProtocol(bitstring, bitstring, bitstring, bitstring, bitstring,
bitstring).

(* ------------------------------------------------------------------------ *)
(* End Definitions *)
(* Start Agent Macros *)
(* ------------------------------------------------------------------------ *)

let pod(talksOnlyToHonest:bool, c:channel, p:bitstring, skp:SecretKey, u:bitstring,
pku:PublicKey, i:bitstring, pki:PublicKey) =

    (* !Connecting to User *)

    (* Message 6 *)
    in(c, m_6:bitstring);
    let (tag6':bitstring, bitgx:bitstring) = m_6 in
    if (tag6 = tag6') then
    let gx = unbitG(bitgx) in

    (* Message 7 *)
    new y: exponent;
    let gy = g(y) in
    let k_UP = GtoSymK(exp(gx, y)) in
    let m_7 = (bitG(gy), enc((tag7, sign((gy, gx), skp)), k_UP)) in
    out(c, m_7);

    (* Message 8 *)
    in(c, m_8:bitstring);
    let (tag8':bitstring, signature:bitstring) = dec(m_8, k_UP) in
    if (tag8 = tag8') then
    if (gx, gy) = check(signature, pku) then

    (* !Talking to User *)

    (* Message 9 *)
    in(c, m_9:bitstring);
    let (tag9':bitstring, uri':bitstring) = dec(m_9, k_UP) in
    if (tag9 = tag9') then
    if (uri' = uri) then

    (* Message 10 *)
    new nc:bitstring;
    let rule = (u, i, attr) in
    let m_10 = enc((tag10, rule, nc), k_UP) in
    out(c, m_10);

    (* Message 11 *)
    in(c, m_11:bitstring);
    let (tag11':bitstring, PT:bitstring, nc':bitstring) = dec(m_11, k_UP) in
    if (tag11 = tag11') then
    if (nc = nc') then
    let (VC:bitstring, scope':bitstring, solid':bitstring) = check(PT, pku) in
    let (claims:bitstring, signed_claims:bitstring) = VC in
    let (u':bitstring, i':bitstring, attr':bitstring) = check(signed_claims, pki) in

    if (i = i') then
    if (u = u') then
    if (scope = scope') then
    if (solid = solid') then
    if (attr = attr') then

    (* Message 12 *)
    new accesstoken:bitstring;
    let m_12 = enc((tag12, sign((accesstoken), skp)), k_UP) in
    event podSendsLastMessageToUser(m_6, m_7, m_8, m_9, m_10, m_11, m_12);
    out (c, m_12);

    (* ---------------------------------------------------------- *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_pod);

    event podCompletesProtocol(m_6, m_7, m_8, m_9, m_10, m_11);

    out(chSymK(k_UP), symmetric_key_UP);

    0.

let user_to_pod(talksOnlyToHonest:bool, c:channel, u:bitstring, sku:SecretKey,
pku:PublicKey, i:bitstring, pki:PublicKey, p:bitstring, pkp:PublicKey,
PT:bitstring, m_1:bitstring, m_2:bitstring, m_3:bitstring, m_4:bitstring, m_5:bitstring) =

    (* !Connecting to Pod *)

    (* Message 6 *)
    new x:exponent;
    let gx = g(x) in
    let m_6 = (tag6, bitG(gx)) in
    out(c, m_6);
```

```
(* Message 7 *)
in(c, m_7:bitstring);
let (bitgy:bitstring, encryption:bitstring) = m_7 in
let gy = unbitG(bitgy) in
let k_UP = GtoSymK(exp(gy, x)) in
let (tag7':bitstring, signature:bitstring) = dec(encryption, k_UP) in
if (tag7 = tag7') then
if (gy, gx) = check(signature, pkp) then

(* Message 8 *)
let m_8 = enc((tag8, sign((gx, gy), sku)), k_UP) in
out(c, m_8);

(* !Talking to Pod *)

(* Message 9 *)
let m_9 = enc((tag9, uri), k_UP) in
out(c, m_9);

(* Message 10 *)
in(c, m_10:bitstring);
let (tag10':bitstring, rule':bitstring, nc:bitstring) = dec(m_10, k_UP) in
if (tag10 = tag10') then
let N = check(PT, pku) in
let (VC:bitstring, scope:bitstring, solid:bitstring) = N in
let (claims:bitstring, signed_claims:bitstring) = VC in
let (u':bitstring, i':bitstring, attr':bitstring) = check(signed_claims, pki) in
let (u'':bitstring, i'':bitstring, attr'':bitstring) = rule' in

(* if (rule = claims) then *)
if (u' = u'') then
if (i' = i'') then
if (attr' = attr'') then

(* Message 11 *)
let m_11 = enc((tag11, PT, nc), k_UP) in
event userSendsLastMessageToPod(m_6, m_7, m_8, m_9, m_10, m_11);
out(c, m_11);

(* Message 12 *)
in(c, m_12:bitstring);
let (tag12':bitstring, signed_accesstoken:bitstring) = dec(m_12, k_UP) in
if (tag12 = tag12') then
let (accesstoken:bitstring) = check(signed_accesstoken, pkp) in

(* ---------------------------------------------------------- *)
(* Query Preparations *)

if talksOnlyToHonest then

out(c, check_reach_user_to_pod);

event userCompletesProtocolFull(m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9,
m_10, m_11, m_12);

out(chSymK(k_UP), symmetric_key_UP);
```

```
    0.

let open_user_to_pod(c:channel, u:bitstring, sku:SecretKey, pku:PublicKey, i:bitstring,
pki:PublicKey, p:bitstring, pkp:PublicKey, PT:bitstring, m_1:bitstring, m_2:bitstring,
m_3:bitstring, m_4:bitstring, m_5:bitstring) =
  in(c, e:bitstring);
  if (e = p) then
    user_to_pod(true, c, u, sku, pku, i, pki, p, pkp, PT, m_1, m_2, m_3, m_4, m_5)
  else
    in(c, pke:PublicKey);
    user_to_pod(false, c, u, sku, pku, i, pki, e, pke, PT, m_1, m_2, m_3, m_4, m_5)
  .

let user(talksOnlyToHonest:bool, c:channel, u:bitstring, sku:SecretKey, pku:PublicKey,
i:bitstring, pki:PublicKey, p:bitstring, pkp:PublicKey, password:bitstring) =

    (* !Connecting to Issuer *)

    (* Message 1 *)
    new x:exponent;
    let gx = g(x) in
    let m_1 = (tag1, bitG(gx)) in
    out(c, m_1);

    (* Message 2 *)
    in(c, m_2:bitstring);
    let (bitgy:bitstring, encryption:bitstring) = m_2 in
    let gy = unbitG(bitgy) in
    let k_IU = GtoSymK(exp(gy, x)) in
    let (tag2':bitstring, signature:bitstring) = dec(encryption, k_IU) in
    if (tag2 = tag2') then
    if (gy, gx) = check(signature, pki) then

    (* Message 3 *)
    let m_3 = enc((tag3, sign((gx, gy), sku)), k_IU) in
    out(c, m_3);

    (* !Talking to Issuer *)

    (* Message 4 *)
    let m_4 = enc((tag4, u, attr, password),k_IU) in
    event userSendsLastMessageToIssuer(m_1, m_2, m_3, m_4);
    out(c, m_4);

    (* Message 5 *)
    in(c, m_5:bitstring);
    let (tag5':bitstring, signed_VC:bitstring) = dec(m_5, k_IU) in
    if (tag5 = tag5') then
    let (VC:bitstring, u':bitstring) = check(signed_VC, pki) in
    let (claims:bitstring, signed_claims:bitstring) = VC in
    let (u':bitstring, i':bitstring, attr':bitstring) = check(signed_claims, pki) in
    if (u = u') then
    if (i = i') then
    if (attr = attr') then
    let N = (VC, scope, solid) in
    let PT = sign(N, sku) in

    (* !Communication with App *)

    !open_user_to_pod(c, u, sku, pku, i, pki, p, pkp, PT, m_1, m_2, m_3, m_4, m_5) |

    (* ---------------------------------------------------------- *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_user);

    event userCompletesProtocol(m_1, m_2, m_3, m_4, m_5);

    out(chSymK(k_IU), symmetric_key_IU);

    0.

let issuer(talksOnlyToHonest:bool, c:channel, i:bitstring, ski:SecretKey, pki:PublicKey,
u:bitstring, pku:PublicKey, password:bitstring) =

    (* !Connecting to User *)

    (* Message 1 *)
    in(c, m_1:bitstring);
    let (tag1':bitstring, bitgx:bitstring) = m_1 in
    if (tag1 = tag1') then
    let gx = unbitG(bitgx) in

    (* Message 2 *)
    new y: exponent;
    let gy = g(y) in
    let k_IU = GtoSymK(exp(gx, y)) in
    let m_2 = (bitG(gy), enc((tag2, sign((gy, gx), ski)), k_IU)) in
    out(c, m_2);

    (* Message 3 *)
    in(c, m_3:bitstring);
    let (tag3':bitstring, signature:bitstring) = dec(m_3, k_IU) in
    if (tag3 = tag3') then
    if (gx, gy) = check(signature, pku) then

    (* !Talking to User *)

    (* Message 4 *)
    in (c, m_4:bitstring);
    let (tag4':bitstring, u':bitstring, attr':bitstring, password':bitstring) =
    dec(m_4, k_IU) in

    if (tag4 = tag4') then
    if (u = u') then
    if (attr' = attr) then
    if (password' = password) then
    (* Message 5 *)
    let claims = (u, i, attr) in
    let VC = (claims, sign(claims, ski)) in
    let m_5 = enc((tag5, sign((VC, u), ski)), k_IU) in
    event issuerSendsLastMessageToUser(m_1, m_2, m_3, m_4, m_5);
    out(c, m_5);

    (* ---------------------------------------------------------- *)
    (* Query Preparations *)

    if talksOnlyToHonest then

    out(c, check_reach_issuer);

    event issuerCompletesProtocol(m_1, m_2, m_3, m_4);

    out(chSymK(k_IU), symmetric_key_IU);

    0.


(* ------------------------------------------------------------------------ *)
(* End Agent Macros *)
(* Start Open Network *)
(* ------------------------------------------------------------------------ *)

let open_issuer(c:channel, i:bitstring, ski:SecretKey, pki:PublicKey, u:bitstring,
pku:PublicKey, password:bitstring) =
  in(c, e:bitstring);
  if (e = u) then
    issuer(true, c, i, ski, pki, u, pku, password)
  else
    in(c, pke:PublicKey);
    in(c, wrong_pw:bitstring);
    issuer(false, c, i, ski, pki, e, pke, wrong_pw)
  .

let open_user(c:channel, u:bitstring, sku:SecretKey, pku:PublicKey, i:bitstring,
pki:PublicKey, p:bitstring, pkp:PublicKey, password:bitstring) =
  in(c, e:bitstring);
  if (e = i) then
    user(true, c, u, sku, pku, i, pki, p, pkp, password)
  else
    in(c, pke:PublicKey);
    in(c, wrong_pw:bitstring);
    user(false, c, u, sku, pku, e, pke, p, pkp, wrong_pw)
  .

let open_pod(c:channel, p:bitstring, skp:SecretKey, u:bitstring, pku:PublicKey,
i:bitstring, pki:PublicKey) =
  in(c, e:bitstring);
  if (e = u) then
    pod(true, c, p, skp, u, pku, i, pki)
  else
    in(c, pke:PublicKey);
    pod(false, c, p, skp, e, pke, i, pki)
  .

(* ------------------------------------------------------------------------ *)
(* End Open Network *)
(* Start Assumptions *)
(* ------------------------------------------------------------------------ *)

(* SECRECY *)

query attacker(check_reach_issuer).
query attacker(check_reach_user).
query attacker(check_reach_user_to_pod).
query attacker(check_reach_pod).

query attacker(symmetric_key_IU).
query attacker(symmetric_key_UP).

(* AUTHENTICATION *)

(* Issuer Completes Protocol *)
query m_1:bitstring, m_2:bitstring, m_3:bitstring, m_4:bitstring;
    inj-event(issuerCompletesProtocol(m_1, m_2, m_3, m_4))
    ==>
    inj-event(userSendsLastMessageToIssuer(m_1, m_2, m_3, m_4))
    .

(* User Completes Protocol*)
query m_1:bitstring, m_2:bitstring, m_3:bitstring, m_4:bitstring, m_5:bitstring;
    inj-event(userCompletesProtocol(m_1, m_2, m_3, m_4, m_5))
    ==>
    inj-event(issuerSendsLastMessageToUser(m_1, m_2, m_3, m_4, m_5))
    .

(* User Completes Protocol Full*)
query m_1:bitstring, m_2:bitstring, m_3:bitstring, m_4:bitstring, m_5:bitstring,
m_6:bitstring, m_7:bitstring, m_8:bitstring, m_9:bitstring, m_10:bitstring,
m_11:bitstring, m_12:bitstring;
    event(userCompletesProtocolFull(m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9,
    m_10, m_11, m_12))
    ==>
    event(issuerSendsLastMessageToUser(m_1, m_2, m_3, m_4, m_5))
    &&
    event(podSendsLastMessageToUser(m_6, m_7, m_8, m_9, m_10, m_11, m_12))
    .

(* Pod Completes Protocol *)
query m_6:bitstring, m_7:bitstring, m_8:bitstring, m_9:bitstring, m_10:bitstring,
m_11:bitstring;
    inj-event(podCompletesProtocol(m_6, m_7, m_8, m_9, m_10, m_11))
    ==>
    inj-event(userSendsLastMessageToPod(m_6, m_7, m_8, m_9, m_10, m_11))
    .

(* ------------------------------------------------------------------------ *)
(* End Assumptions *)
(* Start Main *)
(* ------------------------------------------------------------------------ *)

free c:channel.
process (*use root process for setup *)

    (* create identities *)
    new i:bitstring;
    new u:bitstring;
    new p:bitstring;

    (*advertise identities*)
    out(c,i);
    out(c,u);
    out(c,p);

    (*declare secret keys*)
    new ski:SecretKey;
    new sku:SecretKey;
    new skp:SecretKey;

    (*advertise public keys*)
    out(c,pk(ski));
    out(c,pk(sku));
    out(c,pk(skp));

    (*declare secret password*)
    new password:bitstring;

    (*let agents dance*)
    (
        ! open_issuer (c, i, ski, pk(ski), u, pk(sku), password)
      | ! open_user   (c, u, sku, pk(sku), i, pk(ski), p, pk(skp), password)
      | ! open_pod    (c, p, skp, u, pk(sku), i, pk(ski))
    )

(* ------------------------------------------------------------------------ *)
(* End Main *)
(* ------------------------------------------------------------------------ *)
```