ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Formal Analysis of Web Single-Sign On Protocols using Tamarin

Bachelor Thesis

Xenia Hofmeier

01.04.2019

Supervisors: Sven Hammann, Dr. Ralf Sasse
Professor: Prof. Dr. David Basin

Department of Computer Science, ETH Zürich

Abstract

Single Sign-On protocols are widely used in today's web applications. They enable a user to access multiple services by logging in once. OpenID Connect is a protocol for delegated authentication that can be used for Single Sign-On. With OpenID Connect a user can be authenticated to multiple services, called Relying Parties, by using one account at the Identity Provider. OpenID Connect is widely deployed and used.

Formally verifying security protocols is highly desired. Through formal verification, security properties can be proven and attacks can be found. The Tamarin prover is a symbolic protocol verification tool that can be used to formally verify models of protocols. It provides a language to define the protocol model, security properties, and the protocol infrastructure.

In this work, we use Tamarin to create a model of OpenID Connect. We abstract the web environment OpenID Connect is running on by modeling the functionality of the Browser and TLS. We choose security properties for OpenID Connect and prove that they hold for our model if certain security measures are respected. We present attacks that are possible when those security measures are ignored.

Acknowledgements

Firstly, I want to thank my supervisors Ralf Sasse and Sven Hammann for their great support during the last six months. They invested a vast amount of time to guide my project to the desired goal. Ralf and Sven answered all my questions with great expertise. The good collaboration and the positive working atmosphere left a lasting impact.

Furthermore, I am thankful for being able to write my thesis in the group of Prof. Dr. David Basin.

And lastly, I would like to thank everyone who supported me during this time.

Contents

Chapter 1

Introduction

Today, many web services are used that require the authentication of a user. It is quite impractical for a user to have to create an account with login credentials for every service. Single Sign-On (SSO) protocols can overcome this. They enable a user to authenticate to multiple services, called Relying Parties (RPs) by using their account at an Identity Provider (IdP). A widely deployed SSO protocol is OpenID Connect (OIDC). For example Google, Amazon and Microsoft offer services where they act as IdPs for OIDC. OIDC is an authentication layer on top of OAuth 2.0. OAuth 2.0 is a protocol for authorization, but it was mistakenly also used for authentication, which led to security vulnerabilities. OIDC should resolve those problems. The security of such a widely used delegated authentication protocol is important, because many services and users rely on it.

Having a protocol that is proven to be secure is highly desirable. Unfortunately, the problem of proving the security of protocols is undecidable. For some protocols and their security properties, proofs can still be found. Automated formal verification tools can be useful to find such proofs. Such tools can prove security properties based on formal definitions of the protocol, the security properties, and the adversary. The Tamarin prover is such a tool. It can be used to prove that properties hold or to disprove them by finding an attack. This can be done automated or manually. Because of the undecidability of the problem, some proof searches may not terminate. Tamarin provides a powerful language to define a model of the protocol, the protocol infrastructure, the adversary, and the security properties.

We use Tamarin to create a model of OIDC. We abstract the web environment it runs on and we define desired security properties. We use Tamarin to prove or disprove the properties for this model. Our abstraction of the web environment includes the abstraction of Transport Layer Security (TLS) as a channel and the modeling of the browser as an independent role.

1.1 Related Work

Fett, Küsters, and Schmitz [11, 10] formally and manually prove security properties of OIDC. They use their comprehensive model of the web infrastructure, the FKS model, including the communication between the parties over the web and the behavior of the browser. They present a number of possible attacks and according countermeasures. They model the user as part of the browser. The knowledge and the possible actions of the user are included in the browser.

Mladenov, Mainka, and Schwenk [20] analyze OIDC with a focus on attacks possible through the Registration and Discovery. They present attacks and possible countermeasures.

The security of concrete implementations is well studied. For example Mainka et. al. [19] analyze OIDC implementations with an automated tool. They check whether known attacks are possible for those implementations. Li and Mitchell [14] analyze Google's implementation of OIDC and the implementations of the RPs by treating the RPs and IdPs as black boxes and then analyzing the web traffic. They find implementation flaws and provide implementation recommendations. Li, Mitchell, and Chen [15] analyze possible Cross-Site Request Forgery (CSRF) attacks on RP implementations for OIDC and for OAuth 2.0. They discuss possible countermeasures and present a new countermeasure.

OAuth 2.0, the protocol lying below OIDC, is well analyzed. For example Bansal et. al. [2] use the automated tool ProVerif to find attacks and verify security properties of OAuth 2.0. Another tool based analysis is done by Pai et. al. [21]. They analyze OAuth 2.0 using the Alloy Analyzer. OAuth 2.0 is manually proven by Fett, Küsters, and Schmitz [9]. They formally analyze the security properties of OAuth 2.0 with their FKS model. They find attacks and prove that their security properties hold for an enhanced version of OAuth 2.0.

In this work, we analyze OIDC's protocol specification in a formal, symbolic manner, and we do not analyze concrete implementations. In contrast to Fett, Küsters, and Schmitz, we use Tamarin to analyze the protocol in an automated manner. We also put a focus on modeling the user and browser as separated instances. In contrast to Mladenov, Mainka, and Schwenk, we focus on the OIDC Core and we do not analyze the Registration and Discovery.

1.2 Contributions

We used the Tamarin prover to model the OIDC protocol. This is the first automated, symbolic analysis of OIDC. We created three models, one of the OIDC Implicit Flow and two of the OIDC Code Flow, one with and one without client authentication.

We chose security properties that cover important properties from the user's perspective. For example, the user should start the protocol and give consent to the authen-

tication. We formalized the security properties as Tamarin lemmas. The security properties we looked at are limited to the authentication properties and do not consider the authorization properties of OAuth 2.0. The Tamarin prover proved those lemmas for our models. We were able to find two attacks on our security properties. They represent potential security vulnerabilities that can be prevented by adding security parameters that are not given by the OIDC specification [24]. These attacks were described before [11]. Our model only contains the protocol described in the OIDC Core [24]. We focused on the required parts of the protocol to get the security properties of the protocol without adding optional enhancements.

We model the servers and the user as active agent roles in the protocol that are aware of the protocol and keep state of it. The browser is modeled as an agent role that only reacts to the messages received by the other agent roles, unaware of the running protocol. We abstracted the browser and the communication over TLS and assumed them to be secure. This protocol infrastructure is independent of the protocol and could be reused for other web-based protocols. These abstractions and the separation of this protocol infrastructure and the protocol itself was a large part of our work.

1.3 Outline

In Chapter 2, we give an overview of the functionality and syntax of the Tamarin prover. We use a simple protocol example to explain some important features and the functionality of Tamarin's multiset rewriting rules. Next, we describe the OIDC Core protocol. We look at the Implicit Flow and Authentication Code Flow in detail by describing them and providing message sequence charts. We also look at TLS, because OIDC requires its use.

In Chapter 3 we describe our modeling decisions apart from the OIDC protocol. We describe how we handled the identifiers of the roles and the OIDC parameters and secrets. We describe how we modeled the communication over TLS with a channel and how we treated the browser agent.

The OIDC model and our desired security properties are described in Chapter 4. We state the main differences between our Implicit Flow and Authentication Code Flow model. We describe the security features we decided to focus on.

We present two attacks and the possible countermeasures in Chapter 5.

Chapter 2

Background

Today we rely on security protocols for critical applications like data protection, online banking, internet voting, and more. Thus, it is important to prove the security of these protocols. To reason about protocols by hand can be hard and attacks may easily be missed. Formal automated analysis tools can be very helpful and powerful.

For example, the Needham-Schroeder Protocol was developed in 1978. It took 17 years until Gavin Lowe [16, 17] discovered an attack on the Needham-Schroeder Protocol using an automated model checker.

2.1 Tamarin Prover

The Tamarin prover, as documented in [27], is a symbolic modeling and analysis tool for security protocols. With Tamarin, one can formally define a protocol, the capabilities of the adversary, and the desired security properties. Tamarin can then automatically and also manually analyze the security properties for this protocol. However, since the problem of proving security properties is undecidable, Tamarin may not terminate. If Tamarin terminates and the properties hold, Tamarin provides a proof. On the other hand, if Tamarin terminates and the properties do not hold, Tamarin provides a counterexample that represents an attack. To specify the protocol, Tamarin provides a language based on multiset rewriting rules. Tamarin's built-in adversary model is the Dolev-Yao model [7]. In this model, the network is controlled by the adversary.

In Tamarin, protocols are modeled in the form of theories. We provide such a theory in Example 2.1. In the following sections, we give an overview of Tamarin's functionality and syntax by explaining this example.

Example 2.1. *We look at a simple protocol, where a client sends a nonce to a server and the server answers with the signed nonce. Figure 2.1 shows the message sequence chart of our example protocol. This simple protocol can be modeled by the following Tamarin theory.*

```
1  theory Tamarin_example
2  begin
3  /*1. Cryptographic Primitives*/
4  builtins: revealing-signing
5
6  /*2. Multiset Rewriting Rules*/
7  //Public Key Infrastructure
8  rule Register_pk:
9      [ Fr(~ltk) ]
10   -->
11     [ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)) ]
12
13 rule Get_pk:
14     [ !Pk(A, pubkey) ]
15   -->
16     [ Out(pubkey) ]
17
18 rule Reveal_ltk:
19     [ !Ltk(A, ltk) ]
20   --[ LtkReveal(A) ]->
21     [ Out(ltk) ]
22
23 //The Protocol
24 rule Client_sends_nonce:
25     [ Fr(~nonce) ]
26   -->
27     [ Out(<~nonce, $C>), St_Client_1(~nonce, $C) ]
28
29 rule Server_receives_and_signs:
30     [ In(<nonce, $C>), !Ltk($S, ~ltk) ]
31   --[ Send($S, nonce) ]->
32     [ Out(revealSign(nonce, ~ltk)) ]
33
34 rule Client_receives:
35     [ In(revealSign(~nonce, ~ltk)), !Pk($S, pk(~ltk))
36     , St_Client_1(~nonce, $C) ]
37   --[ Finish(), Secret(~nonce)
38       , Authentic($S, ~nonce) ]->
39     [ ]
40
41 /*3. Lemmas: See Section 2.1.3*/
42 end
```

Listing 2.1: A Simple example theory

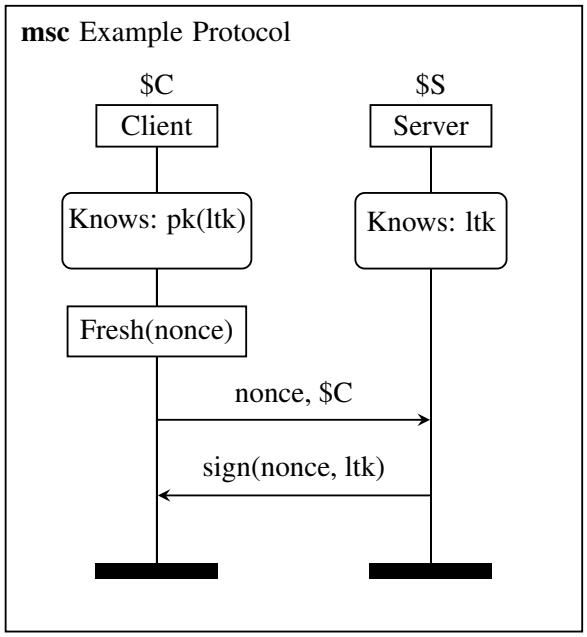

Figure 2.1: Message sequence chart: Simple protocol from Example 2.1

2.1.1 Tamarin Theories

A Tamarin theory begins with the keyword `theory` followed by the name of the theory. In our Example 2.1, the name of the theory is `Tamarin_example`. The keywords `begin` and `end` mark the start and end of the theory. A Tamarin theory contains the following parts that are described later in this section:

1. The cryptographic primitives (lines 3 to 4) contain the used built-in functions and the definitions of new functions.
2. The multiset rewriting rules (lines 6 to 39) define the protocol, the protocol infrastructure and capabilities of the adversary.
3. The lemmas describe the desired properties to be proven. They are provided and explained in Section 2.1.3.

The main ingredients of Tamarin models are:

1. Terms
2. Facts
3. Multiset rewriting rules

In the following, we give a brief overview of those components and we discuss them in more detail afterwards.

In Tamarin, the terms represent messages. They can have the form of variables, constants, and functions applied to terms. For example, `nonce` in line 32 and `revealSign(nonce, ~ltk)` in line 32 are terms.

Facts are composed of a fact symbol `F` and terms `ti`. They have the form `F(t1, ..., tn)`, with a fixed arity. The facts express characteristics of the systems state, for example, what messages were sent, which agent has which private key, and which protocol steps an agent has executed. `Out(revealSign(nonce, ~ltk))` in line 32 is an example of a fact.

A rule begins with the keyword `rule`, followed by a rule name that can be arbitrarily but uniquely chosen, for example, `Server_receives_and_signs`. This rule is stated in lines 29 to 32. A rule consists of a premise and a conclusion, which are multisets of facts. The premise and conclusion are separated by an arrow. Rules can be labeled by action facts. There are user-defined rules as shown in our example, but Tamarin also offers built-in rules, for example for fresh values or for the behavior of the adversary.

The state of Tamarin's transition system is a multiset of facts. The multiset rewriting rules can change this state. A rule can be applied to a state if an instantiation by a substitution σ of the premise of the rule is contained in the state. By applying the rule, this instantiation of the premise is removed from the state and replaced by the instantiation by the substitution σ of the conclusion. The initial state of the transition system is the empty multiset.

Here we give a brief overview of the multiset rewriting rules in Example 2.1. The first three rules, `Register_pk`, `Get_pk`, and `Reveal_ltk`, model the public key infrastructure. They are general, such that every agent but the adversary can register its secret and public key. `ltk` stands for long term key and is a synonym for secret key. The three protocol rules `Client_sends_nonce`, `Server_receives_and_signs` and `Client_receives` define the protocol itself. Each rule describes actions of one role. We have the client and the server role. The rules `Client_sends_nonce` and `Client_receives` model the behavior of the client and rule `Server_receives_and_signs` models the server's behavior.

Terms

In Tamarin, a message is represented as a term. This can be a constant `'c'`, a variable `v` or a message `f(m1,...,mn)` where `f` is an n-ary function symbol and `mi` is a message. Tamarin distinguishes four sorts of variables. They are marked with different prefixes:

- `~x`: A fresh variable used for random numbers like nonces or keys, for example the `~nonce` in line 25. They are generated by the built-in fact `Fr`.
- `$x`: A public variable used, for example, for the identities of the agents. In line 30, the `$C` stands for the client and the `$S` stands for the server.

- `#x`: A temporal variable used for time points of traces. Temporal variables are only used in the lemmas, see Section 2.1.3.
- `x`: Variables that are a superset of the variables `~x` and `$x`. They can be matched to both `~x` and `$x`, but not the other way. For example, the `ltk` in line 19.

There are also string constants `'c'` that are fixed, global, public constants.

Tamarin supports some built-in functions for hashing, signing, encryption and more. The properties of the functions are provided by message theories. To use the built-in theories `f1, ..., fn` in a Tamarin theory, one has to add the line `builtins: f1, ..., fn`. In Example 2.1 we use the built-in theory `revealing-signing`, see line 4. It is used for a message-revealing signature scheme. This theory provides the functions `revealSign`, `revealVerify`, `getMessage`, and `true`. We used the function `revealSign` in the lines 32 and 35. The following equations, taken from [27], define the functions of `revealing-signing`:

```
revealVerify(revealSign(m,sk),m,pk(sk)) = true
getMessage(revealSign(m,sk)) = m
```

Users can also define their own functions by defining such equations that must satisfy the finite variant property, as described in [8].

The functions `pair`, `fst`, and `snd` do not need to be included as they are available automatically. The function `pair` represents the pairing of two messages. The following equations, taken from [27], show the properties of those three functions:

```
fst(pair(x,y)) = x
snd(pair(x,y)) = y
```

As syntactic sugar one can also write `<x,y>` instead of `pair(x,y)` and `<x1,x2,...,xn-1,xn>` instead of `<x1,<x2,...,<xn-1,xn>...>>`. In our Example 2.1 we used the pair function in line 27 where the client sends out the nonce and its public name. Because `Out` has arity one, the two messages are paired to one message `<~nonce, $C>`.

Facts

As mentioned above the state of the system is defined through a multiset of facts. A fact describes a characteristic of the state. For example, in Example 2.1, we have the fact `!Ltk($A, ~ltk)`. This fact describes that the agent `$A` has `~ltk` as its long term key. Note that `$A` is a public name and `~ltk` is a fresh value.

The fact to generate fresh values is the built-in fact `Fr`, see line 25 in Example 2.1. It can only appear in the premise of a rule and has arity one. There is a built-in rule that generates instances of the fact `Fr`. This rule ensures that the fresh values are different from all other terms.

There are two facts to send and receive messages to and from the untrusted network that is controlled by the Dolev-Yao adversary. The fact `In`, as in line 30 of Example 2.1, can only appear in the premise of a rule and is used for receiving messages. The fact `Out`, as in line 27, can only appear in the conclusion of a rule and is used for sending messages. They both have arity one.

In our example protocol, we want that the client only accepts the signed nonce if it matches the nonce it previously sent to the server. We can model this property of the protocol by giving the client a state. To prevent confusion with the system state we will refer to the states of single protocol agents as state facts. In our example, `St_Client_1` is a state fact. It gets introduced in rule `Client_sends_nonce` in line 27, and it occurs in the premise of the rule `Client_receives` in line 36. Thus, rule `Client_receives` can only be applied if rule `Client_sends_nonce` was previously applied, and the received nonce is the same as in `St_Client_1`.

Tamarin distinguishes linear and persistent facts. Linear facts are consumed by a rule. This means if a rule has a linear fact in its premise and this fact is not in its conclusion, after applying the rule, the instantiation of the fact will be removed from the current system state. In contrast, persistent facts are not removed from the state by applying rules. Persistent facts are marked with `!`.

An example of a linear fact is the fact `St_Client_1`. After being consumed in rule `Client_receives` in line 34 the fact cannot be used a second time. For our protocol, this means the client only expects to receive a signed nonce if it has previously sent a nonce to the server. Once the client received a signed nonce, it will not expect another one, until it sends another nonce to the server.

This stands in contrast to the behavior of the persistent fact `!Ltk`. Once created in rule `Register_pk` in line 8, it is not consumed by any rule and it stays in the systems state. Thus, it can be used in the same protocol execution multiple times. This makes sense for a private key, because a key can be used multiple times.

Multiset Rewriting Rules

A Multiset rewriting rule consists of two multisets of facts, the premise and the conclusion. They are separated by an arrow. For example, we look at rule `Register_pk` in line 8. In the first square brackets, we have the premise fact `Fr(~ltk)` and in the second square brackets, we have the conclusion facts `!Ltk($A, ~ltk)` and `!Pk($A, pk(~ltk))`. The rules can also be labeled as seen in rule `Reveal_ltk` in line 18. The arrow contains square brackets with an action fact, in that case `LtkReveal(A)`. The action facts are used by the lemmas to reason about the trace properties, see 2.1.3. The sets of premise, conclusion and action facts can be empty or can contain one or multiple facts. Writing the arrow without square brackets as `-->` is a short form for empty brackets `--[]->`.

Through the application of a rule, an instantiation by a substitution σ of the premise is removed and the instantiation by the substitution σ of the conclusion is added to the state. For simplicity, we will say in the following that facts are removed and added to the state although instantiations of the facts are added and removed.

In the following, we give an overview of the functionality of the multiset rewriting rules by explaining our three protocol rules in Example 2.1. We will describe how the state is changed by applying rules.

The rule `Client_sends_nonce` has the fact `Fr(~nonce)` in its premise. Thus, first, the built-in rule for creating instances of the fact `Fr` is applied to the initially empty state. The new state contains the fact `Fr(~nonce)`. On this state the rule `Client_sends_nonce` can be applied. This causes the fact `Fr(~nonce)` to be removed and replaced by the facts `Out(<~nonce, $C>)` and `St_Client_1(~nonce, $C)`.

The built-in adversary rules translate the `Out` fact to a `In` fact. Through the rule `Register_pk` the facts `!Ltk($S, ~ltk)`[1] and `!Pk($S, pk(~ltk))` can be created. Thus, the facts `In(<~nonce, $C>)` and `!Ltk($S, ~ltk)` are in the state and rule `Server_receives_and_signs` can be applied. With this rule application, the linear `In` fact is removed from the current state and replaced by `Out(revealSign(~nonce, ~ltk))`[2].

This `Out` fact is again translated into an `In` fact. As mentioned above, the fact `!Pk($S, pk(~ltk))` and the state fact `St_Client_1(~nonce, $C)`, as well as the fact `In(revealSign(~nonce, ~ltk))` are in the current state. Thus, rule `Client_receives` can be applied. With that the facts `In(revealSign(~nonce, ~ltk))` and `St_Client_1(~nonce, $C)` are removed from the state and the state only contains the persistent facts `!Ltk($S, ~ltk)` and `!Pk($S, pk(~ltk))`.

The application of the rules uses pattern matching. A rule can only be applied if the facts in the state can be matched with the facts in the premise of the rule. To illustrate this we look again at rule `Client_receives`. The client only accepts the message `revealSign(~nonce, ~ltk)` if there is a state `St_Client_1(~nonce, $C)` with the same nonce, and if there is a fact `!Pk($S, pk(~ltk))` in the state with the matching long term key. Thus, the message is accepted if the client sent the same nonce earlier and if there is a server with the long term key with which

[1] The term `$A` in rule `Register_pk` is instantiated by the term `$S`.

[2] The rule `Server_receives_and_signs` uses the term `nonce` without the `~` symbol. That is, the rule can be executed for any received term, not only fresh values. The server can accept a fresh `~nonce`, because the variable `nonce` can be matched with the fresh variable `~nonce`. In the rule `Client_receives`, we have the fact `St_Client_1(~nonce, $C)` that contains the fresh `~nonce`, because the nonce was freshly created when this fact was created. Thus, the `In` fact in rule `Client_receives` also has to contain the signed fresh `~nonce`.

the message is signed. When an agent has to check signatures or compare values to accept a message, this can be modeled implicitly with this pattern matching.

2.1.2 Adversary

The built-in adversary in Tamarin is a Dolev-Yao adversary [7]. The adversary rules described below are always available, they do not need to be included by the keyword `builtins`.

The Dolev-Yao model assumes a strong adversary that controls the network, but that cannot break cryptography. Every message sent to the network can be read, altered and blocked by the adversary. The adversary can also send messages and alter the source of the message. The network can be seen as the adversary. The agents send and receive messages to and from the adversary.

In Tamarin, this is modeled with the rules for `In` and `Out` facts. There is a set of built-in adversary rules that specify the behavior of the adversary. The adversary can generate messages by combining and applying formulas to the messages contained in its knowledge. The adversary's knowledge is represented by the fact `K(t)`[3]. The terms `t` are in the knowledge of the adversary. The following two rules adapted from [27] describe how the adversary sends and receives messages:

```
1 rule irecv:
2     [ Out( x ) ] --> [ !K( x ) ]
3
4 rule isend:
5     [ !K( x ) ] --[ K( x ) ]-> [ In( x ) ]
```

Listing 2.2: Built-in rules `irecv` and `isend`

With the rule `irecv`, the adversary learns all the messages sent to the network, and with the rule `isend`, the adversary can send messages contained in the adversary's knowledge. The action fact `K(x)` in rule `isend` can be used in the lemmas to express that `x` is in the knowledge of the adversary, see Section 2.1.3.

The adversary can also perform operations on its knowledge. The possible operations are defined by the adversary's built-in message construction and deconstruction rules. For example, the adversary can construct pairs, and also deconstruct them. The following rules adapted from [27] demonstrate this ability.

```
1 rule (modulo AC) c_pair:
2     [ !K( x ), !K( x.1 ) ]
3   --[ !K( <x, x.1> ) ]->
4     [ !K( <x, x.1> ) ]
```

[3] In fact, Tamarin has three facts for the adversary's knowledge, namely `K`, `KU`, and `KD`. This division is made to prevent loops in the proof searches. See [25] for details. For simplicity, we treat the three facts as one. In the following, we replaced `KU` and `KD` by `K`.

```
5
6  rule (modulo AC) d_0_fst:
7      [ !K( <x.1, x.2> ) ] --> [ !K( x.1 ) ]
8
9  rule (modulo AC) d_0_snd:
10     [ !K( <x.1, x.2> ) ] --> [ !K( x.2 ) ]
```

Listing 2.3: Three examples of message construction and deconstruction rules

The abilities of the adversary can be increased and decreased. In our example, we gave the adversary the ability to learn the long term key of the server, with rule `Reveal_ltk` in line 18. The abilities of the adversary can, for example, be decreased by defining channels, see Section 2.1.5.

2.1.3 Lemmas

Lemmas describe the desired properties of the protocol. Tamarin uses trace properties to reason about security properties. A trace is a sequence of action facts. The trace of a protocol execution records the action facts from the rules that are applied during that execution. A trace property describes a set of traces. A trace property holds for a protocol, if for all possible traces produced by the protocol, the property holds. We define a set of traces in Tamarin using first-order logic formulas.

To describe how lemmas in Tamarin are defined, we add the following three lemmas to Example 2.1 and we will describe them in the following.

```
1  //The following lemma holds
2  lemma executable:
3    exists-trace
4      " Ex #i. Finish()@i
5        & not (Ex b #k. LtkReveal(b)@k) "
6
7  //The following lemma does not hold
8  lemma nonce_secret:
9      " All x #i.
10       Secret(x)@i ==> not (Ex #j. K(x)@j)"
11
12 //The following lemma holds
13 lemma message_authentication:
14     "All b m #i. (Authentic(b,m)@i
15         & not (Ex #k. LtkReveal(b)@k))
16       ==> (Ex #j. Send(b,m)@j & j<i)"
```

Listing 2.4: Lemmas for Example 2.1

A lemma begins with the keyword `lemma`, followed by the lemma's name. The first-order logic formula that describes the desired trace property is written in between

two ". Note that the variables with the prefix # are temporal variables that refer to a time point of a trace. `F()@i` states that the action fact `F()` occurs at the time point `i`. The prefix # of variables after the symbol @ can be omitted.

The lemma `executable` in line 2 is a trace existence lemma. Trace existence lemmas are used for sanity checks. They show whether a protocol is executable without adversary interference. One can then reason if the found trace is a normal, expected trace of the protocol. The lemma `executable` describes that there exists a lemma where the formula

```
" Ex #i.  Finish()@i & not (Ex #k.  LtkReveal(b)@k) "
```

holds. `Finish()` references the action fact in line 37 of Example 2.1. `LtkReveal` references the action fact in line 20. The formula tells us that the desired trace should contain the action fact `Finish()` at a time point `#i`, and that the trace should not contain the action fact `LtkReveal` at any time point. That is, the lemma states that there is a trace in which the protocol terminates without any agent revealing its long term key to the adversary.

The other two lemmas are not trace existence lemmas. They describe, in contrast to the trace existence lemmas, security properties that must hold for all possible traces produced by the protocol.

The lemma `nonce_secret` describes a secrecy property. The action fact `Secret(x)` is a secrecy claim. It states that the client expects `x` to be secret at the point of receiving the nonce `x`. The action fact `K(x)` states that `x` is in the knowledge of the adversary. The lemma states that the adversary cannot learn the nonce. This does not hold, because by sending out the nonce in rule `Client_sends_nonce` in line 27 the adversary can learn the nonce.

The lemma `message_authentication` states that if the client receives a message `m` signed by agent `b`, and `b` has not revealed its long term key, `b` must have sent message `m` earlier. This lemma describes that the client and the agent `b` agree on message `m`.

2.1.4 Proofs

Tamarin tries to prove a lemma by negating the lemma and trying to find a trace that satisfies the negated lemma. If Tamarin terminates, it can find such a trace or it can show that there exists no such trace. If Tamarin finds a trace, the lemma is disproven. The found trace is a counterexample that represents an attack on the security property expressed by the lemma. If Tamarin shows that there exists no trace for the negated lemma, Tamarin proved that the lemma holds.

For existence lemmas, Tamarin does not negate the lemma but searches for a trace that satisfies the lemma. A found trace proves the existence lemma, and a proof that there is no trace disproves the existence lemma.

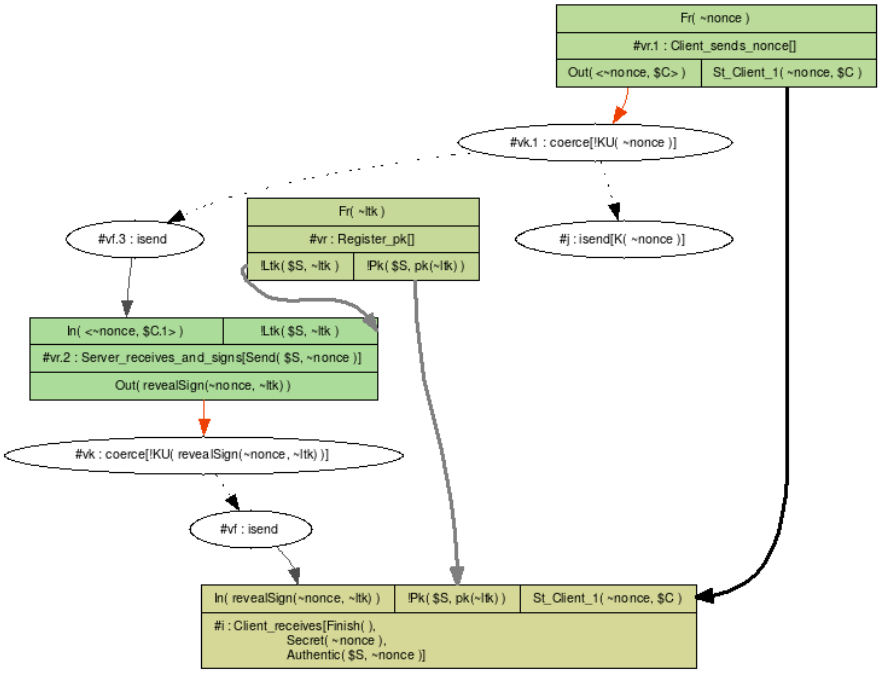

Figure 2.2: Existence trace for Example 2.1 generated by the Tamarin prover

As described above, the lemma `executable` states that the last rule of the protocol can be applied, i.e. there exists a valid execution of the protocol. Tamarin provides the proof for this lemma by providing a trace in form of a graph, shown in Figure 2.2[4].

Each green rectangle represents the application of a rule. The top row of the rectangles contains the premise facts, the middle row contains the time stamp, the name of the rule, and the action facts, and the bottom row contains the conclusion facts. When the conclusion of a rule is empty, the bottom row is missing. The facts generated by one rule and then consumed by another rule are connected by arrows. The linear facts are connected with black arrows and the persistent facts are connected with gray arrows. The ovals represent the network, respectively the adversary. The `Out` and `In` facts are thus connected by arrows and ovals.

This trace meets the lemma `executable`, because the last rectangle contains the action fact `Finish()`, which is required by the lemma. By studying the graph, we can reason if we modeled the protocol as intended. The rules `Client_sends_nonce`, `Server_receives_and_signs`, and `Client_receives` are all executed in the desired order. Thus, we can assume that there is no major error in the protocol rules.

[4] The graph illustrates the trace for lemma `executable` and the counterexample for lemma `nonce_secret`. The graph was generated by Tamarin as a counterexample for lemma `nonce_secret`. In the graph Tamarin generated for lemma `executable`, the oval `#j: isend[K(~nonce)]` is missing.

The counterexample for lemma `nonce_secret` is also shown in Figure 2.2. The last rectangle contains the action fact `Secret(~nonce)` and, as shown in the oval `#j`, the nonce is in the knowledge of the adversary (`K(~nonce)`), which contradicts the lemma.

Tamarin tries to find proofs or counterexamples for the correctness of protocols for an unbounded number of role instances and fresh values. However, the security of protocols is an undecidable problem. Thus, Tamarin may not terminate while proving a lemma. The autoprove function of Tamarin searches the proofs with a heuristics. If this does not terminate in a timely manner, the user can also use Tamarin's interactive mode that enables the user to manually find a proof.

The search space for the traces can also be limited by defining restrictions. A restriction defines a property that must hold for all traces. Through restrictions, traces that we are not interested in are ignored by Tamarin. The lemmas are then proven under the condition of the restrictions, and attacks, not following those conditions, are not found. Thus, the restrictions have to be chosen carefully.

2.1.5 Channels

As stated above, the built-in adversary of Tamarin is a Dolev-Yao adversary. We may want to limit the adversary's control over the network by modeling more or less secure channels. We can do this by specifying channel rules. Authentic channels guarantee the authenticity of the messages sent over the channel. Authenticity describes that the messages sent originate from the stated sender. Confidential channels guarantee the confidentiality of the messages sent over the channel. Confidentiality describes that the messages sent only reach the intended receiver. Secure channels are both confidential and authentic. We will shortly discuss secure and confidential channels.

An adversary can neither modify nor learn messages that are sent over a secure channel. Replay attacks are still possible. The rules for a secure channel can look as follows, taken from [27]:

```
1 rule ChanOut_S:
2     [ Out_S($A,$B,x) ]
3   --[ ChanOut_S($A,$B,x) ]->
4     [ !Sec($A,$B,x) ]
5
6 rule ChanIn_S:
7     [ !Sec($A,$B,x) ]
8   --[ ChanIn_S($A,$B,x) ]->
9     [ In_S($A,$B,x) ]
```

Listing 2.5: Secure channel rules

To use this channel in a protocol, the facts `Out` and `In` are replaced by `Out_S` and `In_S`. For example, if we used secure channels in Example 2.1, the rule `Server_receives_and_signs` would look as follows:

```
rule Server_receives_and_signs_with_secure_channels:
    [ In_S($C, $S, <nonce, $C>), !Ltk($S, ~ltk) ]
  --[ Send($S, nonce) ]->
    [ Out_S($S, $C, revealSign(nonce, ~ltk)) ]
```

Listing 2.6: Example for sending and receiving with secure channels

The first two arguments of the `In_S` and `Out_S` facts state the sender and receiver of the message. With the rules `ChanOut_S` and `ChanIn_S`, the fact `Out_S` is translated into `In_S`. Replay attacks are enabled by the persistent fact `!Sec`. However, the adversary has no other way to send or receive messages over the secure channel.

The model of a secure channel can be simplified by only using the persistent fact `!Sec` as `In` and `Out` fact. The reason to use the translation of the rules `ChanOut_S` and `ChanIn_S` is to have a clear structure. Using the channel rules or using one persistent fact `!Sec`, are in terms of functionality identical.

Next, we consider confidential channels. A confidential channel guarantees that a message can only be received by the intended receiver. Furthermore, anyone can send messages to that channel. We can model a confidential channel as follows, as taken from [27]:

```
rule ChanOut_C:
    [ Out_C($A,$B,x) ]
  --[ ChanOut_C($A,$B,x) ]->
    [ !Conf($B,x) ]

rule ChanIn_C:
    [ !Conf($B,x), In($A) ]
  --[ ChanIn_C($A,$B,x) ]->
    [ In_C($A,$B,x) ]

rule ChanIn_CAdv:
    [ In(<$A,$B,x>) ]
  -->
    [ In_C($A,$B,x) ]
```

Listing 2.7: Confidential channel rules

Analogous to the secure channel, to send messages over our confidential channel, the facts `Out` and `In` are replaced by the facts `Out_C` and `In_C`. The rules `ChanOut_C` and `ChanIn_C` are very similar to the ones of the secure channel. The only difference is that the sender `$A` is left out in the fact `!Conf`. This is because a confidential channel has no guarantees about the source of the message. In rule `ChanIn_C`, the

adversary can provide the source of the message with the fact `In`. The fact `!Conf` is again persistent to enable replay attacks. Additionally, the adversary can send messages to the channel with the rule `ChanIn_CAdv`.

Note that we can model replay protected channels by replacing the persistent facts `!Sec` and `!Conf` by linear facts.

Such channels can provide stronger security guarantees. For example, the lemma `nonce_secret` that does not hold for Example 2.1, would hold if confidential or secure channels were used.

2.1.6 Public Key Infrastructure

We now describe the public key infrastructure used in Example 2.1. The long term key or secret key `~ltk` is represented as a fresh value generated in rule `Register_pk` in line 8. This rule generates the facts `!Ltk` and `!Pk`. The fact `!Ltk($A, ~ltk)` is used by the agent `$A` who owns the key pair. The key owner can use this fact to sign or decrypt messages. The fact `!Pk($A, pk(~ltk))` tells all the agents the public key of agent `$A`. The built-in function `pk` gets the public key of a long term key. The rule `Get_pk` in line 13 sends out the public key. With that, the adversary can learn the public key of an agent.

With the rule `Reveal_ltk` in line 18, the secret key of an agent gets revealed. A revealed secret key corresponds to an agent that was compromised. We call agents that are not compromised honest. Most of the time lemmas are phrased for honest agents. The security property must hold for a group of honest agents, but in the presence of compromised agents. For example, lemma `message_authentication` requires that the server who signed the nonce did not reveal its long term key. The lemma permits other servers to reveal their keys. When using channels we can also define stronger abilities for agents that are compromised. For example, a compromised agent could send out all the messages it receives over a channel.

2.2 OpenID Connect

OpenID Connect (OIDC) is a widely used delegated authentication protocol, specified by [24]. An Identity Provider (IdP), also called authentication server, can authenticate a user to a Relying Party (RP). This enables the user to log in to a RP by using her account at the IdP. The IdP provides an ID Token to the RP that contains a unique identifier for the user and some attributes of the user such as name or age. An overview of the OIDC protocol is given in Figure 2.3.

OIDC can be used for Single Sign-On (SSO). SSO protocols enable users to access multiple services by logging in once, as described by [3]. This saves time and is practical. Users do not need to create an account and provide data to the RP. It simplifies the password management for users because one single account for the IdP can be used to log in to multiple RPs.

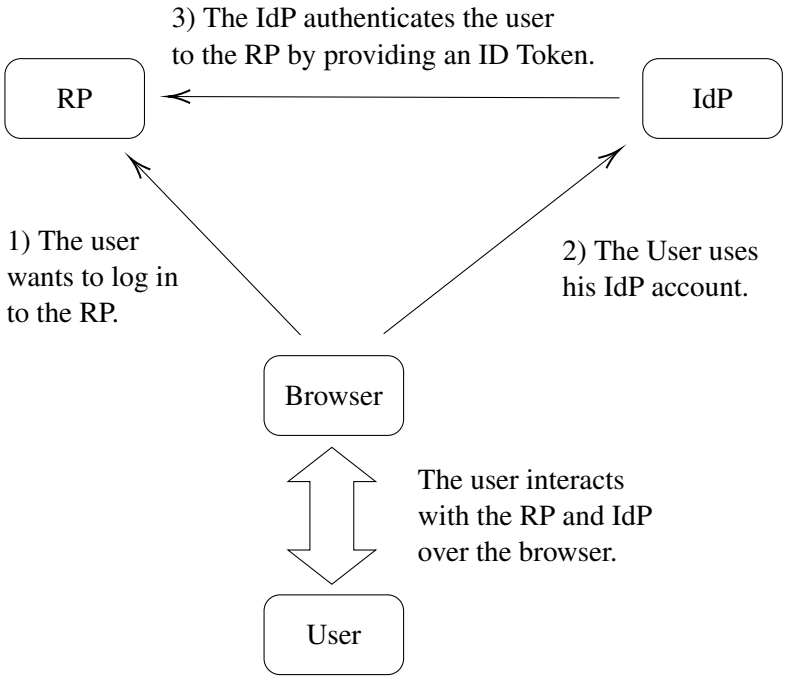

Figure 2.3: Overview of OpenID Connect

For example, Google offers OIDC via Google Sign-In [1], where Google acts as the IdP. To illustrate how OIDC is used, we provide the following example.

Example 2.2. *A user wants to log in to a website, for example, digitec.ch. In this case, digitec.ch acts as the RP. The user clicks on the* ***sign in with Google*** *button and the browser gets redirected to a Google login page. The user authenticates to Google by providing username and password. The user gives consent that Google provides some profile information as the user's name or e-mail address to digitec.ch. Then the user's browser gets redirected back to digitec.ch. The user is now logged in at digitec.ch with the profile information provided by Google.*

OIDC is built on top of OAuth 2.0. OAuth 2.0 is used to authorize access to resources as e-mail or cloud storage, but it does not provide authentication. OAuth 2.0 was often misused for authentication, which resulted in security vulnerabilities [23]. OIDC solved this problem by providing a standard for authentication on top of OAuth 2.0.

In the OIDC [24] and OAuth 2.0 [12] specifications, the RP is often referred to as the client. To avoid confusion between the user and the client we will use the term RP for the client. The term client-ID will still be used for the RP's identifier.

The user interacts with the RP and IdP via his browser. We distinguish between actions that the user takes consciously and automatic redirects of the browser.

The ID Token is a JavaScript Object Notation (JSON) Web Token (JWT) [18]. The ID Token must be signed by the IdP. It contains, among others, the following attributes:

- **Issuer:** The identifier of the IdP.
- **Subject:** An Identifier for the user, chosen by the IdP to uniquely identify the user on the IdP side. We will call it user-ID. The user-ID and the IdP identifier uniquely identify the user.
- **Audience:** The client-ID of the RP.
- **Nonce:** The nonce sent by the RP in the Authentication Request. The nonce is only required for the Implicit Flow.

The ID Token can contain more attributes, especially profile information of the user, but for simplicity, we omit them here.

OIDC consists of the Discovery, the Registration, and the Core. In the Discovery and Registration, the IdP and RP exchange parameters and secrets later used in the Core. The Core describes the actual authentication. In this thesis, we focus on the Core.

There are three possible flows for the authentication:

1. Implicit Flow
2. Authorization Code Flow (We will refer to it as Code Flow)
3. Hybrid Flow

In this thesis, we look at the first two flows. The flows differ in the manner of how the ID Token is passed to the RP. In the Implicit Flow, the ID Token is passed directly by the user's browser, while in the Code Flow, the IdP sends the ID Token to the RP via a secure back channel. In the Code Flow, the RP can be authenticated using a client secret over the back channel. In the Implicit Flow, this is not possible.

Without authenticating the RP, the Code Flow does not offer more security guarantees than the Implicit Flow[5]. In general, the Implicit Flow is simpler and faster. Thus, if the RP cannot hold a client secret, for example, if the RP is a frontend only application, it is reasonable to use the Implicit Flow.

Before the authentication happens, the RP and IdP exchange among others the following parameters and secrets through the Discovery and Registration:

- The **authorization endpoint (authEP)** is the IdP's URI where the user authenticates with its account to the IdP.
- The **token endpoint (tokenEP)** is the IdP's URI where the RP gets the ID Token from the IdP. It is only used in the Code Flow.

[5] This only holds if we use OIDC purely for authentication and not for the authorization provided by OAuth 2.0. OAuth 2.0 uses an Access Token for authorization. The secrecy of this Access Token is more critical than the secrecy of the ID Token.

- The **client-ID** identifies the RP to the IdP.
- The RP's **redirect-uri** is the URI to which the user's browser is redirected after the user has given consent to the IdP.
- The **client secret** may optionally be used in the Code Flow to authenticate the RP to the IdP.

The User and the IdP exchange some account credentials like username and password before OIDC is used. call those credentials that are not defined by the OIDC Core specification [24] user-ID and user secret.

2.2.1 Implicit Flow

In the following, we describe the authentication with the Implicit Flow in a simplified manner. The Implicit Flow is pictured in the message sequence chart in Figure 2.4. The following numbers refer to the messages in that chart.

(1,2) The protocol is started by the user, who chooses on the RP's website to log in with the IdP. In most cases, the user has to click on a button, but it is not specified by the OIDC Core [24] how the user starts the session. The user's browser reacts to the user's action by sending a message to the RP.

(3,4) The RP responds with the Authentication Request that redirects the user's browser to the authEP of the IdP with the arguments client-ID, redirect-uri, and a fresh nonce, that is later included in the ID Token. It is recommended to also send a state, but it is not required.

(5, 6, 7, 8) The user has to authenticate to the IdP. This authentication is not part of OIDC, and the manner of the authentication is not specified by the OIDC specification and can be chosen by the IdP. For our model, we assume that the user authenticates with a user-ID, user secret pair.

(9, 10, 11, 12) The user also has to give consent on the account information, in the chart called user-data, which the IdP will provide to the RP. The manner of giving this consent is also left to the IdP.

(13, 14) After the user has successfully authenticated and given consent, the IdP sends an Authentication Response to the user's browser. The Authentication Response redirects the browser to the redirect-uri of the RP with the signed ID Token and optionally the state.

The user is now logged in at the RP with the profile information provided in the ID Token.

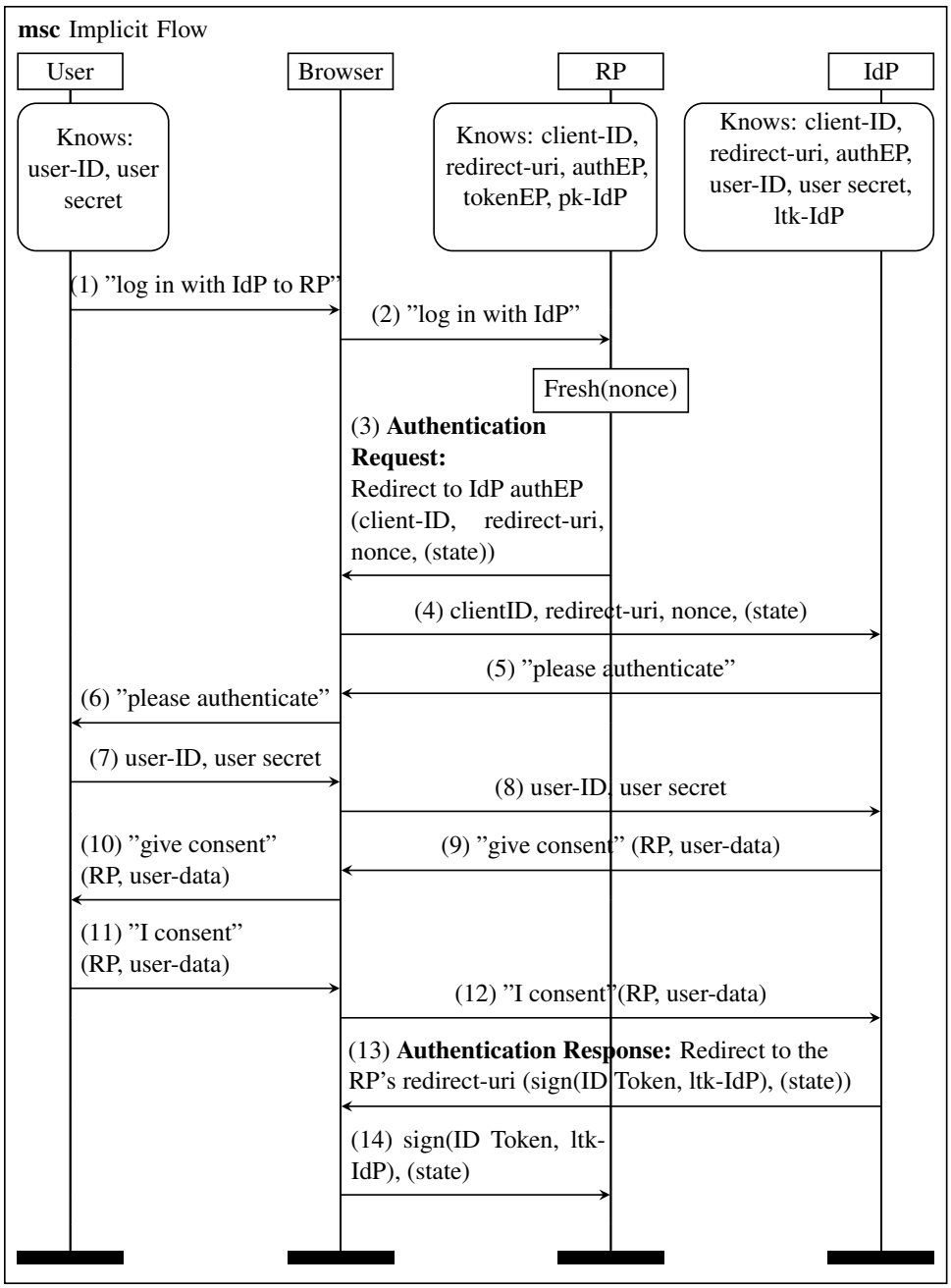

Figure 2.4: Message Sequence chart: OpenID Connect's Implicit Flow, adapted from [10]

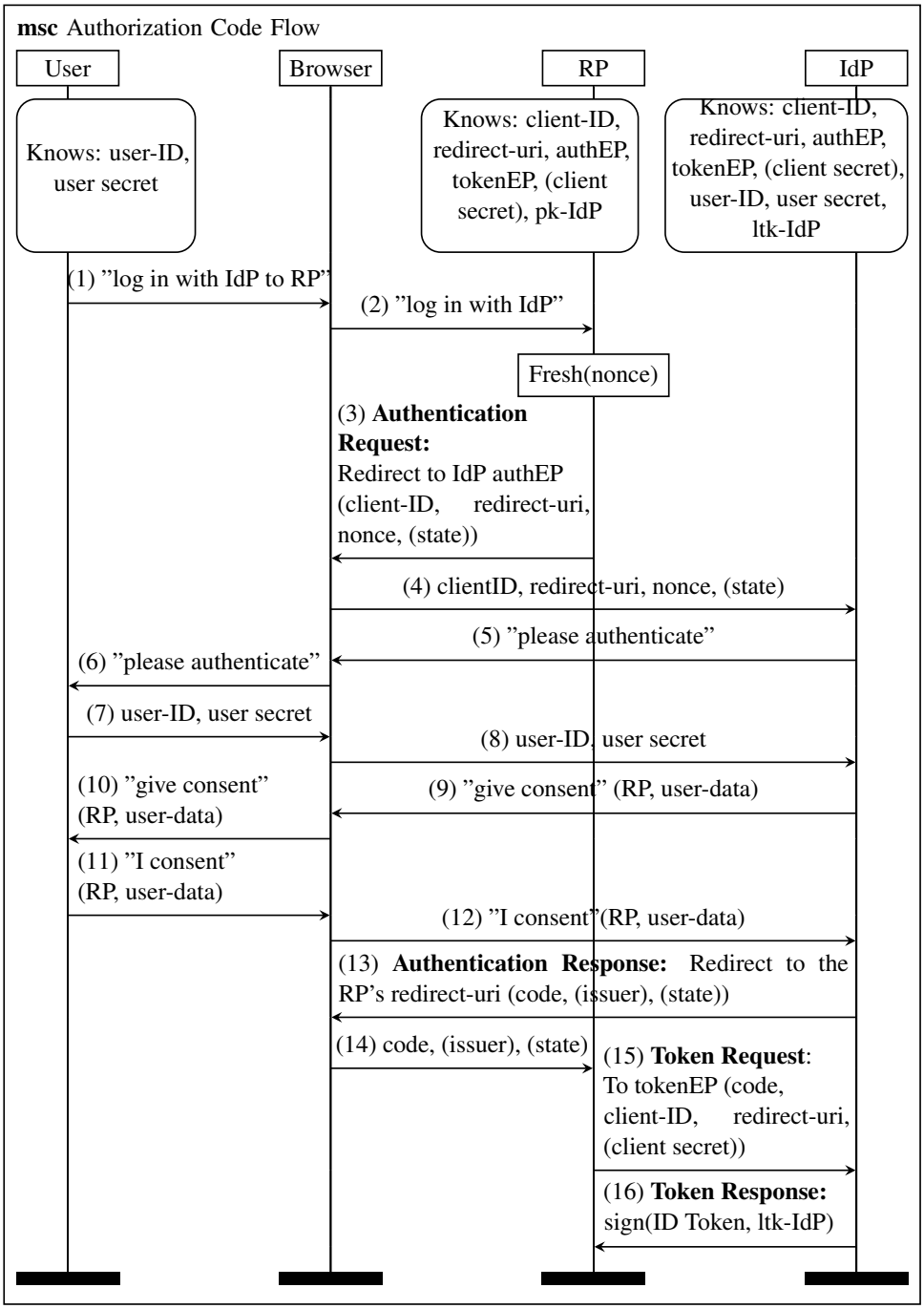

Figure 2.5: Message Sequence chart: OpenID Connect’s Authorization Code Flow, adapted from [11]

2.2.2 Authorization Code Flow

The First 12 Steps of the Authorization Code Flow are the same as for the Implicit Flow. The message sequence chart in Figure 2.5 gives an overview of the OIDC Code Flow.

(13, 14) The Authentication Response does not contain the ID Token, but a code that enables the RP to receive the ID Token. The user's browser is redirected to the redirect-uri of the RP with the code as query parameter and optionally with the state and issuer (the IdP).

(15) The RP sends the Token Request, containing the code, the RP's client-ID and redirect-uri, to the tokenEP of the IdP. The RP must authenticate to the IdP. There are different methods for this authentication. In the chart, we show an authentication with a client secret that was exchanged at the Registration. There is also the authentication method none where the RP does not authenticate. We reduced our model to those two client authentication methods.

(16) The IdP answers with the Token Response that contains the signed ID Token.

2.3 TLS

The OIDC specification [24] requires the use of Transport Layer Security (TLS) for the communication with the authEP and the tokenEP. The OAuth 2.0 specification [12] state that the IdP must use TLS for all user interactions. TLS is widely used and thus often analyzed, for example, by [4, 5, 26]. Thus, we will not analyze TLS and only shortly and on a high level describe TLS and its security properties.

TLS, specified by [22, 6], is a protocol for establishing a secure connection between a client and a server communicating over an insecure channel. A session is negotiated by the TLS Handshaking Protocol. The parties agree on security parameters and establish a secure connection through a key exchange. Today TLS is widely used, prominently as the basis for HTTPS.

With TLS, the server and the client can authenticate to each other with certificates. In practice client certificates are quite rare. Thus, mostly only the server is authenticated to the client.

By establishing a session with session keys, the server and the client are guaranteed that they communicate with the same party over one session. The encryption ensures the secrecy of the messages.

Chapter 3

Modeling Decisions

In this chapter, we give an overview of the agent roles, the protocol infrastructure, and the adversary model that we used in our OIDC model. Our OIDC Tamarin theories are available at [13].

Our OIDC Tamarin model contains the following groups of rules:

1. The public key infrastructure provides asymmetric keys as previously described in Section 2.1.6.
2. The TLS Hello rules are used to establish TLS sessions. Those rules are described in Section 3.2.
3. The TLS Adversary rules enable the adversary to use TLS. We describe them in Section 3.4.1.
4. The browser rules model the behavior of the browser. They are described in Section 3.3.
5. The behavior of compromised agents are expressed by the rules for the compromised IdP and by the rules for the compromised RP, see Section 3.4.2.
6. The agents are initialized by the initialization rules that are described in Section 3.1 .
7. The protocol rules describe the actual OIDC protocol. In Chapter 4 we give an overview of their functionality.

In this chapter, we describe the groups of rules 2 to 6. First, we give an overview of the agent roles, their identifiers, and their initialization.

3.1 Agent Roles and their Initialization

Our model distinguishes four agent roles:

- The user
- The browser
- The RP
- The IdP

A role describes how the agents in this role behave. There can be many agents in the same role. For example, there can be more than one RP, but all those RPs follow the same set of rules. We will, for example, talk about "the RP" although we mean generally agents in the role of the RP and not a specific RP.

Each agent role has a set of identifiers, parameters, and secrets. In Table 3.1 we give an overview of those identifiers, parameters, and secrets. We describe them in detail in the following. Some identifiers and parameters are modeled as fresh values to ensure their uniqueness. Some of them are still public, as they are sent out to the network after they are created.

Agent	User	Browser	RP	IdP
Identifiers	`$User`	`~browserID`, `~client_ID_TLS_Br`	`$RP`, `~client_ID_TLS_RP`	`$IdP`
OIDC Parameters	`~userID`		`~clientID`, `~redirect_uri_RP`	`~authEP_uri_IdP`, `~tokenEP_uri_IdP`
Secrets	`~userSecret`		`~client_secret`	
Private Keys				`~ltk`
Known IDs	`~browserID`, `$IdP`, `$RP`	`$User`, `$IdP`, `$RP`	`$IdP`, `~client_ID_TLS_Br`	`$User`, `$RP`, `~client_ID_TLS_RP`, `~client_ID_TLS_Br`
Known OIDC parameters			`~authEP_uri_IdP`, `~tokenEP_uri_IdP`	`~clientID`, `~redirect_uri_RP`, `~userID`
Known secrets				`~userSecret`, `~client_secret`

Table 3.1: Overview of the identifiers, parameters and secrets for each role. `~Client_ID_TLS_Br` is a short form for `~Client_ID_TLS_Browser`.

Some agents communicate over TLS channels, and the user communicates with the browser over a secure channel. They use different identifiers for the different channels. Figure 3.1 shows the roles, their connections through channels and the identifiers they use for each channel.

In the TLS communication, there are two additional roles, the client and the server. The browser and the RP can take part in a TLS session in the role of the client, and the role of the server can be taken by the IdP and the RP.

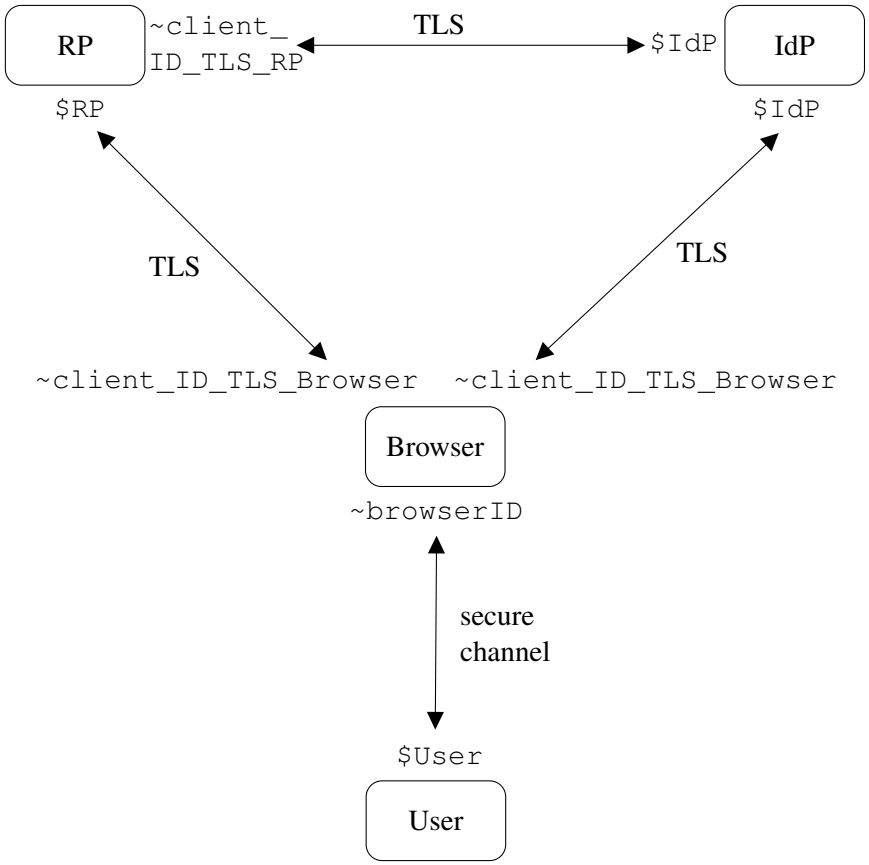

Figure 3.1: The communication channels and their identifiers in our OIDC model

For each TLS session, the server and the client use identifiers. The server uses its public name, namely `$IdP` and `$RP`, and the client uses for each session a fresh `client_ID_TLS`[1], see Section 3.2.

The user and the browser communicate over a secure channel. The user uses its public name `$User` and the browser uses its fresh identifier `~browserID`. Thus, the user, the RP, and the IdP are globally identified by their public name, and the browser by its `~browserID`.

The user needs to be able to refer to the RP and IdP with which she wants to log in. Thus, the RP and IdP need a human-readable name. They also need to be addressed as servers for the TLS sessions. We decided that a human-readable URI like "digitec.ch" would cover both of those requirements. We model this human-readable URI as a public name, namely the above mentioned names `$IdP` and `$RP`.

[1] We will refer to the TLS session identifier as client-ID-TLS, not to be confused with the client-ID of the RP, defined by OIDC.

The following list describes the identifiers in detail.

- `$User`: The user's public name is used for the communication with the browser and to create an account at the IdP, see later.
- `~browserID`: This ID is used to match a browser to exactly one user. It is used for the browser-user communication and it is used to match a client-ID-TLS to a browser, see the next point. This identifier is only known by the user and the browser.
- `~client_ID_TLS_Browser`: The browser has, for each TLS session, a client-ID-TLS as described in Section 3.2. The browser can have TLS sessions with the RP and IdP. Thus, the RP and IdP can remember the client-ID-TLS of the browser.
- `$RP`: The RP is identified through its public name. It can be understood as a human-readable URI. The user knows this name and can compare it. The IdP learns it at the Registration. It is also used by the browser to establish a TLS session.
- `~client_ID_TLS_RP`: The RP can communicate with the IdP over TLS. There the RP acts in the role of the client. Thus, the RP can create a client-ID-TLS, which is shared with the IdP.
- `$IdP`: The IdP is, as the RP, identified through its public name. It can also be seen as a human-readable URI. The user uses this name to create an account at the IdP. The RP uses this identifier to register, and the browser and the RP use it to communicate with the IdP via TLS.

OIDC consists of the Discovery, Registration, and Core. The Discovery and Registration are mostly executed once for each IdP, RP pair, while the Core is run for each user authentication. We only modeled the Core. The OIDC Core requires that the RP and IdP exchanged parameters and secrets at the Discovery and Registration. We model this with two initialization rules, one for the RP and one for the IdP. The public parameters are `~clientID`, `~redirect_uri_RP`, `~authEP_uri_IdP` and `~tokenEP_uri_IdP`. They exchange the secret `~client_secret`. Those parameters and secrets are defined by the OIDC specification [24], and we describe them in more detail below. The public parameters are modeled as fresh values to ensure their uniqueness. In the rule in which they are created, they are sent out to the network. Thus, they are public parameters.

The User needs to have an account at the IdP, where he can log in. We modeled this account with the username, password pair `~userID` and `~userSecret`. This account creation is modeled as one rule, where the two fresh values are created and put into a state fact.

The IdP needs a secret key `~ltk` to sign the ID Token. We used the public key infrastructure as described in Section 2.1.6.

The following list explains the parameters, secrets, and keys in more detail:

- `~userID`: This identifier is used for the user's account at the IdP to identify the user. It is generated at the account creation. The user has one user-ID for each IdP account. The `($IdP, ~userID)` pair uniquely identifies a user. The user-ID is passed to the RP inside the ID Token.
- `~userSecret`: The user uses this secret to authenticate to the IdP. The secret is created at the account creation. The user has one secret for each IdP she has an account at.
- `~clientID`: It identifies the RP to the IdP. It is exchanged between the RP and IdP at the Registration.
- `~redirect_uri_RP`: The redirect-uri is used to redirect the user's browser to the RP after the user has given consent to the IdP. The IdP has to know this URI to check if the URI sent by the Authentication Request is valid.
- `~client_secret`: The RP can authenticate to the IdP via a client secret. This secret is shared between the RP and IdP.
- `~authEP_uri_IdP`: The user authenticates to the IdP at the authEP. It is known by the RP because the RP has to redirect the user's browser to the authEP.
- `~tokenEP_uri_IdP`: The RP sends the Token Request to this URI. It belongs to the IdP and is known by the RP. The tokenEP is only used in the Code Flow.
- `~ltk`: The IdP uses a long term key to sign the ID Token. This key is created at the initialization of the IdP. The long term key infrastructure is described in Section 2.1.6. The corresponding public key `pk(~ltk)` is public and can be used by the RP to verify the signature.

As described above, we interpret the public names `$RP` and `$IdP` as human-readable URIs. We distinguish between those human-readable URIs and the URIs that are public parameters such as the `~redirect_uri_RP`, `~authEP_uri_IdP`, and `~tokenEP_uri_IdP`. They can be seen as some URI paths that are too long to be understood by the user. They cannot be remembered by the user and are thus not contained in the state facts of the user. When a redirect to such a URI happens the user does not check the exact URI, but the user sees the prefix of the URI, which is the public name of the server. Thus, the user gets the public name of the server when the browser shows the user some content, see Section 3.3.

3.2 TLS Model

We modeled that all communication uses TLS, although not all communication in OIDC requires the use of TLS. The authors of the OIDC specification decided that it

is not necessary to use TLS for the communication between the browser and the RP. We decided to use TLS for all communication for simplicity and due to the wide use of TLS.

The OIDC specification [24] requires server certificate checks. Client certificate checks are not required. Thus, we assume an authenticated server and an unauthenticated client. We assumed that TLS and the public-key infrastructure providing certificates are secure and all the security guarantees, described in 2.3, hold.

We abstract TLS as a channel with a setup phase. All the messages sent in our OIDC model are sent over TLS.

In OIDC there are two sorts of TLS communications: The communication between the RP and the IdP (only in the Code Flow) and the communication between the browser and the RP and IdP. In the following, we will call the first type server-to-server communication and the second type browser-to-server communication.

In TLS, one party is called client. To avoid confusion between the TLS client-ID and the client-ID from OIDC, we will use the term client-ID-TLS for the TLS rules. For a clear layout, we distinguish between the `client_ID_TLS_Browser` and the `client_ID_TLS_RP`. Those are both client-ID-TLS.

Our TLS model has three parts:

1. The session initialization that contains two pairs of Hello rules
2. The facts for sending and receiving messages
3. The adversary rules, see Section 3.4.1

In this Section, we will describe the first two parts. The adversary rules are described in Section 3.4.1.

There are two Hello rules for the browser-to-server communication and two analogous ones for the server-to-server communication. In the following, we provide the Hello rules for the browser-to-server communication.

```
rule Hello_Browser_TLS:
    [ !St_Browser_Init(~browserID, $User)
    , Fr(~client_ID_TLS_Browser) ]
  --[ BrowserServerSession(~browserID, $Server
    , ~client_ID_TLS_Browser) ]->
    [ !St_Browser_Session(~browserID, $Server
    , ~client_ID_TLS_Browser) ]


rule Hello_Server_TLS_with_Browser:
    [ !St_Browser_Session(~browserID, $Server
    , ~client_ID_TLS_Browser) ]
  -->
```

```
14      [ !St_Server_Session($Server
15      , ~client_ID_TLS_Browser) ]
```

Listing 3.1: TLS Hello rules for browser-to-server communication

Each rule establishes a state fact. They refer to a session between a client and a server. In the following, we will call them session facts. `!St_Browser_Session` is used by the browser and `!St_Server_Session` is used by the server. The server can be an IdP or a RP. Thus, the public name `$Server` will be instantiated with the public name of a RP or the public name of an IdP. The agents can only send and receive messages over TLS if the state contains a session fact with their public name or, for the browser, with its browser-ID. The client-ID-TLS in the session facts uniquely defines the TLS session.

The server only has the client-ID-TLS of the browser and not the browser's ID in its `!St_Server_Session` fact because the browser is not authenticated. The adversary can also open a TLS session, as described in Section 3.4.1. Thus, from the server's point of view, the client-ID-TLS could originate from anyone.

Sending and receiving messages is modeled with the facts `!Server_to_Client_TLS` and `!Client_to_Server_TLS`. They model a secure channel as previously described in Section 2.1.5. The rules modeling TLS are independent of the OIDC model and can be used to model different protocols that use TLS. To illustrate our model of the TLS communication we adapt the protocol rules from Example 2.1 such that the client communicates with the server over TLS. In this example, we use server-to-server communication. Server-to-server and browser-to-server communication are modeled almost the same, only the session facts differ slightly, as one can see in the following example.

```
1  rule Client_sends_nonce:
2      [ !St_Client_Session($C, $S, ~client_ID_TLS)
3      , Fr(~nonce) ]
4    -->
5      [ !Client_to_Server_TLS(~client_ID_TLS, $S
6        , <~nonce, $C>)
7      , St_Client_1(~nonce, $C) ]
8
9  rule Server_receives_and_signs:
10     [ !Client_to_Server_TLS(~client_ID_TLS, $S
11       , <nonce, $C>)
12     , !St_Server_Session($S, ~client_ID_TLS)
13     , !Ltk($S, ~ltk) ]
14   --[ Send($S, nonce) ]->
15     [ !Server_to_Client_TLS($S, ~client_ID_TLS
16       , revealSign(nonce, ~ltk)) ]
17
```

```
18 rule Client_receives:
19     [ !Server_to_Client_TLS($S, ~client_ID_TLS
20       , revealSign(~nonce, ~ltk))
21     , !St_Client_Session($C, $S, ~client_ID_TLS)
22     , !Pk($S, pk(~ltk)), St_Client_1(~nonce, $C) ]
23   --[ Finish(), Secret(~nonce), Authentic($S, ~nonce) ]->
24     [ ]
```

Listing 3.2: Protocol rules of Example 2.1 using TLS server-to-server communication

In rule `Client_sends_nonce`, the conclusion in line 5 contains the fact `!Client_to_Server_TLS(~client_ID_TLS, $S, <~nonce, $C>)`. This fact states that the client with the `~client_ID_TLS` sends the server `$S` the message `<~nonce, $C>`. Sending via TLS requires a TLS session. Thus, when the conclusion of a rule contains a `!Client_to_Server_TLS` fact, the premise has to contain a `!St_Client_Session` fact. In rule `Client_sends_nonce`, the premise contains the fact `!St_Client_Session($C, $S, ~client_ID_TLS)` in line 2. This fact states that the agent `$C` and `$S` have a TLS session.

For the client to receive a message over TLS, the premise of the rule has to contain a session fact and a fact `!Server_to_Client_TLS` where the client-ID-TLS is matching. This can be seen in rule `Client_receives` in line 19.

In rule `Server_receives_and_signs` in line 9, the server receives a message and sends a message to the sender of the first message. Thus, the premise contains a `!St_Server_Session` fact and a `!Client_to_Server_TLS` fact. The conclusion contains a `!Server_to_Client_TLS` fact.

3.3 Browser Model

We explicitly model the user and the browser as separated roles. There are actions that the user has to do consciously, such as clicking on a consent button, and there are messages the user has to compare, such as on which RP the user gives consent. In contrast, the browser does what it is told to. The browser gets automatically redirected, shows the user content provided by a server, and sends the user's input to the server.

We assume that the browser belongs to a single user. This is not true for all cases, but it reflects the cases where a device is used by a single person.

We assume that the device the browser is running on, for example a laptop or a phone, is not compromised. We also assume an uncompromised browser. Our security properties give guarantees to the user. We cannot reason about the security guarantees a user gets with a compromised browser. The user cannot get any security guarantees when interacting with a compromised device. For example, a compromised platform

could start sessions without the user doing anything, thus it cannot hold that the user started the session.

The browser can communicate with the user and with the servers. A server can be a RP or an IdP. The browser communicates with the servers over TLS, using the rules and facts described in Section 3.2. The user and the browser communicate over the device's screen and keyboard, where the browser is running on. This is modeled as a secure, replay protected channel as described in Section 2.1.5. We will also refer to this channel as the user-browser interface. The channel is modeled using two facts. `User_InputTo_Browser` contains the name of the sending user and the ID for the receiving browser as well as a message that is being sent. The fact `Browser_Shows_User` also contains the browser-ID, user name and a message, but it also contains a server name. The browser shows the user messages from servers. Because the browser has TLS connections to authenticated servers, the user is able to see from which server the message originates, i.e. what the URI of the displayed website is. The user-browser channel facts are linear because a replay attack on a screen or a keyboard would require a corrupted device, which we ruled out as stated above.

The browser is not aware of the protocol it is part of. Thus, the browser has no state facts that keep track of the execution of the protocol. The browser only keeps track of the TLS sessions it is part of and the user it belongs to. The state fact `!St_Browser_Init` tells the browser to which user it belongs. The facts `!St_Browser_Session` tell the browser which TLS sessions it is part of.

The behavior of the browser is modeled by the following three browser rules:

```
1  //The browser reacts to 'sendTo' command of the user
2  rule Browser_Redirects_to_name:
3      [ User_InputTo_Browser($User, ~browserID
4        , <'sendTo', $Server, message>)
5      , !St_Browser_Init(~browserID, $User)
6      , !St_Browser_Session(~browserID
7        , $Server, ~client_ID_TLS_Browser)
8      ]
9    -->
10     [ !Client_to_Server_TLS(~client_ID_TLS_Browser
11       , $Server, message) ]
12
13 //The browser reacts to 'sendTo' command of a server
14 rule Browser_Redirects_to_uri:
15     [ !Server_to_Client_TLS($Server1
16       , ~client_ID_TLS_Browser1
17       , <'sendTo', ~uri, message>)
18     , !St_Browser_Session(browserID, $Server1
19       , ~client_ID_TLS_Browser1)
```

```
20      , !St_Browser_Session(browserID, $Server2
21        , ~client_ID_TLS_Browser2)
22      , !Uri_belongs_to(~uri, $Server2) ]
23    -->
24      [ !Client_to_Server_TLS(~client_ID_TLS_Browser2
25        , $Server2, message) ]
26
27  //The browser reacts to 'show' command of a server
28  rule Browser_Shows:
29      [ !Server_to_Client_TLS($Server
30        , ~client_ID_TLS_Browser
31        , <'show', message>)
32      , !St_Browser_Session(~browserID, $Server
33        , ~client_ID_TLS_Browser)
34      , !St_Browser_Init(~browserID, $User) ]
35    -->
36      [ Browser_Shows_User(~browserID, $User, $Server
37        , message) ]
```

Listing 3.3: The browser rules that model the reactions of the browser to commands

The Browser reacts to commands sent by the user and the servers. Each of the three rules models the browser's reaction to one command. They are independent of the rest of the protocol. The three browser rules are also independent of each other. We modeled the commands as constants. The browser accepts two commands `'sendTo'`, and `'show'`.

`'sendTo'` lets the browser send a message to a server. This is equivalent to a redirect with attached parameters where the message represents the parameters. We modeled this using two rules, one for the case where the user sends the command and one for the case where the command is sent by a server. With the rule `Browser_Redirects_to_name`, the browser reacts to the user input `<'sendTo', $Server, message>`. The browser reacts to that by sending the `message` to the `$Server` over TLS. With the rule `Browser_Redirects_to_uri` the browser reacts to a command sent by a server.

In both rules, the browser is provided a server where the browser has to redirect to. The user provides the public name of that server, `$Server`. As discussed earlier those public names represent human-readable URIs. While in rule `Browser_Redirects_to_uri` the server is provided as an URI, `~uri`. This URI is not human-readable and can be, for example, the `~tokenEP_uri_IdP`. URIs are matched to servers via the fact `!Uri_belongs_to`, see line 22. This fact is created in the rule where the fresh URI value is created. For the RP and IdP, this is in the initialization rules.

The command `'show'` lets the browser show the user some message on the website of the server. The message is sent via TLS to the browser and the browser redirects the message via the user-browser interface to the user. Note that the fact `Browser_Shows_User` contains the public name of the server `$Server` which corresponds to the URI bar in the browser in which the user sees the current server's URI.

3.4 Adversary Model

As described above in Section 2.1.2, Tamarin's built-in adversary is a Dolev-Yao adversary. We modeled the communication over TLS channels, see Section 3.2. To enable the adversary to use TLS, we added adversary TLS rules. We also gave the adversary the ability to compromise agents. In the following, we describe those two abilities of the adversary.

3.4.1 Adversary uses TLS

In our TLS model, the client does not authenticate. A server cannot know whether the TLS session is with an honest client or with the adversary. Thus, we enabled the adversary to use TLS. The adversary can open a new TLS session with any server using the adversary's TLS Hello rule. In this rule, the adversary's client-ID-TLS is freshly created to prevent that the adversary corrupts existing TLS sessions. The adversary can send messages to this session and the adversary can receive the messages sent to this session. We called the sessions opened by the adversary malicious sessions. The role of the adversary in such a malicious session can be seen as the role of the browser an attacker is using. We could look at it as a compromised browser without a user. In Section 3.4.2 we describe how we model an adversary on the server side.

3.4.2 Compromised Agents

We gave the adversary the ability to compromise the agents IdP and RP. A compromised agent is controlled by the adversary. The adversary can send and receive messages over compromised agents and learns the knowledge of the compromised agents. We described in Section 2.1.6 how the security properties around compromised agents are handled.

We decided not to model a compromised browser nor a compromised user. We discussed why we did not model a compromised browser with a user in Section 3.3. The role of a compromised browser without a user is covered by the malicious TLS session described in Section 3.4.1. Almost all abilities of a compromised user can be modeled by the adversary using TLS. The only thing the adversary cannot do is creating an account at the IdP. We created a version where the adversary can create an account. We discuss the security implications in Chapter 5.

For simplicity, we assume that agents can only be compromised before the user starts the protocol. We enforced this with a restriction. In Tamarin, restrictions limit the possible traces, see Section 2.1.4. An agent that gets compromised during the protocol execution gives the adversary fewer abilities than an agent that is compromised from the beginning of the protocol execution. Thus, this restriction does not weaken the adversary.

If an agent gets compromised, it sends out its parameters, its secrets, and its public name. If an agent is compromised, it can still act in its intended role, as described in Chapter 4. We introduce rules to enable the compromised agents to send and receive messages over TLS sessions, they are part of. The rules are similar to the adversary TLS rules. `In` facts are translated into TLS facts. The received TLS messages are sent out with an `Out` fact. This behavior is very similar to the adversary TLS rules but it enables the adversary to act in a TLS session as the server and not only as the client.

Chapter 4

OpenID Connect Model

We wrote three main OIDC theories that are available at [13]. One Implicit Flow version, see Section 2.2.1, and two Code Flow versions, see Section 2.2.2. First, we give an overview of the common structure of those theories and we discuss the differences. Afterwards, we describe the desired security properties defined by the lemmas. We have discussed the protocol infrastructure, containing the TLS model, the browser model, the initialization of the agents, and the adversary model in Chapter 3. Now, we discuss the protocol rules.

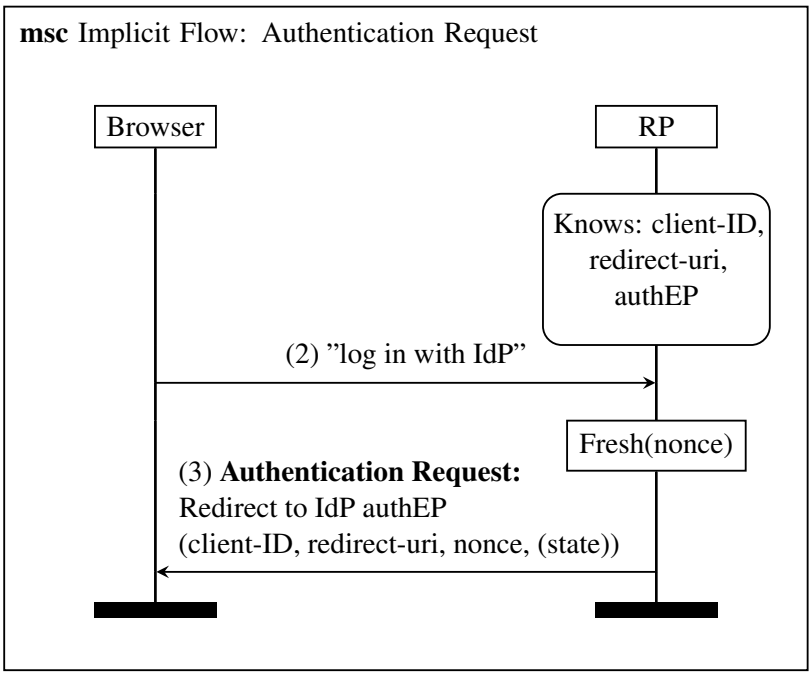

Figure 4.1: Message sequence chart of the Authentication Request in the Implicit Flow

Each protocol rule models the behavior of one agent role. They mostly have the same form. The premise contains an incoming message over TLS or the browser-user interface, and an earlier established state fact. The conclusion contains an outgoing message, also over TLS or the browser-user interface, and a new state fact. The rules are labeled with action facts that are later used by the lemmas. Only the behavior of the user, RP, and IdP are modeled by the protocol rules. The behavior of the browser is modeled by the browser rules, that are independent of the protocol, see Section 3.3.

The protocol rules are mostly straightforward translations of the messages shown in the message sequence charts in Figures 2.4 and 2.5. To illustrate how the protocol rules work, we look at the RP receiving the login request from the browser and then sending the Authentication Request back to the browser. In this example, we look at the Implicit Flow. The protocol snippet we look at is provided in the message sequence chart in Figure 4.1. The following rule describes this behavior. We will describe this rule in the following.

```
1  rule AuthRequest_RP:
2      [ !Client_to_Server_TLS(~client_ID_TLS_Browser, $RP
3        , <'loginWith', $IdP>)
4      , !St_Server_Session($RP, ~client_ID_TLS_Browser)
5      , !St_RP_Registered($RP, $IdP, ~clientID
6        , ~redirect_uri_RP, ~authEP_uri_IdP)
7      , Fr(~nonce) ]
8    --[ AuthRequestRP(~nonce), UsedNonce(~nonce)
9      , RP_sends_AuthRequest($RP, $IdP
10       , ~client_ID_TLS_Browser)
11     , RP_sends_AuthRequest_injective($RP, $IdP
12       , ~client_ID_TLS_Browser, ~nonce)
13     ]->
14     [ !Server_to_Client_TLS($RP, ~client_ID_TLS_Browser
15       , <'sendTo', ~authEP_uri_IdP
16         , <'authRequest', ~clientID, ~redirect_uri_RP
17           , ~authEP_uri_IdP, ~nonce>>)
18     , St_RP_1($RP, $IdP, ~client_ID_TLS_Browser
19       , ~clientID, ~redirect_uri_RP, ~authEP_uri_IdP
20       , ~nonce) ]
```

Listing 4.1: Rule for the Authentication Request in the implicit Flow

The fact `!Client_to_Server_TLS` in line 2 describes that the RP receives the message `<'loginWith', $IdP>` as shown in the message sequence chart above in message (2). This fact can be created by the browser rules in Section 3.3. The session fact `!St_Server_Session` in line 4 guarantees that the RP has a corresponding TLS session with the browser sending the message. The knowledge of the RP is given by the state fact `!St_RP_Registered` in line 5. The fact `Fr(~nonce)` in line 7 lets the RP create a fresh nonce. The action facts in

lines 8 to 12 label the rule and are later referred to in the lemmas. The Authentication Request sent by the RP to the browser is contained in the fact `!Server_to_Client_TLS` in line 14. The RP sends the browser a `'sendTo'` command. This lets the browser redirect to the `~authEP_uri_IdP`, with the message `<'authRequest', ~clientID, ~redirect_uri_RP, ~authEP_uri_IdP, ~nonce>` as shown in message (3). The browser's redirection is modeled by the browser rules in Section 3.3. The RP remembers that it sent the Authentication Request by creating the state fact `St_RP_1` in line 18. It contains the public name of the RP and IdP, the earlier exchanged parameters for this RP-IdP pair, and the new nonce.

In our model, we left out the state parameter which is only recommended and not required by the OIDC specification [24]. The state can be added by the RP to the Authentication Request and must then be sent back by the IdP inside the Authentication Response. The state value is used to keep state between the Authentication Request and the Authentication Response.

On the other hand, in our model, when the RP receives the Authentication Response, it matches the client-ID-TLS of the sending browser with the client-ID-TLS of the browser that sent the start message to the RP. Thus, the RP checks that the browser that started the protocol will also send the Authentication Response. We assumed this because we assume that the TLS session stays open during the short time where the user authenticates and gives consent to the IdP. This though may be a too strong assumption. In reality, the RP would probably also accept an Authentication Response from a different TLS session.

Thus, the state performs the task that the matching of the TLS sessions does in our model. Leaving out the state parameter but matching the TLS sessions should have the same effect as adding the state parameter and not matching the TLS sessions.

The main difference between the Implicit Flow and the Code Flow is the way the IdP sends the ID Token to the RP. In the Implicit Flow, the ID Token is sent in the Authentication Response to the RP over the user's browser. In the Code Flow, the IdP sends a code to the RP over the user's browser. The RP has to send the code in an Token Request to the IdP, and the IdP sends the ID Token to the RP. Thus, the Code Flow requires server-to-server TLS communication between the RP and IdP. In the Implicit Flow, we only have server-to-browser TLS communication.

We modeled two versions of the Code Flow, one where the RP does not authenticate to the IdP and one where the RP authenticates to the IdP with the client-secret. This authentication of the RP is referred to in the OIDC specification [24] as client authentication, not to be confused with the TLS client authentication. We will refer to it as OIDC client authentication.

The method of the OIDC client authentication is decided in the Registration. Thus, there is a fixed OIDC client authentication for every RP, IdP pair. Whether in a protocol instance the Implicit Flow or the Code Flow is used is decided by the response

type parameter which is sent by the RP in the Authentication Request. For simplicity, we modeled the different flows and OIDC client authentication methods in three separate theories, and thus omitted the response type parameter.

We decided to only model the parameters that are required by the OIDC specification [24]. Thus, we did not include the state and we only used the nonce in the Implicit Flow and not in the Code Flow.

The ID Token is a core element of the OIDC protocol. It is defined as a JWT. We thus modeled the ID Token as tuples of constants and parameters. The following shows the message that represents the ID Token in our model.

```
1 <'idToken'
2   , <'iss', $IdP>
3   , <'sub', ~userID>
4   , <'aud', ~clientID>
5   , <'nonce', ~nonce>>
```

Listing 4.2: Message that represents the ID Token

The implementation of the code is not specified by the OIDC specification [24]. We decided to model it as a fresh value to match the code with the Authentication Request at the IdP.

The user has to check to what she gives consent to. The user should only be able to check data a human user could actually understand and compare. As described in Section 3.1, the public names of the RP and IdP, `$RP` and `$IdP`, represent human-readable URIs, for example, “digitec.ch” and “google.com”. The TLS session IDs of the browser and the redirect-URIs and the RP’s client-ID cannot be understood by the user. Thus, the user only compares the public names `$RP` and `$IdP` with the agents the user wanted to log in.

4.1 Security Properties

The OIDC specification [24] does not explicitly define security properties for OIDC. We choose three security properties that we consider to be important and expressed each in a lemma. In the following, we will mainly describe the captured security properties and not the lemmas themselves, although we will refer to the security properties by the names of the lemmas that describe them.

We describe the security properties depending on the receiving of the ID Token. In the following, we will call an ID Token with a user’s user-ID in it the user’s ID Token. When a RP receives and accepts an ID Token of a user this corresponds to the user being logged in to that RP. Thus, our lemmas use the action fact `RP_gets_IDToken(rp, uid, idp)` that describes that the RP `rp` receives and accepts an ID Token of the user with user-ID `uid` used by the IdP `idp`. We will

call the IdP that authenticated a user and issued the according ID Token to the RP "the issuing IdP".

The properties we will describe should hold for a group of honest agents. The RP that receives and accepts the ID Token, the IdP that authenticates the user, the user who is being authenticated, and the user's browser have to be uncompromised. The security properties should hold, even if those honest agents in parallel have sessions with compromised agents.

The lemma `User_gives_Consent_to_RP_getting_ID_Token` captures a security property desired by the user. A user's ID Token should only be received and accepted by an RP if the user has earlier consented to that and started the session. The user has to send a consent message to the IdP. With that consent message, the user consents to his account information being sent to the RP. According to the OIDC specification [24], the user should also give consent to the provided profile information. The ID Token in our model only contains the user-ID as profile information. Thus, we left the profile information out for simplicity.

The second lemma `RP_sent_AuthRequest_for_IDToken` describes that the RP should only receive and accept a user's ID Token if that RP has earlier sent an Authentication Request to the issuing IdP. This Authentication Request should be a reaction to the user's message "log in with IdP". This message is sent over the user's browser, thus the RP only sees the client-ID-TLS of the user's browser. To express that the Authentication Request needs to be a reaction to the user's login message, we matched the client-ID-TLS with the browser-ID and then the browser-ID with the user that owns this browser.

In the Implicit Flow, it is required to send a nonce with the Authentication Request and to include this nonce in the ID Token. The nonce should, according to the OIDC specification [24], prevent replay attacks. A replay attack violates an injective agreement property. An agent a in role A, has an injective agreement with an agent b in role B, on the message m if a has run the protocol in role A apparently with agent b in role B, and agent b has run the protocol in role B apparently with agent a in role A, they both have agreed on the message m, and there is a unique run of b for the run of a. The above described properties only describe non-injective agreement properties. Thus, we added to the Implicit Flow an injective agreement lemma of the above properties.

The last security property we are interested in is captured by the lemma `Intended_User_is_logged_in`. The user who gets logged in at the RP is the user whose browser was redirected to the RP with the according Authentication Response. In case of the Implicit Flow, the Authentication Response contains the ID Token, and in the case of the Code Flow, it contains the code for the ID Token. This lemma should rule out that an adversary logs in under the user's account and that the user gets logged in under the adversary's account[1].

[1] In our theories, this is yet not possible, because the adversary cannot create an account.

We did not explicitly formulate a secrecy lemma for the ID Token. A compromised RP is able to receive and leak an ID Token, but no honest RP would accept this ID Token. The ID Token contains the audience, which an honest RP checks before accepting an ID Token. Thus, the secrecy of the ID Token is not a required security property.

Chapter 5

Results

In our three main theories, the lemmas described in Section 4.1 all hold. We were able to find two attacks on versions where some parameters are left out. They can be easily prevented but can be possible with careless implementations. We will describe those two attacks and the countermeasures we built inside the main theories in the following sections. The attack graphs and the disproven Tamarin theories are available at [13].

As mentioned in Section 3.4.2, we created a theory where the adversary can create an account. We were able to prove the first two of our security lemmas, and we got a trace where the adversary successfully authenticates via OIDC. We could not yet get the third lemma to terminate, we stopped the proof search after some hours.

5.1 Adversary forges the User's Consent

In an earlier version of the Implicit Flow theory, we found an attack on the lemma `User_gives_Consernt_to_RP_getting_ID_Token`. The adversary was able to forge the consent of the user. The consent message required by the IdP was `<'Consent', $IdP, $RP, $User>`, see message (12) in the message sequence chart in Figure 2.4. The IdP expects this message to originate from the user's browser.

In the attack, the adversary sends the browser over a TLS session the following message: `<'sentTo', ~uri, <'Consent', $IdP, $RP, $User>>`. The `~uri` is an authEP of the IdP. The browser reacts to this message by redirecting the consent message to the IdP. The IdP accepts this message because it originates from the user's browser, but the user, in fact, did not give consent. The IdP will give the ID Token to the RP without the user having consented.

This attack is a Cross-Site Request Forgery (CSRF) attack, as for example described by [28]. In a CSRF attack, the user's browser is forced to make a request to a target server. The server cannot distinguish between requests made by the user and requests

sent to the browser by an attacker. Thus, the target server will assume the request came from the user. Mostly CSRF happens when a user is authenticated to some target server and the user visits at the same time a malicious website of the attacker. The attacker then sends a request to the browser, which is redirected to the target server. In our case, the CSRF happens in the middle of the user consent. The attacker would need to get the point where the user has already authenticated but not yet given consent or even not yet received the consent request of the IdP. Thus, our found attack is not very likely, but possible.

The OIDC specification [24] does not state how exactly the user has to give consent, but they require the IdP to prevent CSRF attacks. Through careless implementations, such an attack could still be possible.

A common countermeasure would be to use a nonce for each session that must be contained in the responses of the user's browser. This is mostly done via browser cookies. For our model, the exact implementation of the countermeasure was not important. We assumed that the implementation of giving consent is secure. We modeled this by attaching a fresh nonce to the consent request of the IdP. The user replays this nonce in the consent message. The adversary cannot learn this nonce because it is sent over TLS. With this adaptation, the lemma `User_gives_Consent_to_RP_getting_ID_Token` holds.

5.2 IdP Mix-Up Attack

In one version of the Code Flow theory, we found an attack on the lemma `RP_sent_AuthRequest_for_IDToken`. An honest RP is tricked into sending the code to a compromised IdP instead of to the honest IdP to which the user authenticated to. The user intended to authenticate to the honest RP using the honest IdP, but the honest RP thinks the user used the compromised IdP.

The Tamarin prover does not search for the simplest attack. It tries to find one and outputs the first one it finds. Thus, the attack trees are often quite large and chaotic. They contain a lot of duplications. Therefore, we will here only present the main aspects of this attack. The main problem is that the adversary learns the code. We did not explicitly formulate a lemma that forbids this, but the adversary that learns the code can lead to a RP receiving an ID Token for which it did not send an Authentication Request, which contradicts our lemma and is thus found by Tamarin as an attack.

We will now describe how the adversary learns the code for an honest user, RP, and IdP. The message sequence chart in Figure 5.1 illustrates how the adversary gets the code coarsely. The honest RP has a session with a compromised IdP. Simultaneously, the user starts an OIDC session with the honest RP and IdP, but the adversary prohibits that the RP receives the user's message. This is illustrated in Figure 5.1 by the dot at message (2). The Authentication Request is sent to the IdP by the adversary over the user's browser. This happens using CSRF as described in Section 5.1. The

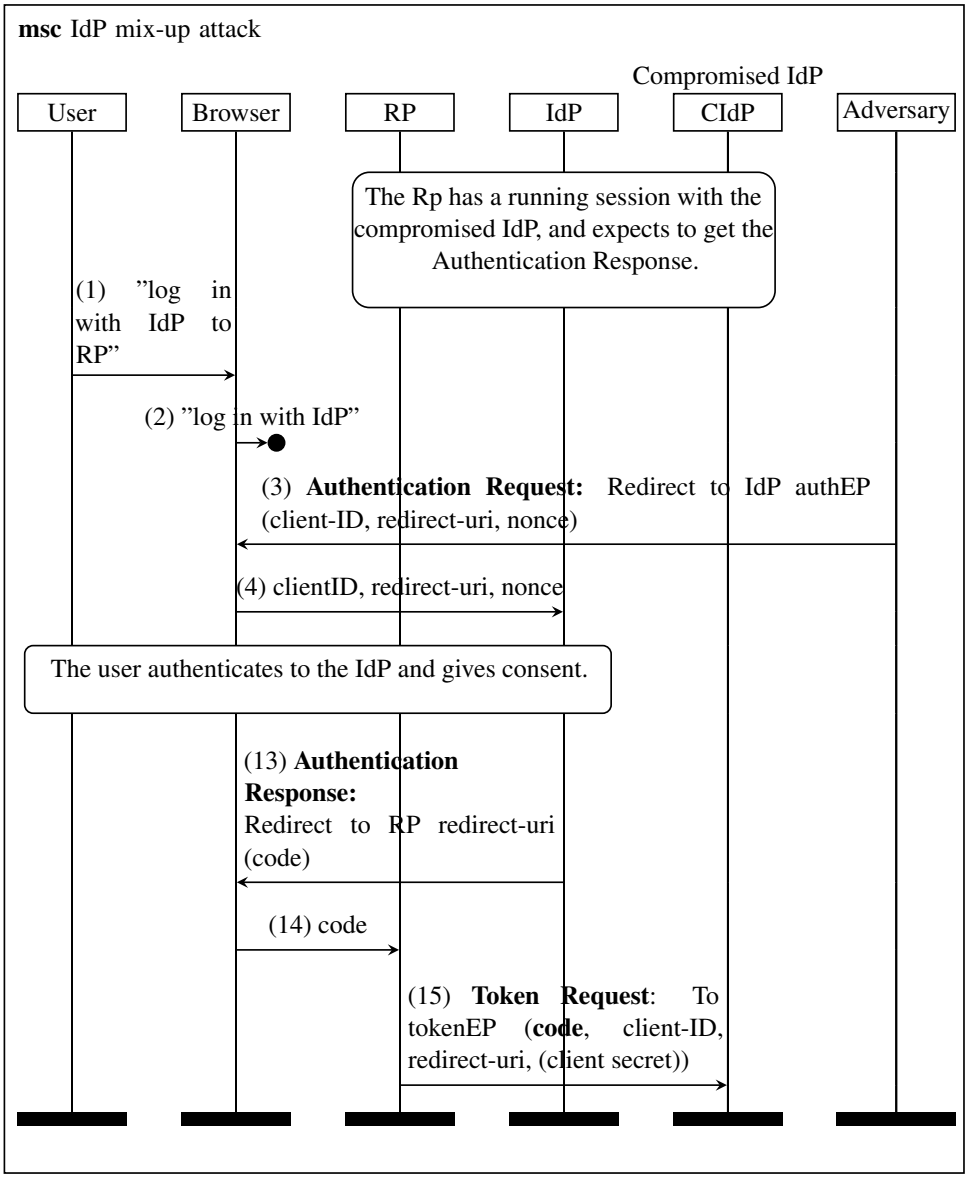

Figure 5.1: Message sequence chart of the IdP mix-up attack

user then authenticates and gives consent to the IdP normally. The IdP then sends the Authentication Response containing the code to the browser which redirects it to the RP. The RP expects a code from the compromised IdP. Thus, the RP sends the code belonging to the honest IdP to the compromised IdP. The adversary learns the code.

Fett, Küsters, Schmitz [10] describe this sort of attack as IdP mix-up attack. There are different forms of this attack. The one they describe uses the Hybrid Flow and they include the Discovery and Registration in their attack, but the concept is the same: The RP is confused with which IdP the user wants to log in. They recommend preventing this attack by sending the issuer (the IdP) with the Authentication Response. In our main theories, we added the public name of the IdP to the Authentication Response. With this adaptation, our lemmas hold and the IdP mix-up attack is not possible. We did not include the issuer in the Implicit Flow theories, because the Authentication Response of the Implicit Flow contains the ID Token, and the ID Token contains the issuer. Thus, additionally adding the issuer is not necessary.

Chapter 6

Conclusion

We modeled the Implicit Flow and two versions of the Code Flow, one Code Flow version where the RP does not authenticate and one where the RP authenticates with a client secret. With the Tamarin prover we were able to prove that the following security properties hold for those theories:

- The RP can only receive and accept the ID Token of a user if the user has earlier consented to this and started the protocol with the according RP.
- The RP can only receive and accept an ID Token of a user if the RP has previously sent an according Authentication Request which is a reaction to the request of the user's browser.
- When the RP receives and accepts an ID Token belonging to a user, this user's browser needs to have redirected the Authentication Response to the RP. This property describes that the user whose browser sent the Authentication Response should be logged in at the end of the protocol.

We found two attacks for slightly altered theory versions. Without attaching a consent nonce to the consent messages sent between the IdP and user, the adversary can forge the consent of the user via a CSRF attack. The specification [24] state that such attacks should be prohibited by the IdP, but the specification is underspecified in how the consent should happen.

In the OIDC specification, it is not stated that the issuer (the IdP) needs to be attached to the Authentication Request and Response. Without issuer, an IdP mix-up attack is possible. This attack is also described by [10]. The RP gets confused about to which IdP it is speaking to, which results in the RP sending the code to the adversary.

It cannot be guaranteed that a Tamarin theory models a protocol correctly. Abstractions and assumptions have to be made. There are two factors that improve our confidence that our model is correct. First, the traces generated for the trace existence lemmas show the desired protocol behavior. Secondly, the fact that slight changes re-

sult in attacks shows that we have an adversary with strong enough abilities to attack a too weak version of the protocol.

We abstracted TLS and the browser as independent modules that could be reused for other protocols. We modeled TLS as a secure channel, where only the server side is authenticated. The adversary is also able to open a TLS session in the role of a client.

The browser is modeled as an independent agent that is not aware of the protocol, and that serves as a link between the user and the servers. It has a secure channel to the user and can communicate with the servers via TLS. The browser reacts to commands sent by the user and servers. It can show the user messages sent by the servers and it can redirect to servers with messages provided by the user or other servers.

One has to abstract and simplify a protocol to verify it with Tamarin. Too strong assumptions may make the model obviously secure and too complex models may get confusing or too complex to be analyzed. There is not one correct way to model a protocol, but there are different abstractions that can be compared. We present here two alternative design decisions that could be interesting for future work.

We modeled the user as an active protocol participant and the browser as a participant that only reacts to commands of the other agents. The user is only able to compare human-readable messages. Thus, in the user's knowledge, or in the user's state facts, there are only such human-readable messages. Never the less, we mixed the behavior of a human user with the behavior of web applications running on the user's browser. For example, when the user gives consent, he gets a nonce from the IdP and sends this nonce back. The user should not be able to send such a nonce back, this would for example be done by a JavaScript application. Modeling the user separated from web applications could be interesting for future work. One could model web applications as additional agents or one could extend the model of the browser such that the browser can execute code.

We modeled the OIDC flows in separate theories. In reality, the RPs and IdPs support all flows. There could be attacks that only occur when different flows can be executed in parallel. For future work, it could be interesting to model the flows in one theory by adding the parameter response-type which determines in the Authentication Request which flow is executed.

Bibliography

[1] Goolge Identity Platform - OpenID Connect. https://developers.google.com/identity/protocols/OpenIDConnect, accessed 2019-02-26.

[2] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[3] Victoria Beltran. Characterization of web single sign-on protocols. *IEEE Communications Magazine*, 54:24–30, 2016.

[4] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788. ACM, 2017.

[5] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485. IEEE, 2016.

[6] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2, 2008. https://tools.ietf.org/html/rfc5246, accessed 2019-02-19.

[7] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[8] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. In *International Workshop on Rewriting Logic and its Applications*, pages 52–68. Springer, 2010.

[9] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215. ACM, 2016.

[10] Daniel Fett, Ralf Küsters, and Guido Schmitz. The Web SSO Standard OpenID Connect: In-Depth Formal Analysis and Security Guidelines. Technical Report. 2017.

[11] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web SSO standard openid connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202. IEEE, 2017.

[12] D. Hardt. The OAuth 2.0 Authorization Framework, 2012. https://tools.ietf.org/html/rfc6749, accessed 2019-02-21.

[13] Xenia Hofmeier. OpenID Connect Tamarin Theories. https://tamarin-prover.github.io/.

[14] Wanpeng Li and Chris J. Mitchell. Analysing the Security of Google's Implementation of OpenID Connect. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 357–376. Springer International Publishing, 2016.

[15] Wanpeng Li, Chris J Mitchell, and Thomas Chen. Mitigating CSRF attacks on OAuth 2.0 and OpenID Connect. *arXiv preprint arXiv:1801.07983*, 2018.

[16] Gavin Lowe. An Attack on the Needham- Schroeder Public- Key Authentication Protocol. *Information processing letters*, 56(3), 1995.

[17] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996.

[18] N. Sakimura M. Jones, J. Bradley. JSON Web Token (JWT), 2015. https://tools.ietf.org/html/rfc7519, accessed 20.3.2019.

[19] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. SoK: Single Sign-On Security—An Evaluation of OpenID Connect. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 251–266. IEEE, 2017.

[20] Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. On the security of modern single sign-on protocols: Second-order vulnerabilities in openid connect. *arXiv preprint arXiv:1508.04324*, 2015.

[21] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. Formal verification of OAuth 2.0 using Alloy framework. In *2011 International Conference on Communication Systems and Network Technologies*, pages 655–659. IEEE, 2011.

[22] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, 2018. https://tools.ietf.org/html/draft-ietf-tls-tls13-28, accessed 2019-03-02.

[23] Justin Richer. User Authentication with OAuth 2.0. https://oauth.net/articles/authentication, accessed 2019-03-28.

[24] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0 incorporating errata set 1, 2014. https://openid.net/specs/openid-connect-core-1_0.html, accessed 2019-02-21.

[25] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 78–94. IEEE, 2012.

[26] Vincent Stettler. Formally Analyzing the TLS1.3 proposal, 2016. Bachelor's Thesis.

[27] The Tamarin Team. Tamarin-Prover Manual - Security Protocol Analysis in the Symbolic Model, January 2019. https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf, accessed 2019-02-28.

[28] William Zeller and Edward W Felten. Cross-site request forgeries: Exploitation and prevention. 2008.