Module 1: Formal Verification of Security Protocols

Lab 2: The Open-ID Connect protocol

David Basin, Xenia Hofmeier and Sofia Giampietro

basin@inf.ethz.ch, xenia.hofmeier@inf.ethz.ch, sofia.giampietro@inf.ethz.ch

October 2023

Welcome to the second lab session of the Information Security Lab. This session is part of the first module on *Formal Verification of Security Protocols*.

Overview

The goal of this lab is for you to deepen your modeling and analysis skills, with a real-world protocol that many of you use daily: the OpenID Connect (OIDC) protocol. This protocol is used for delegated authentication, allowing third-party applications to verify the identity of the end-user and to obtain basic user profile information.

Specifically, you will learn how to model (i) cryptographic mechanisms , (ii) communication channels with different levels of security, (iii) multiple roles and (iv) branching execution in protocols (similar to *if* statements in imperative languages).

Overview of the open ID Connect protocol

OpenID-Connect is a protocol for delegated authentication in which a user accesses a service, henceforth called the *relaying party* (RP), using an existing account at a recognized authorization service, called an *Identity Provider* (IdP).

Example: Alice, the user, wants to log in to a website, for example, digitec.ch (digitec.ch here acts as the RP). She doesn't have a digitec.ch account but clicks on the 'sign in with Google' button (Google is an OIDC IdP) and her browser gets redirected to a Google login page. Alice authenticates to Google by providing her username and password and gives consent that Google provides some profile information such as her name or e-mail address to digitec.ch. Then Google redirects Alice's browser back to digitec.ch and is now logged in at digitec.ch with the profile information provided by Google.

Different variations of the protocol specify what the IdP sends back to the user to log in to the RP. In this lab we will focus on (a simplified version of) the *Implicit Flow* protocol, in which the IdP sends the user's browser a signed ID Token that is verified by the RP before logging the user in.

For simplicity, we will model the user and the browser as a unique entity.

Task 1.1. The protocol We consider a scenario where the user can choose the method to log in to the RP. The user can either log in directly via credentials shared with the RP or the user can log in through OIDC.

(1,2) The protocol is started by a user `$User`, who indicates to an RP that it wishes to log in by sending it the message

```
<$User, 'login'>.
```

The RP replies by requesting the user to send its credentials or to choose to log in with OIDC.

(3,4) If the user has an account with the RP, the user can choose to directly authenticate to the RP. For this model, we assume that the user authenticates with credentials (e.g. a username and password). The user replies by sending its credentials. The RP checks if the password corresponds to the username and if successful, logs the user in. Otherwise the RP aborts the login attempt.

(5) Alternatively, the user can choose to log in with OIDC. The user then indicates to the RP it wants to login with a particular IdP `$IdP` (with which it has an account), by sending it the message

```
<'loginWith', $IdP>.
```

(6,7) The RP responds with an Authentication Request with a fresh nonce (later included in the ID Token), which is forwarded by the User to the IdP.

(8,9) The IdP requests user authentication. Again, we assume that the user authenticates with a username and password pair. The user also gives consent to the IdP providing data to the RP.

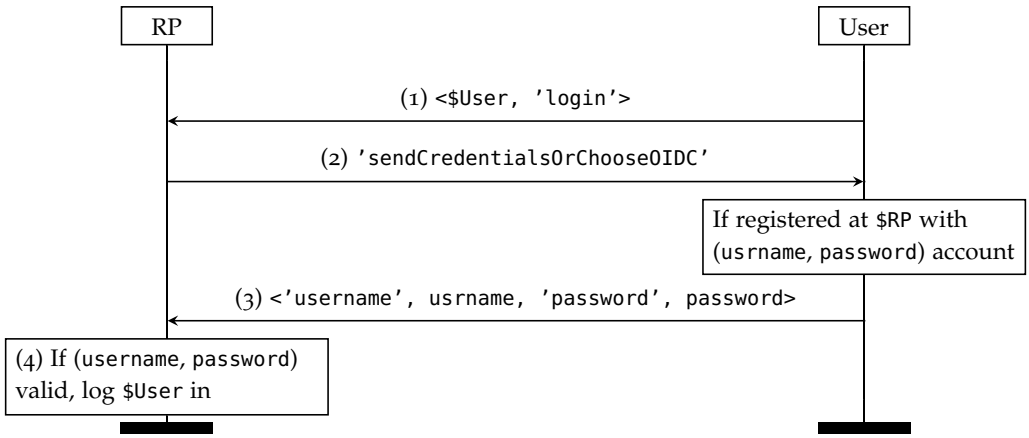
(10,11) After the user has successfully authenticated and given consent, the IdP sends an authentication response with a signed ID Token to the user. The user does not check the token but forwards it to the RP (this models the browser getting automatically re-directed, as we consider the browser and user as one agent). The ID Token is of the following form:

```
id_token = <'idToken'
         , <'iss', $IdP>
         , <'sub', username>
         , <'aud', $RP>
         , <'nonce', nonce>>
```

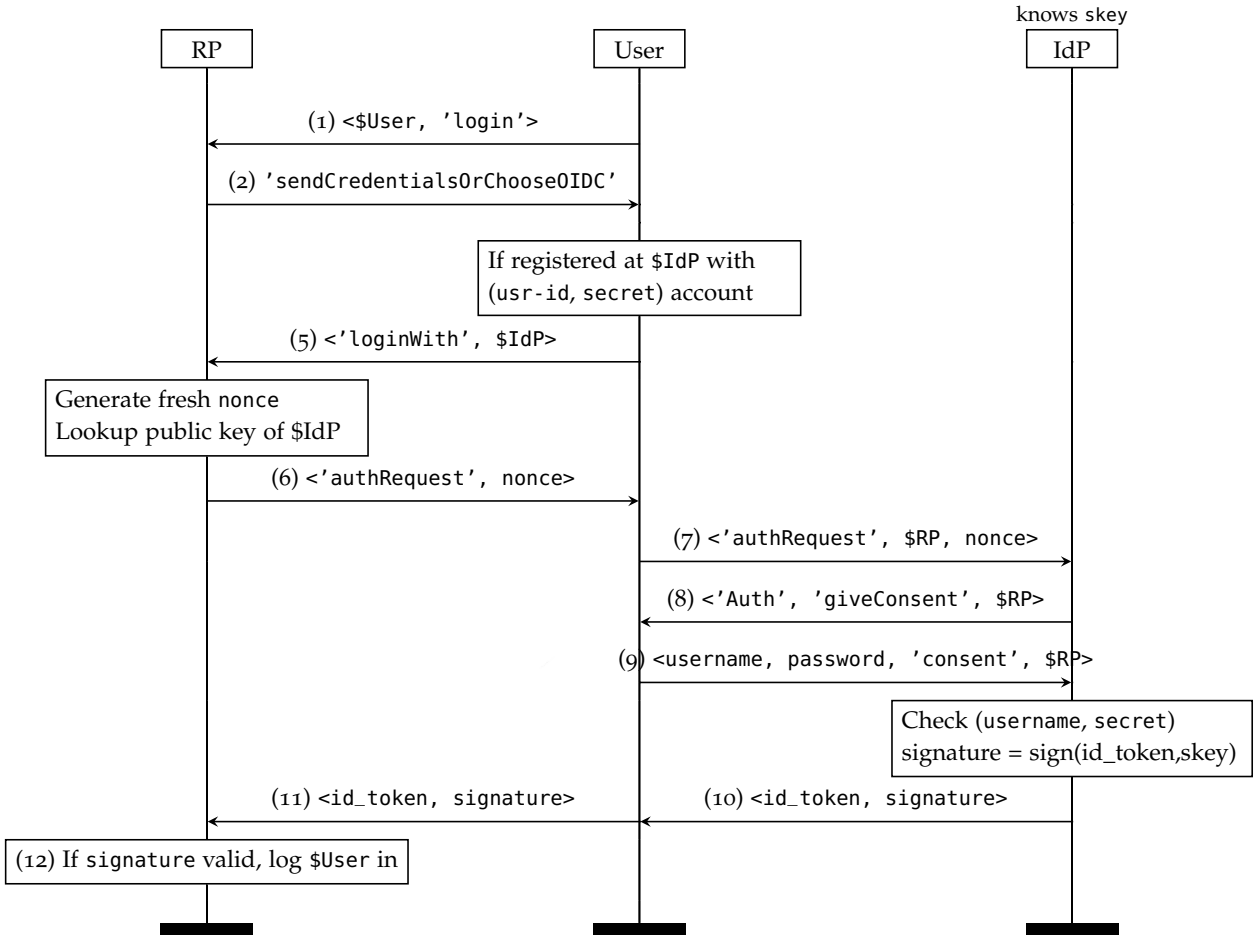
(12) When the RP receives the ID token, it checks that the nonce in the token corresponds to the one it has sent with the authentication request, it validates the signature with the IdP's public key and, if successful, it will log the user in, and otherwise it will abort.

The precise format of messages exchanged is illustrated in the message sequence charts below.

msc Direct Login



msc OIDC



Task

Download the complementary material from:
`https://moodle-app2.let.ethz.ch/mod/folder/view.php?id=960173`,
which contains the file `OIDC.spthy`.
Model the above protocol by completing the skeleton model by adding the appropriate Tamarin code (e.g. rules, lemmas, and restrictions) to it such that the resulting model satisfies the following requirements.

- Use Tamarin's built-in `signing` theory for the signature scheme. It defines the function symbols `sign/2`, `verify/3`, `pk/1`, and `true`, which are related by the equation `verify(sign(m,sk),m,pk(sk)) = true`. The skeleton file `OIDC.spthy` already includes this built-in theory.
- In the skeleton file, some rules are preceded by instructions indicating the rule should be annotated by some action fact(s). **It is crucial for tests to pass that you do so.** Note that the executability lemmas refer to these action facts.
- The IdP needs a public/private key pair to sign the ID Token. Model a public key infrastructure rule that generates such pair. The built-in `signing` theory provided by Tamarin, and already loaded in the skeleton file, provides a `pk` function. Furthermore, we want to model that each IdP can generate only one such pair. As mentioned above, make sure to annotate your rule with an action fact `IdP_Initializes($IdP, ~skey)` (indicating that entity `$IdP` has generated as secret key `~skey`), since we refer to such fact in the restriction `IdP_initializes_Once`, which you can already find in the skeleton file.
- The user must have an account at the RP or IdP. We model this account with a username and password, in the rule `Register_User_account` (already in the skeleton file), where the two fresh values (`~username` and `~password`) are created and put into a persistent state fact (since users can log in multiple times with the same account) for the user to acces, and another persistent fact for the RP or IdP to use. Notice that we model the server (RP or IdP) **only** storing the username and password pair. Furthermore, note that this rule is annotated with an action fact `OnlyOnce($User, $Server)`, restricting users to create only one account per website (see restriction `OneAccount` already in the skeleton file). It is also annotated with two action facts `Is_User($User)` and `Is_Server($Server)` ensuring that users cannot be servers (see restriction `Users_are_not_Servers` already in the skeleton file).
- To deal with execution branching in Tamarin, it is a common practice to duplicate rules. For example, say you have a protocol where an agent goes to two different states, depending on another state fact. This can be modeled with the two rules:

```
rule Go_To_1:
  [ State_0(), Condition1 ]-->[ State_1() ]

rule Go_To_2:
  [ State_0(), Condition2 ]-->[ State_2() ]
```

You can use this idea to model the **if** statement for direct versus delegated authentication.

- Decline actions need not be modeled explicitly (at least not in this lab). For example, suppose you are modeling an agent that stops the execution, provided that some condition *C* fails. Then you should simply have a rule that models the continuation of the execution, provided that *C holds*.
- All communication between an IdP and users is secured by TLS. Instead of modelling TLS (a whole protocol in itself!), we will just model TLS-secured communication as a secure channel. You will find rules modeling such a channel in the provided skeleton file (see `https://tamarin-prover.github.io/manual/master/book/011_advanced-features.html` if you're interested in learning how we can model different channels). Keep in mind that the communication between the RP and the user is, in contrast, not secure.
- We will assume that all agents are honest (i.e. the secure channel can never leak information). However, we will consider the scenario in which the secret keys of IdPs can be (maybe accidentally) leaked. Complete the rule `CompromiseAgent` that reveals an IdP's secret key. Add the action fact `Compromised($A)` to this rule. This action fact indicated that the secret key of agent `$A` was revealed.
- To simplify our model and rule out some unexpected attacks, we assume that the user performs the following two checks before sending the `<'loginWith', $IdP>` message:

  - The RP is not an IdP.
  - The RP and IdP are not the same agent.

  For those two checks, we provide two restrictions in the skeleton file: `Inequality` and `RP_shouldnt_be_an_IdP`. The skeleton file indicates which action facts you should add accordingly.

Expected results

Run the executability lemmas already provided in the skeleton. Tamarin should verify both.

Task 1.2. Security properties

We will consider the following properties.

- `lemma User_gives_Credentials`: an RP `$RP` accepts the credentials including a certain username, only if the user corresponding to that username has earlier sent these credentials to that RP.
- `lemma User_gives_Consent_to_RP_getting_IDToken`: an RP `$RP` should receive and accept an ID Token from a *non-compromised* IdP `$IdP` for username `username` only if the user corresponding to that username has previously consented to the IdP `$IdP` to give `$RP` such ID Token.

- `lemma User_Logged_In_Direct`: An RP `$RP` logs a user in via Direct Login with a certain username, only if this user is the actual owner of the username for that RP.
- `lemma User_Logged_In_OIDC`: An RP logs a user in with a username `username` received in a valid ID Token from a *non-compromised* IdP `$IdP`, only if the user is the actual owner of the username `username` with that IdP.
- `lemma Unique_Credentials`: If a RP receives valid credentials (i.e. username and password pair) for user `$User`, it cannot validate the same credentials from different User `$User2`.
- `lemma Unique_IDToken`: If a RP receives a valid ID Token from a *non-compromised* IdP `$IdP` for user `$User`, it cannot validate the same ID Token for a different User `$User2`.

We formalize these properties using the following action facts:

- `User_sends_credentials($User, $RP)`: indicates that `$User` sends their credentials to `$RP`
- `User_gives_consent($User, $RP, $IdP)`: indicates that `$User` gives consent to `$IdP` to give information to `$RP`
- `Username_belongs_to(~username, $User, $Server)`: indicates that `$User` registers `~username` with the `$Server`. This action fact is already present in the skeleton.
- `RP_accepts_credentials($RP, ~username)`: indicates that `$RP` receives and accepts the credentials including `~username` from a user.
- `RP_Logs_User_in_Direct($RP, $User, ~username)`: indicates that `$RP` successfully validates the credentials including `~username` for `$User`.
- `RP_Logs_User_in_OIDC($RP, $User, username, $IdP)`: indicates that `$RP` logs in a user `$User` with `username` after validating the ID Token containing `username` and received from `$IdP`.
- `RP_gets_IDToken_for_username($RP, username, $IdP)`: indicates that `$RP` receives a valid ID Token from `$IdP` containing `username`.
- `RP_gets_IDToken_for_user($RP, $User, id_token, $IdP)` indicates that `$RP` receives a valid ID Token `id_token` from `$IdP` for the user `$User`.
- `Compromised($A)`: indicates that agent `$A` gets compromised and their secret key is published. You should have added this fact to rule `CompromiseAgent` in the last task.

Task

Define the lemmas listed above using the action facts listed above. You will need to add these action facts in your model accordingly. Some action facts are already present in the skeleton. You don't need to add those. You may only use the above nine action facts in your lemmas.

Depending on how you model the protocol, the above lemmas might need different heuristics to be *automatically* proven. If you see that Tamarin takes a long time when trying to prove a lemma, use Tamarin's heuristic that first tries to break loops. You can indicate this heuristic should be used by annotating your lemmas by the keyword `[heuristic=S]`, for example as follows:

```
lemma User_gives_Consent_to_RP_getting_Token[heuristic=S]:
```

Recall that normally lemmas should be proven/disproven in less than a couple of minutes.

Expected results

You should obtain the following results:

- `lemma User_gives_Credentials`: verified
- `lemma User_gives_Consent_to_RP_getting_IDToken`: verified
- `lemma User_Logged_In_Direct`: falsified
- `lemma User_Logged_In_OIDC`: falsified
- `lemma Unique_Credentials`: falsified
- `lemma Unique_IDToken`: verified

Exercise sheet. What do the lemmas `User_Logged_In_Direct` and `User_Logged_In_OIDC` indicate? Why do neither hold? Try to understand the counterattacks found by Tamarin.

Submission

Your task is to complete the provided skeleton file following the tasks above.

Do **not** modify the lemmas and any uncommented code that is already provided in the skeleton files (unless we explicitly mention to do so). In particular, please do not change the lemma names, rule names, and theory names. Also, do not add comments within the provided sections.

You must provide the resulting Tamarin `.spthy` file. As in the previous lab, please name your model file `OIDC_<leginumber>.spthy`, where `<leginumber>` should be replaced by your matriculation (legi) number *without* dashes, for exam-

ple: `OIDC_15821606.spthy`

Tamarin must be able to process these files and prove or disprove each of the properties automatically, i. e. by running

```
tamarin-prover --prove <model>.spthy
```

Use the current release version 1.8.0 of Tamarin for you proofs.

Make sure that Tamarin does not raise warnings about wellformedness checks failing or any other errors.

As in the first lab, use the baseline test that checks for the above requirements. Please run the script on your solution before submission, as **we will not accept solutions that do not pass the baseline tests,** i.e. you will not receive any points for them.

You can find the script with the skeleton file at

`https://moodle-app2.let.ethz.ch/mod/folder/view.php?id=960173`.

You have to provide your path to Tamarin and the path to your submission to the script, i. e. by running:

```
./BaseLineScript <path-to-submission> <path-to-tamarin>
```

You can get the path to Tamarin via `which tamarin-prover`

If all tests pass, the script should output:

```
SUCCESS: You passed all the base line tests. You may submit your theory.
```

Otherwise you will get an error message.

Please submit your file on Moodle:

`https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=954340`

As usual, you may submit as many times as you want.

Evaluation

As stated above, your submissions must pass the baseline test. Submissions that do not pass this tests will not receive any points.

The grading will be fully automatic. You will receive points for the following tests:

10 points if the executability lemmas are successfully verified.

40 points if your lemmas pass hidden tests: we test them against different models.

50 points if your models pass hidden tests: we test them against different lemmas.

We will then make a weighted average with the three models of the first lab using the following weights:

- `Protocol1.spthy`: 0.15 ;
- `Protocol2.spthy`: 0.15 ;
- `Protocol3.spthy`: 0.20 ;
- `OIDC.sphty` : 0.5.

In addition, we will run anti-cheating tests and check for plagiarisms. Points might be subtracted for not following the task and protocol description.

Good luck!