Security Protocol Verification with Proverif

Laboratory for the class "Security Verification and Testing" (01TYASM/01TYAOV)
Politecnico di Torino – AY 2022/23
Prof. Riccardo Sisto

prepared by:
Riccardo Sisto (riccardo.sisto@polito.it)

v. 0.1 (21/10/2022)

Contents

Purpose of this laboratory

The purpose of this lab is to make experience with security protocol verification using Proverif.

For the proposed exercises you are invited to use the typed version of extended pi calculus (for which Proverif provides type checking). The full language reference can be found in the Proverif manual and Tutorial (`https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf`).

This is a summary of the commands to use Proverif:

- Run Proverif on a script file:

```
proverif filename
```

- Run Proverif on a script file and request Proverif to generate an attack trace graph in a given directory:

```
proverif -graph directory filename
```

- Run Proverif on a script file in interactive mode to perform a simulation:

```
proverif_interact filename
```

1 Verifying and fixing the sample handshake protocol

In this part of the lab we make some experiments with the sample handshake protocol that was discussed in the lectures.

1.1 Reproducing the results already shown in the classroom

First of all, in order to reproduce the same results shown in the lectures, let's run Proverif on the file handshake1cie.pv, which includes the description of the protocol and the following security queries:

1. secrecy of s
2. reachability of the end of each process (these are not security queries but sanity checks to verify that each process can really reach the end).
3. A to B authentication:

   ```
   query x:pkey,y:pkey,z:bitstring; inj-event(eB(x,y,z)) ==>
   inj-event(bA(x,y,z)).
   ```

   which means: if a B process with public key y correctly receives key z, apparently coming from an A process with public key x, then an A process with public key x really sent key z to the same process and each receive operation corresponds to a distinct send operation (injectivity).

From the Proverif verification report, you can see that properties 1 and 2 are satisfied (note that reachability is satisfied if the queries return false), while 3 is not. Indeed, the report specifies that the non-injective correspondence holds, i.e. only injectivity does not hold.

Run again Proverif with the switch -graph (see command at the beginning), which produces a graphical view of the attack trace found by Proverif and look at the trace. What is the behavior of the attacker in the attack trace?

→

1.2 Fixing the protocol

If we do want injectivity we need to fix the protocol. One way to do it is to add a preliminary step in the protocol, by which pB first sends a randomly generated value (a nonce, which acts as session id) to pA in order to request the key, and pA responds to this request by sending the same nonce together with the key. In this way, the attacker should not be able to replay the message with the key into a different session. Write a Proverif script that describes the fixed version of the protocol (you can start from the version in handshake1ci.pv and modify it). Note that the events of the correspondence have to be changed accordingly, by including the nonce as well, otherwise events belonging to different sessions cannot be distinguished.

After having described the fixed version, check that it passes the injective version of the authentication property (and that all processes reach the end). In case of problems, you can use the simulator to understand why.

Report the Proverif script with the fixed version of the protocol:

→

1.3 Verifying the authenticity and integrity of the secret

Now that the protocol has been fixed, we can try to verify that the secret sent by pB to pA is authentic and that integrity holds for it, i.e., whenever pA receives the secret, indeed the secret was sent by pB, and the correspondence is injective, i.e. each receive of the secret is preceded by a distinct send of the secret. Note that, in order to express this property, we need to introduce other events in the protocol description.

Report the Proverif script, modified to include the verification of this additional authentication property.

→

What is the result? Is the property already true or does the protocol need to be fixed to satisfy it?

→

1.4 Verifying the integrity of the secret when different secrets can be sent

Let's try to verify that the secret cannot be changed while in transit even when it is not always the same. In order to verify this property, we can modify the description of the processes, so that there is no longer a single secret, but two different secrets, s1 and s2. We can have then some pB sessions that send s1 and some that send s2. Finally, we verify that what is received corresponds (injectively) to what was sent.

Report the Proverif script, modified to include the verification of this additional integrity property.

→

What is the result? Is the property already true or does the protocol need to be fixed to satisfy it?

→

2 Verifying and fixing the sample hash protocol

In this part of the lab we make some experiments with the sample hash protocol that was discussed in the lectures.

2.1 Reproducing the results already shown in the classroom

First of all, in order to reproduce the same results shown in the lectures, let's run Proverif on the file hash.pv, which includes the description of the protocol and its secrecy properties:

1. secrecy of s
2. reachability of the end of each process (these are not security queries but sanity checks to verify that each process can really reach the end).
3. resistance to offline guessing of s.

From the Proverif verification report, you can see that properties 1 and 2 are satisfied (note that reachability is satisfied if the queries return false), while 3 is not. This means that, if s is assumed to be a secret with low entropy (i.e., one with few different possible values), it could be obtained by the attacker by means of an offline guessing attack.

Run again Proverif with the switch -graph, which produces a graphical view of the attack trace found by Proverif and look at the trace. What is the behavior of the attacker in the attack trace?

→

2.2 Fixing the protocol

If we assume the two processes pS (sender) and pR (receiver) share another high-entropy secret (e.g., a long randomly chosen number), there is a simple fix to this protocol, so that it resists to offline guessing attacks on s. Can you find it and verify that the solution is correct?

→

3 Defining and verifying a simple signature protocol

In this part of the lab we try to develop and verify the model of a simple protocol to guarantee the authenticity of a software issued and delivered by a software vendor.

We model the software file by means of a name (sw), and its associated metadata, such as software name, author, version, etc, by means of another name (swd). We use a process to model the issuer of the software, and we assume the issuer issues a software sw and its associated metadata swd protected by a signature (we assume the issuer has a private/public key pair). On the other side, we model the receiver of the software as another process that checks the signature before accepting the software with its metadata as authentic.

Assuming that the issuer issues several distinct software packages, each one including a software file and its corresponding metadata, we want to make sure that the receiver will accept as valid only packages identical to those issued by the issuer, and not fake software file nor fake metadata nor other combinations (e.g. right software file with wrong metadata).

Specify the protocol and the property described above in a Proverif script and try to verify it.

```
→
```