The University of Melbourne

SWEN90010: High Integrity Systems Engineering

Assignment 2

Due Date: 11:59pm Melbourne Time, Sunday May 3rd, 2020

Introduction

This handout is the Assignment 2 sheet. The assignment is worth 20% of your total mark. You will carry out the assignment in the same pairs as for Assignment 1, *unless you were allocated to a new pair in case your partner withdrew from the subject.*

In this assignment you will use Alloy to analyse the security of a simple, fictitious protocol for talking to a *PGP Key Server*. A PGP Key Server is an Internet server that manages a database storing which email encryption keys belong to which email addresses.

Suppose we have two users, Alice and Bob, who want to send each other emails that nobody else can read. To do that, Alice needs to encrypt the messages she sends to Bob using Bob's email encryption key. Bob needs to do the same: encrypt the emails he sends to Alice with Alice's encryption key. Alice and Bob can communicate securely as long as they each know which is the correct key for the other party. This is where the PGP Key Server comes in: its job is to help Alice and Bob find the correct key to use for each other.

When Alice wants to send an encrypted email to Bob, she can look up which encryption key she needs to use to encrypt her message by talking to the PGP Key Server. To do this, she sends a message to the key server asking it which encryption key is associated with Bob's email address, e.g. `bob@example.com`. The PGP Key Server replies by telling Alice which key to use to encrypt messages for `bob@example.com` so that only Bob can read them.

Alice and Bob can also send messages to the server to tell the server which encryption key is associated with their email addresses. For instance, Bob can talk to the PGP Key Server to tell it which key is associated with `bob@example.com`. This process is called *registering* a key with the PGP Key Server.

However, for the server to do its job properly, we need to be sure that when Bob tries to register his key for his email address `bob@example.com` that no attackers are able to fool the PGP Key Server into storing a different key. For example, if an attacker Eve is able to fool the PGP Key Server into associating Eve's key with `bob@example.com`, instead of Bob's key, then when Alice tries to look up Bob's key the PGP Key Server would mistakenly tell her Eve's key instead. Then Alice would mistakenly encrypt the secret messages using Eve's key instead of Bob's, which might allow Eve to read the messages that were meant for Bob!

Therefore, the registration protocol is security-critical. In this assignment you will analyse a proposed registration protocol, and use Alloy to diagnose potential vulnerabilities in the protocol and to help propose fixes to the protocol to help make it more secure.

Formal Security Analysis with Alloy

The scenario of the PGP Key Server is depicted in Figure 1. Here there is a *User* (e.g. Alice, Bob), which talks to the *PGP Key Server* via the *Network*. However the *Attacker* (e.g. Eve) is also on the network and can interfere with the communication between the User and the PGP Key Server.

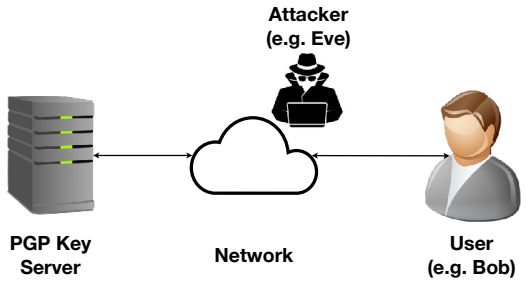

Figure 1: PGP Key Server Scenario

In this assignment we explore a particular attack scenario in which the attacker has certain abilities to interfere with the communication between the PGP Key Server and the User; however, the attacker's abilities are not unrestricted (otherwise it would be impossible to build a secure server).

User Key Registration

The protocol between the User and PGP Key Server (i.e. the sequence of steps followed by these parties) to register a key with an email address is as follows. Suppose Bob wants to register a key K with his email address `bob@example.com`.

Register Request Bob sends a *Register Request* message to the PGP Key Server, which includes his key K and email address `bob@example.com`.

Register Response When the server receives his request, it first checks that key K is not already registered. If not, it generates a *confirmation token* t. The token t is a long string of random bits. The server remembers Bob's email address `bob@example.com`, the key K and token t and stores these together as a triple: $(K, \texttt{bob@example.com}, t)$. However before it is sure that the association between Bob's email address and the key K is valid, it first needs to check with whoever controls the email address `bob@example.com`. To do that, it sends a *Registration Response* email message to the email address in the Register Request message, namely to `bob@example.com`. This email includes the token t.

Confirm Request To confirm the registration request, Bob then replies to the Registration Response message and sends the reply back to the PGP Key Server as a *Confirmation Request* message. Importantly, this message contains the token t from the Registration Response message. When the server receives this message, it knows that Bob received the token t. The server is now convinced that the registration of key K with Bob's email address `bob@example.com` is valid. So it updates the triple $(K, \texttt{bob@example.com}, t)$ to

replace the temporary token t with a special flag `CONFIRMED`, i.e. it updates this triple to be (K, `bob@example.com`, `CONFIRMED`).

Now when somebody queries which key is registered with the address `bob@example.com` the server will say that key K is registered, because the registration is confirmed.

The Alloy Model

You are provided with a partial Alloy model `pgp.als` of the PGP Key Server scenario, which you will extend. To keep the assignment tractable, the model of the network is simplified by assuming that the network can hold at most one message at a time.

The model includes signatures for the basic types: `Key` for encryption keys K, `Token` for confirmation tokens t, and `Identity` for representing email addresses like `bob@example.com`. The `MessageSubject` signature represents the different message types: `RegisterRequest`, `RegisterResponse`, and `ConfirmRequest`. The `Message` signature represents the messages: messages include an identity `addr : Identity`: for Request messages (sent *to* the PGP Key Server) `addr` is the identity of the sender; for `Response` messages (sent *from* the PGP Key Server) it is the identity of the intended receiver of the message.

Traces of system actions are captured using the `util/ordering` module described in lectures. *Hint: you may find it helpful to make use of the functions defined in that module when writing assertions and predicates for this assignment. A copy of the Alloy source for the module is available on the LMS with the assignment.*

Your Tasks

Task 1: Completing the Initial Model; Finding an Attack (10 marks)

Your first task is to complete the parts of the model that are missing, as well as to document some parts of the model by adding comments to the file. This will help you to understand the model and to practice your skills at interpreting formal specifications.

1. **[3 marks]** Add comments describing, in words, the pre- and postcondition of the `recv_register_request` predicate.
2. **[1 mark]** Add comments describing, in words, the pre- and postcondition of the `send_confirm_request` predicate.
3. **[3 marks]** The implementation of the `user_recv_register_response` predicate is missing. Complete its implementation by referring to the accompanying comments.
4. **[3 marks]** With the `user_recv_register_response` predicate implemented in accordance with its description in the comments, you can now use Alloy to find an attack on the protocol.

   In particular, you should now execute the "**check** `no_bad_states`" assertion at the bottom of the file. The counter-example returned by Alloy represents a real attack on the protocol as specified.

You should describe the attack in the comments above the check.

Note: If you execute the check of this assertion before properly implementing `user_recv_register_response`, Alloy will likely tell you that the predicate does not hold. However, this is because `user_recv_register_response` is unconstrained and so, until this predicate is properly implemented, the model allows behaviours that could not occur in reality. In other words, the counter-example given by Alloy won't represent a real attack on the protocol until you have properly implemented `user_recv_register_request`.

Task 2: Understanding the Attacker's Abilities (5 marks)

The second part of this assignment is to use Alloy's animation features to help understand the abilities of the attacker. These abilities are encoded in the `attacker_action` predicate, which models all actions of the attacker in the system. This will help you for the third part of the assignment, where you need to work out how to modify the protocol to remove the vulnerability found in Task 1.

Specifically, this task is to fill in the implementations for four different predicates. Each of these predicates is designed to encode when the attacker is able to perform a certain kind of action, i.e. they encode various abilities that the attacker may or may not have.

The potential ability that each predicate is meant to encode is described above it in comments, as well as below. You need to think about how to specify each ability, i.e. how to fill in the missing part of the implementation of each of these predicates. You also need to read and understand the provided implementation for `attacker_action`, to work out what behaviours it does and does not permit.

For each potential ability, if the attacker has this ability you should be able to get Alloy to generate an instance of that behaviour via the **run** command (once you have implemented the predicate, of course). In this case, annotate the **run** statement with "**expect** 1". Otherwise, annotate it with "**expect** 0", to indicate that no instance is found by Alloy and that you believe that the attacker does not have this ability.

Note: naturally you will need to experiment with different bounds for these **run** commands: just because Alloy cannot find a satisfying behaviour with a particular bound, does not mean that the behaviour is impossible in general. Try with larger bounds until you are convinced whether the behaviour is possible or not.

The abilities and predicates are as follows:

Drop Messages : Encoded by the `attacker_can_drop` predicate, this predicate holds between states `s` and `s'` when `attacker_action[s,s']` holds and a message is on the network in `s` but not in `s'`: i.e. it holds when the attacker has the ability to remove messages from the network.

Modify Messages : Encoded by the `attacker_can_modify_messages` predicate, this predicate holds between states `s` and `s'` when `attacker_action[s,s']` holds and each state has a message on the network that is different to the message in the other state: i.e. it holds when the attacker has the ability to modify a message that is on the network.

Forge IDs : Encoded by the `attacker_can_forge_ids` predicate, this predicate holds between states `s` and `s'` when `attacker_action[s,s']` holds, and `s'` has a network message whose

`addr` field is `UserId` but there was no `UserId`-message on the network in state `s`: i.e. when the attacker can send a message onto the network that appears either like it was sent from the User, or that it was sent from the server to the User.

Inject Messages : Encoded by the `attacker_can_inject` predicate, this predicate holds between states `s` and `s'` when `attacker_action[s,s']` holds, and `s'` has a network message but `s` does not: i.e. the attacker can cause a new message to appear on the network.

These abilities are not non-overlapping: clearly if the attacker can Forge IDs they might also be able to Inject Messages. The purpose here is to help you understand what is allowed by the attacker model and what is not.

Task 3: Fixing the Protocol (5 marks)

Now that you have a better idea of what abilities the attacker has, your job is to work out how to fix the vulnerability discovered in Task 1, to make the protocol secure against the attacker model encoded in `attacker_action`.

Specifically, your job is to modify one or more of the *Message Predicates* defined in the file, and then to update any other parts of the model accordingly. These predicates define what valid messages of each type look like, e.g. what information they need to contain.

You are not allowed to add new messages. However you can add extra information to existing messages. You don't need to add any new signatures or data types. You are allowed to modify other parts of the protocol in response to changes you make to the message predicates, e.g. modifying the `user_recv_register_response` predicate you implemented earlier. You should document any updates that you make by adding appropriate comments.

You are *not* allowed to alter the attacker's abilities: i.e. do not modify `attacker_action`. The fixed protocol should be secure against the supplied attacker model.

You should discuss the vulnerability together in your pair and carefully think about why this vulnerability exists in the original protocol, and how the protocol can be strengthened to remove it and make the protocol secure against the attacker model.

At the very bottom of the file, document your changes to the protocol by adding comments describing how you fixed the vulnerability.

Maximum marks will be awarded here for the *simplest* fixes: i.e. if you manage to fix the protocol while making as few changes to the protocol as possible. Your fixed protocol should be secure against the attacker model encoded in the `attacker_action` predicate.

Academic Misconduct

The University misconduct policy applies to this assignment. Students are encouraged to discuss the assignment topic, but all submitted work must represent the pair's understanding of the topic.

The subject staff take plagiarism very seriously. In the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.

Submission

Submit your Alloy file using the link on the subject LMS.

Only *one* student from the pair should submit the solutions, and each submission should clearly identify **both** authors.

Late submissions Late submissions will attract a penalty of 10% (2 marks) for every day that they are late. If you have a reason that you require an extension, email Toby *well before the due date* to discuss this.

Please note that having assignments due around the same date for other subjects is not sufficient grounds to grant an extension. It is the responsibility of individual students to ensure that, if they have a cluster of assignments due at the same time, they start some of them early to avoid a bottleneck around the due date.