The University of Melbourne

SWEN90010: High Integrity Systems Engineering

Assignment 2

Due Date: 11:59pm Melbourne Time, Friday April 19th, 2024

1 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 diagnose and investigate a fix for a recent vulnerability in the the popular Secure Shell (ssh) protocol. This protocol allows a *client* to connect to a *server* and to execute shell commands on the server and observe their output. The protocol provides security, including authenticating the server to the client (so the client knows they are talking to the intended server) and protecting the client/server communication by encrypting it, preventing adversaries from being able to observe and tamper with the shell commands/responses sent between the client and server. The protocol is also designed to provide additional guarantees, explained below (see Section 1.6).

The vulnerability you are investigating is real[1]. But we will consider a simplified model of this protocol for this assignment that captures the *essence* of the vulnerability sufficient to allow you to investigate a fix for it. The vulnerability in question is in the SSH Binary Packet Protocol. This protocol exchanges various messages between the client and the server that authenticate the server to the client and negotiate encryption keys by which further messages are encrypted. However, for this assignment we will consider a simple protocol whose intended sequence of interactions is depicted in Fig. 1.

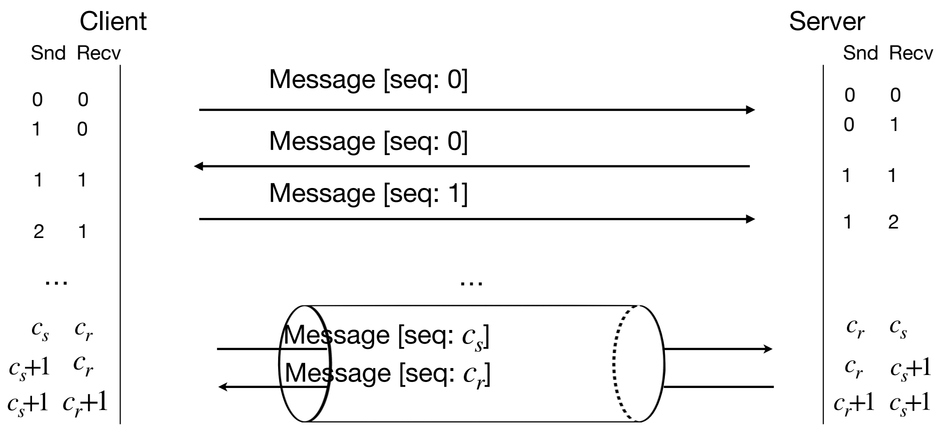

Figure 1: Example intended interactions in the protocol.

[1]https://terrapin-attack.com/

1.1 Protocol Phases

The protocol has two phases: the *insecure* phase followed by the *secure channel* phase. During the insecure phase, messages are exchanged between the client and the server to negotiate parameters needed to set up a secure channel between the client and the server. Once this channel has been set-up, the protocol enters the secure channel phase. This phase is depicted by the exchange of messages via the cylinder in Fig. 1. The cylinder represents a secure channel between the client and the server. The messages exchanged prior to the establishment of the secure channel form the insecure phase of the protocol.

We assume the existence of a potental *attacker* who sits between the client and the server and attempts to break the security of the protocol. The purpose of negotiating the secure channel is to ensure that messages exchanged during the secure phase of the protocol cannot be read or modified by the attacker. However, messages exchanged during the insecure phase of the protocol can be read or tampered with by the attacker. Exactly *how* the secure channel works is not important for this assignment.

Also not important to this assignment is the precise sequence of messages that need to be exchanged in the insecure phase to negotiate the secure channel. Instead we assume at some point the protocol transitions from the insecure phase to the secure channel phase (see Section 1.5).

1.2 Messages and Sequence Numbers

Each message sent between the client and the server contains the following pieces of information:

Sequence number: An ascending number used to order messages in a session, explained below.

Source: The identity of the sender of the message.

Destination: The identity of the receiver of the message.

Data: The actual message contents, which are not especially important for the purpose of this assignment.

Each party A running the protocol remembers, for each other party B who they might communicate with, how many messages they have sent to B and how many messages they have received from B respectively. We call these respectively A's *send count* and *receive count* for B. B maintains its own send and receive counts for A. These counts are depicted for the client and server in Fig. 1. All counts naturally start at 0. When party A sends a message to party B, A increments its send count for B. When B receives a message from A, B increments its receive count for A. B can receive a message from A only when the message's destination is B (and its source is A) and the message's sequence number matches B's receive count for A (the message's sender).

When party A sends a message to party B, the message contains a *sequence number* which is simply A's send count for B before the message was sent. Thus the first message A sends to B has a sequence number of 0; the second, a sequence number of 1, and so on.

1.3 Sending and Receiving Messages

We assume the existence of a network that connects the client and the server. Sending a message places it onto the network. Receiving a message removes it from the network. Thus for party A to successfully communicate a message to party B, the following needs to occur: first, A needs to *Send* the message, which places it on the network; then once the message is on the network, party B then needs to *Receive* the message.

1.4 Attacker Actions

While the message is on the network, the attacker might try to interfere with it. Depending on what phase the protocol is in, the attacker may be able to interfere more or less. Specifically, during the insecure phase, the attacker can modify network messages arbitrarily, or inject new messages onto the network pretending that they were sent by another party. During all phases of the protocol, the attacker can remove messages from the network. This is the only thing they can do in the secure channel phase of the protocol.

1.5 Transition from Insecure to Secure Channel Phase

As stated above, we choose to ignore the precise details of how the secure channel is negotiated in the protocol. To abstract away from these details, we allow at any time for the parties to the protocol to decide to transition to the secure channel phase. This can only be done when there are no messages on the network (as otherwise, attacker generated messages already on the network could spoil the guarantees provided by the secure channel). When this happens, *all* parties transition to the secure phase simultaneously[2]. This helps to simplify modelling the protocol (see Section 2 below). You will explore the implications of this design choice later in this assignment.

1.6 Security Goal

The real ssh protocol is designed to provide integrity and confidentiality for the messages exchanged in the secure channel phase of the protocol. For the purpose of this assignment, we will *assume* that the protocol provides these standard security guarantees and instead focus our attention on a further specific security goal.

The security goal is that once the protocol enters the secure channel phase, messages sent from a party A to party B have to be received in order. In particular, if A sends a message m_1 to B during this phase and later sends a message m_2 to B, then B can receive m_2 only if it has already received m_1 while the protocol was in the secure channel phase.

---

[2]Even those who were not involved in negotiating the secure channel!

2 The Formal Model

You are provided with a partial Alloy model of this protocol. Part of your tasks for this assignment is to complete this model using the description above and the following information.

The Alloy model formally describes the state of all participants in the protocol. We refer to a participant as a *Principal*. Messages are sent between Principals. Each `Message` contains a *source* `src` and a *destination* `dest` principal, as well as a *sequence number* `seq_num` and some opaque `data`.

```
sig Principal {}

sig Data {}

sig Message {
  seq_num : one Int,
  src : one Principal,
  dest : one Principal,
  data : one Data
}
```

The Alloy model defines a `State` type that records the send and receive counts of each principal for all principals, plus some other information described below.

```
abstract sig ChannelState {}
one sig Insecure, Secure extends ChannelState {}

abstract sig DebugAction {}
one sig SendAction, RecvAction, AttackerAction, GoSecureAction
  extends DebugAction {}

one sig State {
  // send_counter[from][to] records the number of messages that Principal
  // 'from' has sent to Principal 'to'
  var send_counter : Principal -> Principal -> one Int,
  // recv_counter[to][from] records the number of messages that Principal
  // 'to' has received from principal 'from'
  var recv_counter : Principal -> Principal -> one Int,

  var channel_state : one ChannelState,
  // the network can hold at most one message at a time
  var network : lone Message,
  // for debugging purposes (to aid reading counter-examples, etc.)
  // records the most recent action that has occurred
  var debug_last_action : lone DebugAction
}
```

The Alloy model purposefully abstracts away from the details of how the protocol implements the secure channel. To further simplify the model, once the protocol enters the secure channel phase, the attacker loses various abilities to interfere with messages not only between the principals who negotiated the secure channel but also with all other principals too. This greatly simplifies modelling the attacker actions while still providing just enough detail to allow reasoning about the attack on the protocol's security goal. Therefore, the model simply remembers whether the protocol is in the `Insecure` or the `Secure` `channel_state`.

The Alloy model also describes the state of the `network`, specifically by remembering the last message that was sent on the network.

The `State` also provides a field `debug_last_action` to remember the last action that was performed: either Sending a message on the network (`SendAction`), Receiving a message from the network (`RecvAction`), the attacker interfering with the network (`AttackerAction`), or the principals deciding to transition from the insecure to the secure channel phase (`GoSecureAction`). The `State` remembers the last action performed to make it easier to read the counter-examples and traces produced by Alloy.

Sending a message places it onto the network. To simplify things, sending can occur only when there is no message already on the network. The message has to have a sequence number that is equal to the sender's send counter for the receiver at the time the message is created. Sending the message increments the sender's send counter for the receiver. This is done in Alloy using the `add` function.

Hint: remember in Alloy that `1+2` is the set {1,2}. To add two integers together, we write `add[1,2]`.

```
// sender 'from' sends a message to receiver 'to' with sequence number
// 'seqnum' and data 'd'
pred Send[from, to : Principal, seqnum : Int, d : Data] {
  no State.network
  (some m : Message | m.src = from and m.dest = to and m.seq_num = seqnum
    and seqnum = State.send_counter[from,to] and m.data = d
    and m in State.network')
  State.send_counter' = State.send_counter ++ (from -> to -> (add[seqnum,1]))
  State.recv_counter' = State.recv_counter
  State.channel_state' = State.channel_state
  State.debug_last_action' = SendAction
}
```

Receiving a message can occur only when there is a message on the network. The sequence number in the message has to be equal to the receiver's receive count for the sender—otherwise, the Receive cannot occur. Receiving a message increments the receiver's receive count for the sender and removes the message from the network. It does not affect which phase the protocol is in, i.e., receiving a message on its own cannot transition the protocol from the insecure phase to the secure channel phase.

One of your first tasks is to complete the definition of the Receive action, `Recv`:

```
// receiver 'to' receives a message from sender 'from' with sequence number
// 'seqnum' and data 'd'
pred Recv[from, to : Principal, seqnum : Int, d : Data] {
  // FILL IN HERE
}
```

Transitioning from the insecure phase to the secure channel phase simply updates the relevant part of the `State`. As stated above, it can happen only when there is no message on the network.

```
// models the establishment of a secure channel
pred Go_Secure {
  no State.network
  State.channel_state = Insecure and State.channel_state' = Secure
  State.send_counter' = State.send_counter
  State.recv_counter' = State.recv_counter
  State.network' = State.network
  State.debug_last_action' = GoSecureAction
}
```

The attacker has certain abilities to interfere with the network depending on what phase the protocol is in (i.e., depending on the `channel_state`). See Section 1.4. Part of your job will be to formally specify these abilities by completing the definition below.

```
pred Attacker_Action {
  // FILL IN STUFF HERE to define how the attacker can affect State.network'

  State.send_counter' = State.send_counter
  State.recv_counter' = State.recv_counter
  State.channel_state' = State.channel_state
  State.debug_last_action' = AttackerAction
}
```

The initial state of the protocol has all send and receive counters as 0, with the protocol in the insecure channel phase, and no message on the network.

```
// the initial state
pred Init {
  State.send_counter  = Principal -> Principal -> 0
  State.recv_counter  = Principal -> Principal -> 0
  State.channel_state =  Insecure
  no State.network
  no State.debug_last_action
}
```

The actions that can occur in the system are either Sending or Receiving a message, or an Attacker Action, or the protocol transitioning to the secure channel phase, or doing nothing. These are the *only* actions that can occur in each state.

```
pred State_Transition {
  (some from, to : Principal, seqnum : Int, d : Data |
    Send[from,to,seqnum,d] or Recv[from,to,seqnum,d])
  or
  Go_Secure
  or
  Do_Nothing
  or
  Attacker_Action
}

pred Do_Nothing {
  State.send_counter' = State.send_counter
  State.recv_counter' = State.recv_counter
  State.network' = State.network
  State.channel_state' = State.channel_state
  no State.debug_last_action'
}

// constrain traces
fact traces {
  Init and
  (always State_Transition)
}
```

3 Your Tasks

3.1 Task 1: Completing, Understanding, and Diagnosing the Vulnerability in the Initial Model (14 marks)

Your first task is to complete the parts of the model that are missing. Doing so will require you to carefully understand the model you are given so you can work out how to fill in the missing parts.

*You should not add any additional **sig** declarations nor modify any of the existing **sig** declarations during this assignment. You should not add or remove parameters to the **pred** declarations during this assignment.*

1. [**5 marks**] Complete the `Recv` predicate (4 marks) and the `Attacker_Action` predicate (1 marks) to complete the initial model.

2. [**2 marks**] Notice in Fig. 1 how just after whenever a message is received, the receiver's send and receive counters for the sender mirror the sender's for the receiver. Should this always be the case? Write an assertion `in_sync_always` that captures this property and asserts that it is always true. Write a **check** command to check whether your assertion is true in general. You should find that it does *not* hold in general. Write a comment below your assertion to explain why it doesn't hold in general.

3. [**1 mark**] Think about when this property *should* hold. Write a second assertion `in_sync` that is similar to your `in_sync_always` assertion but only requires that property to hold under suitable conditions. Validate your assertion with a **check** command. Explain *why* it holds with a comment below the **check** command.

4. [**6 marks**] Use Alloy to discover the vulnerability in the protocol. Write an assertion `security_goal` that encodes the protocol's security goal (see Section 1.6). Write a **check** command with a suitable bound to get Alloy to find a counter-example to this property.

   Hint: you may like to make use of Alloy's past-time *temporal logic operators, to help write your property. See `https://alloytools.org/alloy6.html`.*

   Hint: the vulnerability means that the protocol is subject to a prefix truncation attack*. If you are wondering whether you have found the "right" vulnerability, you might like to try to familiarise yourself with this concept, see e.g. Figure 2 of `https://terrapin-attack.com/TerrapinAttack.pdf`.*

   Describe the vulnerability in comments above your **check** command that produces the counter-example to this assertion. That is, describe what is wrong with the original protocol and how the vulnerable behaviour arises.

Task 2: Fixing the Model (6 marks)

1. [**3 marks**] The actual fixes for this vulnerability that vendors have implemented in reality

include sequence number resets, as well as modifying the protocol to prevent the attacker injecting messages during the insecure phase of the protocol.

Focusing just on resetting the sequence numbers, think about *when* the sequence numbers should be reset to remove the vulnerability you found above. Then modify the appropriate predicate in the Alloy model so that sequence numbers will be reset to 0 at that time.

Add a comment *to the bottom* of your Alloy file describing how you implemented this fix and what part you changed.

You should not need to add any extra messages, **sig**s, or any extra information to any messages to implement the fix for the protocol as modelled here. Neither should you add any new parameters to any existing predicates when implementing your fix.

2. [**1 mark**] Show that your `security_goal` assertion now holds. Use a suitably high bound when carrying out this check.

   Add comments justifying / explaining your choice of bound and, specifically, what guarantees you think are provided by this check.

   To obtain the mark here we want to see you think critically about the bound you have chosen and what guarantees you think it provides, i.e., what behaviours are covered by that bound and why showing the absence of the attack for all of those behaviours provides assurance about the fixed protocol.

   If you think that your chosen bound does not provide good guarantees you should say so and explain why.

3. [**2 marks**] Finally, consider the limitations of how your Alloy model has been designed. In particular, recall how we simplify the model of the protocol so that once it enters the secure channel phase for *any* principals, it effectively provides the secure channel for *all* principals (see Section 1.4). Think about what sorts of attacks or vulnerabilities might still be present in a protocol like this that would not be detected because of this simplification. Add comments to the bottom of your Alloy file to explain this and justify your conclusions.

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.