LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

INSTITUTE FOR INFORMATICS
Software and Computational Systems Lab

Information Flow testing of a PGP server

Bachelorarbeit

BEARBEITER: Lukas Johannes Rieger
BETREUER: Prof. Dr. Gidon Ernst

Dokument erstellt
16. Juli 2020

Statement of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, drawings, maps and images included in the thesis.

Place and date

Signature

Information Flow testing of a PGP server

Abstract

Lorem Ipsum

Contents

1 Introduction

[austin2009efficient] [BanerjeeNR08] [callas1998openpgp] [chudnov2014information] [chudnov2018assuming] [claessen2011quickcheck] [Goguen˙Meseguer˙82] [Lourenco˙Caires˙15] [myers2001jif] [Zheng˙Myers˙07] [CVE˙2019]

2 Structure of a PGP server

2.1 Traditional approach

2.2 Secure approach

2.3 The HAGRID keyserver

3 Methodology

3.1 Tools and technology used

3.1.1 Property based testing with ScalaCheck

3.2 Testing dynamic declassification policies

The following chapters cover the approach chosen to test the information flow and declassification of PGP keys in the abstract server model as well as the actual HAGRID implementation.

4 Practical implementation in Scala

4.1 The abstract server model

In order to test the previously illustrated privacy constraints of PGP keys, we require a model of a PGP server which is sufficiently abstract to hide irrelevant implementation details, yet still exposes all functional specifications of the HAGRID server.

A minimal representation of a secure PGP server according the the HAGRID specifications would therefore require the following operations:

- *Request adding a key* - Request adding a new key to the database. This key may potentially contain an arbitrary amount of associated identities. The uploaded key is not made publicly available immediately after uploading. Instead, the server issues a confirmation code that may be used in subsequent confirmation requests.

- *Confirm an addition* - Confirm a previously uploaded `(Identity, Key)` pair given a valid confirmation code. This action does not directly confirm the selected identity but instead issues a confirmation email. The confirmation is only completed if the user uses the code contained within the email.

- *get(ByMail, ByFingerprint, ByKeyId)* - Retrieve a key from the database given some identifying index (e.g.the fingerprint of the PGP key or one of the associated identities).

- *Request a deletion* - Request the removal of some previously confirmed `(Identity, Key)` pair. This action does not directly delete the selected identity but instead issues a confirmation email. The removal is only completed if the user uses the code contained within the mail.

- *Confirm a deletion* - Finalize the removal of some `(Identity, Key)` pair. This operation requires the confirmation code, which the user must have obtained from a confirmation mail issued by the previous operation.

This minimal set of requirements allows for a relatively direct translation into source code. Our implementation in Scala bases its general server structure on the following trait definition:

```
trait HagridInterface {
    def byEmail(identity: Identity): Option[Key]
    def byFingerprint(fingerprint: Fingerprint): Option[Key]
    def byKeyId(keyId: KeyId): Iterable[Key]
    def upload(key: Key): Token
    def requestVerify(from: Token, emails: Set[Identity]): Seq[Body]
    def verify(token: Token)
    def requestManage(identity: Identity): Option[EMail]
    def revoke(token: Token, emails: Set[Identity])
}
```

Where the datatypes `Fingerprint` and `KeyId` represent the concepts as defined in the OpenPGP message format according to [**callas1998openpgp**]. The `Identity` type serves as a simple wrapper for a string containing an email address. `Key` is a type that can generally be defined as:

```
    trait Key {
        def keyId: KeyId
        def fingerprint: Fingerprint
        def identities: Set[Identity]
    }
```

thus combining the previously mentioned types into what our server model will treat as a representation of a full PGP key.

The implementation of this trait that we chose to use as a basis for our tests is non permanent. Specifically, the model state is kept as a set of maps whose elements transition between each other:

```
    class Server extends HagridInterface {
        var keys: Map[Fingerprint, Key]
        var uploaded: Map[Token, Fingerprint]
        var pending: Map[Token, (Fingerprint, Identity)]
        var confirmed: Map[Identity, Fingerprint]
        var managed: Map[Token, Fingerprint]
        ...
    }
```

Changing the internal server state Any of the actions exposed by the `HagridInterface` (see definition in section **??**) may modify these internal state maps. For example, we can inspect the implementation of the `upload` action:

```
    def upload(key: Key): Token = {
        val fingerprint = key.fingerprint

        if (keys contains fingerprint)
        assert(keys(fingerprint) == key)

        val token = Token.unique
        keys += (fingerprint -> key)
        uploaded += (token -> fingerprint)
        token
  }
```

As we can see, we modify both `keys` map, as well as `uploaded` map. For a complete overview of the remaining implementation, please refer to the file `Server.scala` under `src/main/scala/pgp` in the attached Scala project.

Invariants Internally, we defined a number of consistency invariants to ensure that our server model always behaves correctly. Any state manipulating action exposed by the server will trigger a re-evaluation of said invariants.

1. A key is valid if its fingerprint is registered in the `keys` map.

   ```
   for ((fingerprint, key) <- keys) {
   assert(key.fingerprint == fingerprint)
   }
   ```

2. Upload tokens must refer to a valid fingerprint that itself refers to a valid key (see invariant **??**)

   ```
   for ((token, fingerprint) <- uploaded) {
       assert(keys contains fingerprint)
   }
   ```

3. Pending validations must always refer to a valid key. Additionally, all pending validations must be for identity addresses that refer to the respective key.

   ```
   for ((token, (fingerprint, identity)) <- pending) {
       assert(keys contains fingerprint)
       val key = keys(fingerprint)
       assert(key.identities contains identity)
     }
   ```

4. All confirmed identity addresses must refer to valid keys. Additionally, all confirmed identities must be valid for the associated key.

   ```
   for ((identity, fingerprint) <- confirmed) {
     assert(keys contains fingerprint)
     val key = keys(fingerprint)
     assert(key.identities contains identity)
   }
   ```

5. All issued management tokens must refer to valid keys.

   ```
   for ((token, fingerprint) <- managed) {
       assert(keys contains fingerprint)
     }
   }
   ```

4.2 Simulating actions and responses

The server model itself cannot execute any relevant actions without some accompanying actor that causes this action. The following sections will explore two different approaches to simulating user actions, that were considered during development.
In general, we wanted to express the following abstract actions in a composable way:

- *Upload some key* ***k*** *to the server*
- *Verify some identity* ***i*** *in relation to some parent key* ***k***
- *Revoke some verified identity* ***i*** *from its parent key* ***k***
- *Request some key* ***k*** *from the server, using one of its identifying characteristics*

Notice how these operations closely align with those defined by the server model.

4.3 Initial approach to simulating actions through "actors"

Our initial approach to simulating user actions was based on fine grained *actors* that each encapsulated a specific capability. Using this approach, any of the aforementioned operations would be represented by a single actor that runs on a simple state machine. The actor trait is defined as following:

```
trait Actor {
    def canAct: Boolean
    def act(): Unit

    val inbox = mutable.Queue[Data]()
    def canReceive = inbox.nonEmpty

    def isActive = canAct || canReceive

    def handle(from: Actor, msg: Message)
    def handle(from: Actor, msg: Body)
    def send(to: Actor, msg: Message): Unit =
        Network.send(this, to, msg)
    def send(to: Identity, msg: Body): Unit =
        Network.send(this, to, msg)

    def register(identity: Identity): Unit =
        Network.register(identity, this)
}
```

An actor could define a custom `act()` method that would initialize the specific action of this actor. Any future interaction with another actor would then be handled by the actors `handle()` method.

TODO

: Describe why there were two different handle methods. What is Network? What was bad about this approach? Show execution strategy and why order of execution was a problem.

4.4 Arbitrarily sized histories of events

To overcome the complications of our initial approach, specifically the inability to define the *execution order* in a simple and expressive way, we were forced to reconsider the usage of actors in terms of user actions.

As an alternative, we introduce the notion of a sequential `History` of events that represent interactions between a user and the PGP server. Through this method we can define a high level model of *what* operations should be executed in a specific order while keeping the *how*, i.e. the concrete execution strategy, completely separate.

Once again, the different kinds of events are closely related to the capabilities exposed by our server model. Specifically, we recognise three distinct event types, that are modelled as subclasses of a `sealed trait Event`:

```
case class Revoke(ids: Set[Identity], fingerprint: Fingerprint)
    extends Event

case class Upload(key: Key) extends Event

case class Verify(ids: Set[Identity], fingerprint: Fingerprint)
    extends Event
```

Given some history of arbitrary length, we can now define a simple execution strategy that sequentially runs the given history and sends the corresponding data the server. The signature of the execution algorithm is given by `def execute(server: HagridInterface, history: History): Unit`. The method then simply loops through

4.5 History evaluation strategies

Given some history h we want to determine precisely which associations between identities and keys should be visible depending on the combination of events in the history. To determine the visibility of an identity at any time, we introduce a `State` type, that tags the current state of an identity throughout the evaluation of the history. The state of an identity may take any of these three forms:

- Public: The identity has been confirmed and therefore publicly visible
- Private: The identity has not been confirmed yet and therefore remains private
- Revoked: The identity had been confirmed at some prior point in the history but has since been revoked.

In terms of visibility, a `Private` identity and a `Revoked` identity can be considered equal from the perspective of a user. In both cases, the identity should not be included in any key requests.

In order to determine these states for every possible identity, we define a method `def states: Map[Fingerprint, Set[(Identity, Status)]]` on `History`. The method starts off with three separate maps `Uploaded`, `Confirmed` and `Revoked`. These maps are then populated by folding of the sequence of history events. Depending on the currently folded over value, one or several of the maps will be updated.

This explanation is currently pretty bad and unfinished.

Maybe illustrate this with a diagram? Visualize the maps and how a event can change one or more of them.

5 Testing the real-world implementation of HAGRID

Given that we had successfully developed a working testing framework for our abstract model, we decided to integrate the actual HAGRID server into our testing efforts. For obvious reasons, testing HAGRID requires a local server instance to be running on the same machine.

Communicating with HAGRID The main effort of integrating HAGRID into our implementation focussed on the relaying and receiving of messages between our Scala frontend and HAGRID. To this end, we use *Sttp*, which is a simple library that enables us to communicate with the server through REST-full http calls. The transferred data is being encoded as *JSON*, for which we rely on *Circe* as a library.

HAGRID itself defines a specific interface, through which external applications may communicate with the PGP server. This interface, called *Verifying Keyserver (VKS) Interface*, exposes the following endpoints:

- `GET /vks/v1/by-fingerprint/FINGERPRINT`
- `GET /vks/v1/by-keyid/KEY-ID`
- `GET /vks/v1/by-email/URI-ENCODED EMAIL-ADDRESS`
- `POST /vks/v1/upload`
- `POST /vks/v1/request-verify`

The interface which actually communicates with the server exposes exactly the same methods as our model, due to it implementing the same `HagridInterface`.

Instead of keeping track of keys and identities in the frontend, our `HagridServer` simply relays all actions to HAGRID by http. As an example, lets look at the `upload` method once again:

```
  override def upload(key: Key): Token = {
      val response = basicRequest
          .post(hag("/vks/v1/upload"))
          .body(UploadBody(key.armored))
          .response(asJson[UploadResponse])
          .send()
          .body

      Token(response
      .toOption
      .get
      .token)
}
```

First, we construct a simple *POST*-Request which targets the *VKS* endpoint responsible for uploading PGP keys. We then attach the *armored* key-text as the request body and finally instruct the request to encode any eventual response as an instance of `UploadResponse`. The most crucial bit of information included in the `UploadResponse` is the token, which we require in order to send any verification requests at a later point.

However, it must be noted that our implementation, while fully working, generally *ignores* the possibility of an error occurring while communicating with HAGRID. We are aware that this may cause unforeseen problems in the future.

5.1 Receiving verification mails from HAGRID

While we were able to simulate the transmission of verification mails in an abstract and direct way for our server model, we had no such option when dealing with the real-world HAGRID server.

During the earlier development phases, we assumed that we would be forced to provide a dummy implementation of `sendmail`, as the documentation claimed that HAGRID would require a working `sendmail` command to be available in the environment in order to send and receive mails. Luckily, we discovered that HAGRID also provides an alternative mode of operation, in which all email transfer is handled directly through the filesystem. Modifying the configuration file `Rocket.toml` and adding the line `mail_folder=/PATH/TO/MAIL`, causes HAGRID to write all mails to a unique file in the specified folder.

This allows us to simply observe the given directory and react to any file modifications within the folder. More specifically, we defined a method, that would block its execution until it successfully read all expected mails:

```scala
  private def consumeMail(expectedSize: Int): Seq[Body] = {
      val watchKey = mailWatcher.take
      val events = watchKey.pollEvents.asScala
      val updatePaths = events
          .take(expectedSize)
          .map(_.asInstanceOf[WatchEvent[Path]])
          .map(_.context())

      val mails = updatePaths.flatMap { currentPath =>
          val resolved = mailPath.resolve(currentPath).toFile
          val source = Source.fromFile(resolved)
          val bodies: Seq[Body] = parseMail(source.mkString)
          source.close
          resolved.delete
          val isValid = watchKey.reset
          bodies
      }
      mails
}
```

where the returned sequence of mail bodies takes the form of:

```
case class Body(fingerprint: Fingerprint, token: Token, identity: Identity)
```

Parsing plain text confirmation mails In order to consume the emails within the selected directory and actually use the information contained within them, we had to parse the plain text content of said mails and extract the relevant information. In both cases of identity confirmation or revocation, these relevant bits of data would be the `Fingerprint`, the `Token` and finally the respective `Identity` in concern.

For example, a typical mail issued by HAGRID (running as a local instance) in repsonse to a confirmation request might look like this:

```
To: <ISSUE_EMAIL@ADDRESS>
Hi,

Dies ist eine automatische Nachricht von localhost.
Falls dies unerwartet ist, bitte die Nachricht ignorieren.

OpenPGP Schlüssel: 23B2E0C54487F50AC59134C3A1EC9765D7B25C5A

Damit der Schlüssel über die Email-Adresse "<ISSUE_EMAIL@ADDRESS>" gefunden werden kann
klicke den folgenden Link:

http://localhost:8080/verify/vrTohvV8q552KMvARBE7foqkvGtUrfDl3iiyX9yqeOX

Weitere Informationen findest du unter http://localhost:8080/abo
```

This mail should be in english as well.

We currently use a regular expression to parse the relevant information from these mails. This functionality is summarized in a method `parseMail`:

```
val VERIFY_PATTERN: Regex =
    "(?s).*To: <(\\S+)>.*OpenPGP key: ||
    (\\S+).*http://localhost:8080/verify/(\\S+).*<!doctype html>.*".r

def parseMail(mail: String): Seq[Body] =
    decode[HagridMail](mail)
        .map(mail => new String(mail.message))
        .map {
        case REVOKE_PATTERN(identity, fingerprint, token) =>
            Body(FingerprintImpl(fingerprint), Token(token), PgpIdentity(identity))
        case VERIFY_PATTERN(identity, fingerprint, token) =>
            Body(FingerprintImpl(fingerprint), Token(token), PgpIdentity(identity))
        }
        .toSeq
```

As seen in the code sample in **??**, this function is responsible for parsing the mails, that `consumeMail` read from the dedicated mail directory.

5.2 Generating valid PGP keys

While our testing efforts were mainly focussed on our own abstract model, we were able to use a simplified representation of our Key datatype to represent a PGP key.

```
sealed trait Key {
    def armored: String
    def keyId: KeyId
    def fingerprint: Fingerprint
    def identities: Set[Identity]
    def restrictedTo(ids: Set[Identity]): Key
}
```

For example, keyId simply consists of a unique string identifier that does not hold any specific meaning:

```
sealed trait KeyId {
    def value: String
}
case class KeyIdImpl(id: String) extends KeyId {
    def value: String = id
}
object KeyId {
    def random: KeyId = KeyIdImpl(UUID.randomUUID().toString)
}
```

In order to transmit any Key values to the actual HAGRID server, we couldn't rely on the existing dummy implementation, as HAGRID would reject any PGP key that is not properly formatted.

In order to solve this problem, we decided to implement a mechanism that is capable of generating valid PGP keys. For this purpose, we use *BouncyCastle*, which is a Java library for cryptographic use cases.

The general signature to generate a PGP key is as following:

```
def genPublicKey(identities: Set[Identity]): (PGPPublicKey, String)
```

It should be noted though that we currently do not support the generation of keys with arbitrary amounts of identities. This is due to several unsolved problems which we encountered during development. In the end, these hurdles led to the decision that temporarily supporting a maximum amount of only *two* identities is acceptable.

5.3 Technical challenges

One major challenge that we faced when trying to incorporate the actual HAGRID implementation in our testing approach was the fact that HAGRID maintains internal state across several distinct instances. This is due to HAGRID persisting uploaded keys and their contents within its installation folder. In contrast to this, our high level model of HAGRID maintains no state at all between any two separate test runs.

This is problematic, because our testing approach fundamentally relies on a limited amount of *Identities*, that are subsequently used to construct PGP Keys. This would necessarily lead to unexpected results when applying our approach to HAGRID, where there could potentially already be an identity associated to some PGP key, that was uploaded during a previous testing session. To elaborate on this problem, let us assume these two separate test runs in pseudocode:

Assuming that there exists a value `key1 = Key(identity1,identity2)` and a value `key2 = Key(identity3,identity4)` where all arguments to the `Key` constructor are instances of `Identity` with an arbitrary email address, we can define the following two test runs:

```
upload(key1)
verify(identity1)
```

Test run 1

```
verify(identity1)
upload(key2)
```

Test run 2

Looking *only* at test run 2 we would expect an outcome, in which the server returns no public keys/identities at all. Instead, when we actually execute these two runs in sequential order, we receive a PGP key when querying the server with `identity1`, even though we *never* uploaded it during the second test. Any solution to this problem therefore requires us to reset the internal state of HAGRID after having completed a test. This in turn also imposes an additional non trivial performance penalty on the execution speed of our ScalaCheck test, besides the slowdown caused by having to generate real PGP keys. For our current solution to this problem, we decided to compile HAGRID into a static image. This allows us to create a fresh instance of HAGRID at the beginning of each test iteration and shutting it down afterwards, at which point we simply clear the local directory of PGP keys. This ensures that each test run starts on a truly equal playing field.

6 Discussion

7 Related Work

8 Conclusion