LMU 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
18. September 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

[1] [2] [3] [4] [5] [6] [8] [9] [10] [13] [11]

2 Structure of a PGP server

2.1 Traditional approach

2.2 Secure approach

2.3 The HAGRID keyserver

3 Information Flow Theory

The following sections give a brief introduction to the general notion of *security* in regards to the concept of information in computational systems. Building upon this, we then focus specifically on the key concepts of non-interference and declassification as they relate to the notion of information flow.

3.1 Security models and policies

Many modern computational systems will at some point have to handle certain kinds of information, which must never be accessible to an *unauthorized* entity. A popular description for a system, which keeps its information from leaking to the outside or any unintended entity/process can be found in the term of a *secure* system. In this context, "security" or "secure information flow" means, that "no unauthorized flow of information is possible" [7].

While such a general definition may be relatively easy to arrive at, actually *enforcing* the necessary constraints on the numerous information pipelines and flows within even a single program is often no easy task at all. In order to properly define which information may flow in what way within a part of a program or between programs, a common approach is to specify the individual components, between which a flow of information may occur and consequently assigning certain security *levels* to these individual constituents. This approach can for example be found in [7] by Denning, in which the concept of *security classes SC* is introduced. These classes can then be bound to any information receptacles or "objects" in order to denote their respective security clearance.

This chapter is way too short at the moment.

3.2 Non-interference and Declassification

An important concept in terms of secure systems that can be operated by multiple users either simultaneously or sequentially, is the concept of *non-interference*.

> One group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see

Mention, that the lattice model is very widespread and fundamental. Tie into this later aswell. (Public/Private can be seen as a binary lattice model.)

To explain this concept in terms of our problem domain of PGP servers, let us assume running instance of HAGRID which is publicly accessible. Let us further assume that there are two users $u1$ and $u2$ that are interacting with the server at the same time. Additionally, each user holds a certain set of keys:

$$u1 := \begin{cases} k1 = \text{Key(id1,id2)} \\ k2 = \text{Key(id3,id4)} \end{cases} \quad \text{and} \quad u2 := \begin{cases} k3 = \text{Key(id5,id6)} \\ k4 = \text{Key(id7,id8)} \end{cases}$$

Based on this, we can image the following, highly simplified interaction between the two users and HAGRID

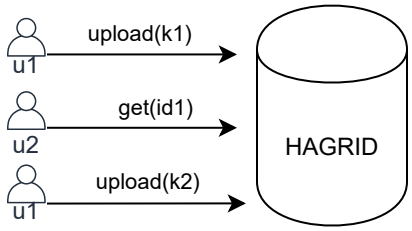

Figure 1: A non-interfering interaction

The second call of $u2$ in this sequence is a key request for identity $id1$ which belongs to the previously uploaded key $k1$. Yet, because the identity had not been verified when $u2$ issued its key request, the resulting response will be empty. That means, that the actions of $u1$ hat no effect *at all* on the experience of $u2$. As a matter of fact, based on the interactions with HAGRID alone, $u2$ has no way of identifying whether there even *is* another user currently interacting with the same system. In contrast to that, let us assume that this sequence of interactions is continued in the following way:

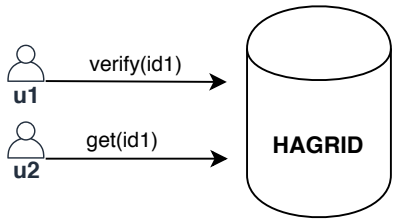

Figure 2: The interaction has become interfering

The second *get* request now yields a different response as before. Instead of an empty response, $u2$ now receives a key with an identity. This change in behaviour was caused by $u1$ issuing a verification for $id1$. Therefore, the actions of $u1$ were able to influence the sequence of interactions between $u2$ and HAGRID, which causes this interaction to no longer be *non-interfering*.

Given this definition of non-interference, it can be trivially concluded that a strict enforcement of this principle would render most information systems practically unusable. Instead, what we need is a formal mechanism to determine the circumstances under which the release of classified information is permissible. As mentioned in [12], one key concern of any system that permits the declassification of private information is, whether that release is actually *safe*. That means, whether the introduction of additional declassification policies would allow an attacker to obtain more information from the given system than intended by its designers.

This safety concern has given rise to a multitude of different approaches, which aim at formally verifying the correct flow and release of information. Some of these approaches will be briefly laid out in chapters 8 and *REF TODO*.

However, the central approach of this thesis will specifically focus on the testing of *dynamic* information flow policies.

TODO: Why is this too strict? Examples. TODO: How does declassification work?

Quote Security Policies and Security Models for their definition of non-interference

4 History based security specifications

Our approach must specifically be able to encode and evaluate *dynamically* changing privacy constraints on a certain subset of relevant information. Additionally, these pieces of information are also dynamic and constantly changing while our system under test is actively operating. This poses several challenges in regards to how we can specify these constraints over the whole body of information beforehand. Instead we have decided to utilise a *history* based encoding scheme, that takes the sequential nature of privacy changes over several time steps into account.

4.1 What constitutes a history

We rely on a central data structure, a sequential *history* of abstract events, that each encode a high level operation depending on the system under test. Formally, we can thus define any history h as

$$h_n = \{ev_1, ev_2, \ldots, ev_n\},$$

where n denotes the size of the given history and any ev_i represents the i-th operation or *event* in the history. Given that the history itself is in no way responsible for the way in which it is interpreted, we impose no requirements on the internal structure of the event type.

4.2 Generating histories for HAGRID

Given our general definition of a history, we can now look at the content of histories that we use to test our model of HAGRID. As described in chapter 2.3, we want to be able to describe the process of verifying a given subset of valid identities belonging to a PGP key and similarly revoking these identities in a similar fashion. Note that these two operations, *verifying* and *revoking* identities are the central operations, that drive the dynamic declassification and reclassification of private data in our server. We therefore define 3 abstract types of events, that can be included in a history:

1. *Upload(k)* – Given some key k, this event describes the action of adding a fresh PGP key to the internal state of our server.

2. *Verify(ids,fingerprint)* – Given a set of identities, this event describes the action of *declassifying* these formerly private identities, given that they belong to the key identified by the corresponding *fingerprint*

3. *Revoke(ids, fingerprint)* – Given a set of identities, this event describes the action of *reclassifying* these formerly public identities, given that they belong to the key identified by the corresponding *fingerprint*

4.3 How to evaluate histories

Given a history of some arbitrary size, we now require a method of inspecting the sequence of events and compute the *effect* that each event in the history has on the privacy state space of our server. Specifically, our goal is to run a symbolic execution of a history, which ultimately computes the totality of valid visibility associations between keys and their identities. Note, that this symbolic history execution doesn't have to match the actual execution process, because we can omit the server model as our middle man and compute the privacy state for all keys directly.

A formal description of history evaluation Evaluation of arbitrary histories can be formally approximated by a set of state machines S_i. Each state machine tracks the visibility state of its associated identity i.

In order to show how this formalization works, let us first look at a simpler case, where we only consider histories of events that affect a single key with only one associated identity.

We can then define a state machine S which is a quintuple $(\Sigma, Q, s_0, \delta, F)$, where:

- Σ is the input alphabet given by the set $\{\text{Upload}, \text{Verify}, \text{Revoke}\}$, corresponding to our history event types;
- Q is the finite set of states $\{\Theta, U, C, R\}$;
- s_0 is the initial state of any identity, denoted as S_0;
- δ is the state transition function, defined by the following state transitions:

$$\begin{aligned} \Theta &\xrightarrow{\text{Upload}} U \\ U &\xrightarrow{\text{Verify}} C \\ C &\xrightarrow{\text{Revoke}} R \\ R &\xrightarrow{\text{Verify}} C \end{aligned} \tag{1}$$

- F is the empty set of final states

The states of this state machine represent the privacy state after each transition, where a transition to U means, that the given identity has been uploaded, yet remains undiscoverable until further state changes. The C state occurs, when the given identity gets verified and therefore publicly available, whereas the R state describes the exact opposite, the given identity is private must first be re-verified to become available again.

We can now generalize this state machine to support histories containing arbitrary amounts of keys and identities. Let's consider the set of state machines

$\{S_{i,k}\}$, where all state machines share their set of states and initial state with S.

Given some event $E_{i,k}$ and some state $s \in Q$, we define $\exists s' \subseteq \delta_{i',k'}(E_{i',k'}, s)$ if, and only if $\exists S_{i',k'}.\, i' = i \wedge k' = k$. Finally, it is easy to see, that any event $E(t, f)$, where t is a set of identities can also be denoted as the set $E' = \{E_{t_i,f} | t_i \in t\}$. That means, that any event that affects multiple identities at once can be equivalently formulated as a sequence of the same event that each affects a single event from the original set.

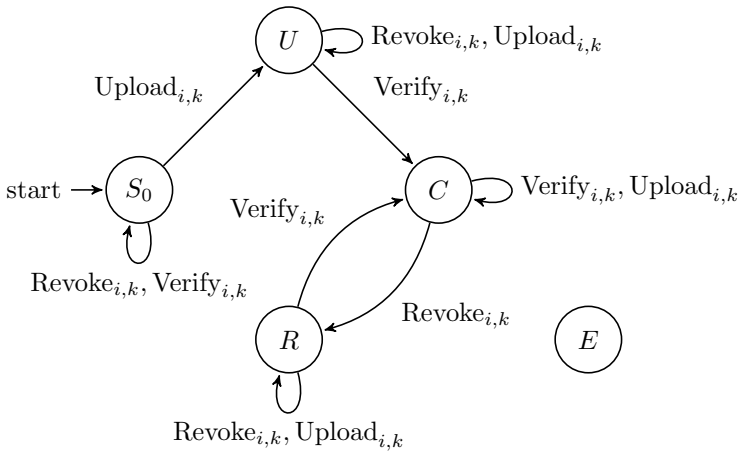

Figure 3: Visualization of a state machine over a history

In order to emphasise which states support a successful key request, we added an additional state E, which is only reachable by transitions labeled with $\text{Get}_{i,k}$. We can think of E as a kind of implicit error state, which tells us, that the system encountered an attempt to retrieve a key k, while the state machine associated with k was still in a state, that made returning a key impossible.

It is easy to see, that this can only happen, if the state machine is either in its initial state S_0, or all state machines $t_{k,i}$ for a given k are currently in the state U.

Reformulating the example runs from chapter 3.2 using state machines Looking at our previous history examples from chapter 3.2 we can see quite clearly, that our encoding of histories in terms of state machines gives us a surprisingly natural representation of our privacy states at every intermediate step of our history. Let $h_1 = \{Upload(k1), Get(id1), Upload(k2)\}$ and $h_2 = \{Upload(k1), Get(id1), Upload(k2), Verify(id1), Get(id1)\}$.

We can now visualize the both history runs accordingly, by showing the respective state transitions of their state machines at each step:

The first step of evaluating a history requires us to divide the data contained in the history into three distinct maps U, C and R, where U holds an entry for every uploaded key mapping the fingerprint to the key itself. C then maps each identity which remains *confirmed* at the end of the history to the fingerprint of its parent key. R on the other hand, maps all identities which remain revoked to its parent key.

Formally, these three sets can now be defined as:

$$\begin{aligned} U =& \{(k.fingerprint, k) \mid k \in H(\text{Upload})\} \\ C =& \{(i, fingerprint_i) \mid i \in H(\text{Verify}) \land \phi(i, H_n)\} \\ R =& \{(i, fingerprint_i) \mid i \in H(\text{Revoke}) \land \neg\phi(i, H_n)\} \end{aligned} \tag{2}$$

where ϕ is defined as

$$\phi(i, H_n) = \exists e_t \in H(\text{Verify})(i \in e.ids \land (\forall e_k \in H(\text{Revoke})k < i)) \tag{3}$$

Ultimately, we want to obtain a map, that associates the fingerprint of every uploaded key to a set of (`Identity`,`Status`) tuples. In this case, `Status` indicates the visibility of the associated identity, which may take either of the following three values:

- *Public* – The identity has been declassified and can be made publicly available.
- *Private* – The identity remains classified and must therefore not be leaked by the server.
- *Revoked* – The identity has been reclassified after having been confirmed at some prior point in the history.

TODO: formalize withUploaded, etc.

History evaluation against a server model Determining which associations between identities and keys are valid based on a history alone is only the first step of actually *testing* HAGRID. To complete our history based approach, we require a way of comparing the results of the symbolic history execution to the actual server state. Only then can we determine, whether the model adheres to our chosen privacy constraints.

The solution to this requirement consists of two parts in total. As a first step, we need to *execute* the history on a given server backend. In a second step, we can then collect all available data from the server and compare the results to our previously computed visibilities.

In order to visualize the execution of a history, we can look at the following example:

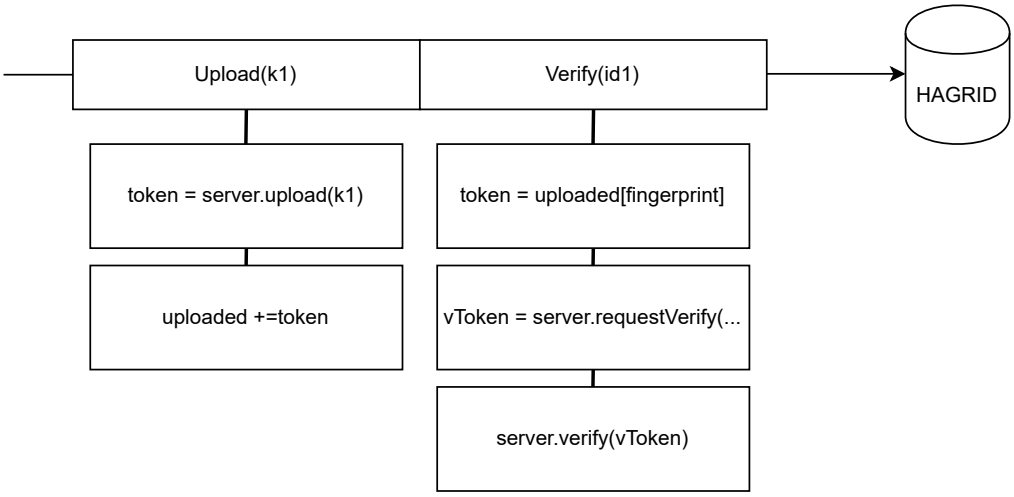

Figure 4: An example of a history execution

More examples here!

Notice, how `Verify` abstracts over the two separate steps of *requesting* a verification and the subsequent actual verification step. **TODO** The algorithm for testing HAGRID can then be summarized as:

1. Using all keys contained in the given history, send a key-request to HAGRID for every key k
2. Collect all returned identities in a map that associates each fingerprint with the corresponding set of identities: fingerprint $\rightarrow$ identities
3. For all keys contained in either the given history, or the server response: Compare the symbolic execution result $\{(\text{Identity}, \text{Status})\}$ with the matching server response $\{\text{Identity}\}$.

Determining invalid associations Focusing on the last step, all that is left now is to compute all differing entries. Given that the format of both structures is very similar, this process is very straight forward:

First we introduce a custom result type that encapsulates the two possible outcomes of comparing $(\text{Identity}, \text{Status})$ tuples with a given identity. If the comparison shows that both identities have equal visibility states, we return an instance of the `Ok` type. Otherwise, we return a `Mismatch` instance, which contains further information regarding the specific *kind* of mismatch we encountered.

For this purpose we can define the following function $d(s_i, h_i)$):

$$d(s,h) = \begin{cases} \text{Ok} & \text{if } s = \epsilon \wedge h = \epsilon \\ \text{Ok} & \text{if } s = \epsilon \wedge h_2 = \text{Private} \\ \text{Ok} & \text{if } s = \epsilon \wedge h_2 = \text{Revoked} \\ \text{Ok} & \text{if } s \neq \epsilon \wedge h_2 = \text{Public} \\ \text{Mismatch} & \text{if } s \neq \epsilon \wedge h = \epsilon \\ \text{Mismatch} & \text{if } s = \epsilon \wedge h_2 = \text{Public} \\ \text{Mismatch} & \text{if } s \neq \epsilon \wedge h_2 = \text{Private} \\ \text{Mismatch} & \text{if } s \neq \epsilon \wedge h_2 = \text{Revoked} \end{cases}$$

where ϵ denotes the absence of a value.

5 Practical implementation in Scala

5.1 Tools and technology used

5.1.1 Property based testing with ScalaCheck

5.2 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 [3]. 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 5.2) 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 1)

```
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)
  }
}
```

5.3 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* $\boldsymbol{k}$ *to the server*
- *Verify some identity* $\boldsymbol{i}$ *in relation to some parent key* $\boldsymbol{k}$
- *Revoke some verified identity* $\boldsymbol{i}$ *from its parent key* $\boldsymbol{k}$
- *Request some key* $\boldsymbol{k}$ *from the server, using one of its identifying characteristics*

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

5.4 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.

5.5 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

5.6 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.

6 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 RESTfull 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.

6.1 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:

```
  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 6.1, this function is responsible for parsing the mails, that `consumeMail` read from the dedicated mail directory.

6.2 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.

6.3 Initializing

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.

7 Discussion

8 Related Work

9 Conclusion

References

[1] Thomas H Austin and Cormac Flanagan. "Efficient purely-dynamic information flow analysis". In: *Programming Languages and Analysis for Security (PLAS)*. 2009, pp. 113–124.

[2] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. "Expressive Declassification Policies and Modular Static Enforcement". In: *Security and Privacy (S&P)*. IEEE, 2008, pp. 339–353.

[3] Jon Callas et al. *OpenPGP message format*. Tech. rep. RFC 2440, November, 1998.

[4] Andrey Chudnov, George Kuan, and David A Naumann. "Information flow monitoring as abstract interpretation for relational logic". In: *Computer Security Foundations Symposium (CSF)*. IEEE. 2014, pp. 48–62.

[5] Andrey Chudnov and David A Naumann. "Assuming You Know: Epistemic Semantics of Relational Annotations for Expressive Flow Policies". In: *Computer Security Foundations Symposium (CSF)*. IEEE. 2018, pp. 189–203.

[6] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *ACM sigplan notices* 46.4 (2011), pp. 53–64.

[7] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056.

[8] Joseph Goguen and José Meseguer. "Security Policies and Security Models". In: *Security and Privacy (S&P)*. Oakland, California, USA: Computer Society, 1982, pp. 11–20.

[9] Luísa Lourenço and Luís Caires. "Dependent Information Flow Types". In: *Principles of Programming Languages (POPL)*. ACM, 2015, pp. 317–328.

[10] Andrew C Myers et al. *Jif: Java information flow*. 2001.

[11] Red Hat, Inc. *CVE-2019-13050: Certificate spamming attack against SKS key servers and GnuPG*. https://access.redhat.com/articles/4264021. Accessed: 2020-06-07. 2019.

[12] A. Sabelfeld and David Sands. "Dimensions and principles of declassification". In: July 2005, pp. 255–269. ISBN: 0-7695-2340-4. DOI: 10.1109/CSFW.2005.15.

[13] Lantian Zheng and Andrew C. Myers. "Dynamic security labels and Static Information Flow Control". In: *International Journal of Information Security* 6.2–3 (2007).