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. Juni 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

[Austin and Flanagan, 2009] [Banerjee et al., 2008] [Callas et al., 1998] [Chudnov et al., 2014] [Chudnov and Naumann, 2018] [Claessen and Hughes, 2011] [Goguen and Meseguer, 1982] [Lourenço and Caires, 2015] [Myers et al., 2001] [Zheng and Myers, 2007] [Red Hat, Inc, 2019]

2 Background

2.1 Structure of a PGP server

2.1.1 Traditional approach

2.1.2 Secure approach

2.2 The HAGRID keyserver

2.3 Information Flow Theory

2.3.1 Security models and policies

2.3.2 Non-interference and Declassification

3 Methodology

3.1 Tools and technology used

3.1.1 Property based testing with ScalaCheck

3.2 General structure of the testing approach

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.

3.2.1 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 [Callas et al., 1998]. 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]
    ...
}
```

TODO: Should I explain exactly how upload/manage etc. modify the server state ?

3.2.2 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*

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

Our test approach is based on *Histories* that record events of the interaction explicitly, but not the internal states of the stateful actors. These histories are created by custom ScalaCheck generators with respect to a small collection of known email addresses. Three types of events can occur: upload of a key, and confirmation resp. revocation of the association between an email address and a key (via its fingerprint that is supposed to be unique).

The top-level specification is expressed over histories, which precisely determines those associations that are intended to be visible to another user of HAGRID (i.e. the adversary). To test whether an actual execution is correct and secure we thus compare the intended status of each *potential* association from the predetermined set of identities and keys to the actual observations that can be made by calling server operations.

3.2.3 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.

3.2.4 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

3.2.5 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.

TODO: This explanation is currently pretty bad and unfinished. **TODO**: Maybe illustrate this with a diagram? Visualize the maps and how a event can change one or more of them.

3.3 Testing the real-world implementation of HAGRID

4 Discussion

5 Related Work

6 Conclusion

List of Figures

List of Tables

References

[Austin and Flanagan, 2009] Austin, T. H. and Flanagan, C. (2009). Efficient purely-dynamic information flow analysis. In *Programming Languages and Analysis for Security (PLAS)*, pages 113–124.

[Banerjee et al., 2008] Banerjee, A., Naumann, D. A., and Rosenberg, S. (2008). Expressive declassification policies and modular static enforcement. In *Security and Privacy (S&P)*, pages 339–353. IEEE.

[Callas et al., 1998] Callas, J., Donnerhacke, L., Finney, H., and Thayer, R. (1998). OpenPGP message format. Technical report, RFC 2440, November.

[Chudnov et al., 2014] Chudnov, A., Kuan, G., and Naumann, D. A. (2014). Information flow monitoring as abstract interpretation for relational logic. In *Computer Security Foundations Symposium (CSF)*, pages 48–62. IEEE.

[Chudnov and Naumann, 2018] Chudnov, A. and Naumann, D. A. (2018). Assuming you know: Epistemic semantics of relational annotations for expressive flow policies. In *Computer Security Foundations Symposium (CSF)*, pages 189–203. IEEE.

[Claessen and Hughes, 2011] Claessen, K. and Hughes, J. (2011). QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM sigplan notices*, 46(4):53–64.

[Goguen and Meseguer, 1982] Goguen, J. and Meseguer, J. (1982). Security policies and security models. In *Security and Privacy (S&P)*, pages 11–20, Oakland, California, USA. Computer Society.

[Lourenço and Caires, 2015] Lourenço, L. and Caires, L. (2015). Dependent information flow types. In *Principles of Programming Languages (POPL)*, pages 317–328. ACM.

[Myers et al., 2001] Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. (2001). Jif: Java information flow.

[Red Hat, Inc, 2019] Red Hat, Inc (2019). Cve-2019-13050: Certificate spamming attack against sks key servers and gnupg. `https://access.redhat.com/articles/4264021`. Accessed: 2020-06-07.

[Zheng and Myers, 2007] Zheng, L. and Myers, A. C. (2007). Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3).