Information Flow Testing of a PGP Keyserver (Abstract for the VerifyThis challenge 2020)

Gidon Ernst and Lukas Rieger

LMU Munich, gidon.ernst@lmu.de

1 Abstract

We report on the progress in the VerifyThis long-term challenge 2020[1] that targets a formal analysis of the secure PGP keyserver HAGRID.[2]

A major concern in the design of the PGP keyserver HAGRID is that confirmed identities in published keys are authenticated and confidential otherwise. The underlying security property requires reasoning about *non-interference* [7], *value-dependent* information flow [10,8], as well as *declassification* [2]. We present a high-level reference model, written in Scala, that is detailed enough to capture these key concerns, but abstracts away from the internals of the server. Functional correctness and information flow security of the model is currently being analyzed using automated random testing via ScalaCheck, a variant of QuickCheck [6]. A comparison of behavior against the implementation in HAGRID is planned for future work.

2 Models

We have modeled the server, the client, the communication, and the attacker.[3] Scala is a programming language that supports both functional and object-oriented concepts, so that adequate choices regarding the abstraction level of data types and the encoding of state transitions can be made. The added benefit is that models can be executed and debugged interactively within an IDE.

Data Model: In contrast to other preliminary work in the challenge,[4] we model keys explicitly as an immutable data type that stores a set of abstract identities (i.e., email addresses) [3, Sec 5.11]. This aspect is relevant, because regardless of which identities were originally uploaded with a given key, only the subset of confirmed identities should be visible in the results returned by queries to the server.

Components: The client and the server are modeled as stateful classes which expose their functionality a set of operations. The server interface closely resembles that of the HAGRID API[5] and consists of operations for lookup,

[1] https://verifythis.github.io
[2] https://gitlab.com/hagrid-keyserver/hagrid
[3] https://github.com/gernst/verifythis2020
[4] https://verifythis.github.io/2019-09-10-eventb/
[5] https://keys.openpgp.org/about/api

requests for email validation and management access, and finally confirmation and deletion of associations between keys and identities. Internally, the server stores all keys every uploaded, the association between confirmed identities and keys, as well as the necessary bookkeeping for authenticated access in terms of tokens. A client object, on the other hand, stores a set of keys alongside those tokens received by the server, in order to be able to execute requests (e.g. in unit tests or randomly).

Communication: HAGRID uses two channels to communicate with users: a web-based interface for the lookup, upload, request for validation of keys, and management of these; and regular email for authentication tokens that are needed for certain operations. Each channel is modeled as an unordered collections of messages of the respective type, containing sender/receiver information. These messages are distributed by glue code in an actor-based approach, where the association between a client and its mailbox is explicit.

Adversary: We assume that all communication is secure, i.e., both channels are encrypted. The adversary has ordinary access to all operations of the server but we assume that he/she cannot guess authentication tokens. As a consequence, the adversary has access to all information that is supposedly public, unless of course the server leaks secret or unconfirmed information.

3 Test Approach

We consider two kinds of properties of interest: functional correctness and security. In our setting, correctness means primarily that uploaded keys can be looked up successfully. Security, on the other hand, expresses that no secret or unconfirmed information is visible to the adversary.

We use property-based testing via ScalaCheck, a tool that enumerates inputs according to a generator schema, runs the system under test, and checks properties of the resulting states. For functional correctness, we might specify, for example, that a key can be successfully downloaded from the keyserver after an upload; and that certain previously confirmed identities in the key are present.

Dynamic monitoring of information flow is a little trickier. There are some existing approaches, such as tagging runtime values [1], and security type systems, such as Jif [9] for Java, and hybrid monitors that employ logical reasoning [4].

However, we aim to test non-interference directly: Starting out with a pair of initial states that are indistinguishable by the adversary, the system is secure if any subsequent pair of states after some interaction steps is still indistinguishable. This semantic criterion is parametric in the model of the adversary, and gives fine-grained control over what is deemed secret or public information, based on the current state or in terms of the history of the execution. The flexibility is desirable, because the association between keys and confirmed email addresses is dynamic and changes over time with declassification of the association between identities and keys once a key is confirmed. Recent knowledge-based models of security show that indeed, declassification can be conflated with assumptions on

attacker knowledge (i.e., the declassified bit of information), such that a test run can simply be aborted when such an action happens [5].

The current state of the testing efforts is quite preliminary but we expect to make substatial progress soon. As a future step, we aim to cross-validate the model against the actual implementation, e.g., by replaying "interesting" traces from the model against the real server code.

References

1. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Programming Languages and Analysis for Security (PLAS). pp. 113–124 (2009)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: Security and Privacy (S&P). pp. 339–353. IEEE (2008)
3. Callas, J., Donnerhacke, L., Finney, H., Thayer, R.: OpenPGP message format. Tech. rep., RFC 2440, November (1998)
4. Chudnov, A., Kuan, G., Naumann, D.A.: Information flow monitoring as abstract interpretation for relational logic. In: Computer Security Foundations Symposium (CSF). pp. 48–62. IEEE (2014)
5. Chudnov, A., Naumann, D.A.: Assuming you know: Epistemic semantics of relational annotations for expressive flow policies. In: Computer Security Foundations Symposium (CSF). pp. 189–203. IEEE (2018)
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM sigplan notices **46**(4), 53–64 (2011)
7. Goguen, J., Meseguer, J.: Security policies and security models. In: Security and Privacy (S&P). pp. 11–20. Computer Society, Oakland, California, USA (1982)
8. Lourenço, L., Caires, L.: Dependent information flow types. In: Principles of Programming Languages (POPL). pp. 317–328. ACM (2015)
9. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow (2001)
10. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. International Journal of Information Security **6**(2–3) (2007)