Faculty of Economic Sciences, Communication and IT
Department of Computer Science

Anders Ellvin
Tobias Pulls

Implementing a Privacy-Friendly Secure Logging Module into the PRIME Core

Degree Project of 30 ECTS credit points
Master of Science in Information Technology

Date/Term:	2010-01-22
Supervisor:	Hans Hedbom
Examiner:	Donald F Ross
Serial Number:	E2010:03

Implementing a Privacy-Friendly Secure Logging Module into the PRIME Core

Anders Ellvin

Tobias Pulls

This thesis is submitted in partial fulfillment of the requirements for the Master’s degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Anders Ellvin

Tobias Pulls

Approved, 2010-01-22

Opponent: Andreas Lavén

Advisor: Hans Hedbom

Examiner: Donald F Ross

Abstract

When individuals access services online they are often required to disclose excessive amounts of personally identifiable information, with little to no transparency on how the information is used[28, 3]. One of the goals of the EU research project PrimeLife is to help people regain control of their private sphere in today's networked world[3]. As part of PrimeLife a software prototype, named the PRIME Core, is being developed that contains a number of different privacy enhancing technologies. This thesis describes the implementation and integration of a privacy-friendly secure logging module into the PRIME Core. The logging module's purpose is to provide transparency logging to the PRIME Core, giving individuals access to a detailed log of how their disclosed personally identifiable information is used, in a secure and privacy friendly manner.

The thesis resulted in a privacy-friendly secure logging module being implemented into the PRIME Core. The client for the logging module still lacks features to be suitable for use by the Data Track. Further research is needed to make the implementation mitigate the risks posed by memory and disk forensics.

Acknowledgements

We would like to thank our supervisor Hans Hedbom for giving us the opportunity to pursue this interesting thesis topic and the productive discussions throughout our work. To Peter Hjärtquist and Andreas Lavén, thank you for your work in the Topics in Computer Security course which laid the foundation for our work.

Contents

List of Figures

Chapter 1

Introduction

Today's networked world has negative consequences on privacy. Individuals are required to disclose excessive amounts of personally identifiable information to access services with little to no transparency on how the information is used[28, 3]. To enable people to verify that custodians of personally identifiable information are behaving in accordance with agreed upon policies and laws, some form of event log is needed to keep track of the processing and access to this information. Such a log, working together with other privacy enhancing technologies, could greatly aid people in regaining control over their private sphere.

The purpose of this thesis is to implement an event log into a software prototype, named the PRIME Core, being developed as part of the EU research project PrimeLife. The event log is a privacy-friendly secure logging module.

1.1 The Goals of the Thesis

At the start of the thesis a primary and a secondary goal was set. Both goals were completed as part of the work in this thesis.

1.1.1 Primary Goal - Implementation

Our primary goal was to fully implement and integrate our log into the PRIME Core. While writing the paper[28] describing the design of the privacy-friendly secure log a number of changes were made which are incorporated throughout this thesis. We also needed to implement the parts missing from the initial prototype, most notably the negotiation of secrets and the API, which were only touched upon superficially in the design.

1.1.2 Secondary Goal - Database Implementation Affecting Linkability

The seconday goal was to investigate if the database used (HSQLDB) to some extent reveals the order in which the log entries were added to the database, due to for example some internal structure or functionality.

1.2 Disposition and Timeline

In chapter 2 the EU research project and prototype are presented together with an overview of a number of cryptographic primitives used when creating secure logs. The chapter ends with a look at the Kelsey-Schneier[36] secure log. Chapter 3 describes the work that took place prior to this thesis, which were the design and proof of concept implementation of a privacy-friendly secure logging module. The proof of concept prototype is used as a foundation for our implementation done in this thesis. Chapter 4 describes the implementation work done in detail divided into three milestones. In chapter 5 we put all of our work into context highlighting, describing and evaluating the different components we have contributed to or modified in the EU research prototype. We end chapter 5 with evaluating our tests and look at the performance of our logging module. Chapter 6 concludes the thesis.

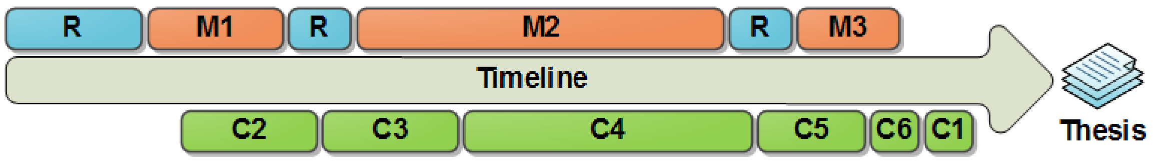


Figure 1.1: The timeline for our thesis showing when the research(denoted R), implementation milestones(prefixed with a M) and chapters(prefixed with a C) were worked on in relation to each other.

Figure 1.1 shows a timeline for our work with this thesis. The blocks above the timeline represents work related to the actual implementation while the blocks below the timeline shows roughly when each chapter of this thesis was written. At best one could say that we tried to follow the suggested thesis timetable, got behind on the writing and caught up at the end.

Chapter 2

Background

This chapter starts with presenting the EU research project and its software prototype to which the work in this thesis has been contributed to. Next a number of cryptographic primitives are briefly described that are needed to understand the secure logs presented later. The chapter ends with a look at the Kelsey-Schneier secure log.

2.1 PrimeLife

PrimeLife is an EU research project that started in March of 2008 as an extension to the PRIME project[3]. Currently the PrimeLife project has 15 participating institutes including IBM, Microsoft and several universities. The research surrounds the privacy-issues encountered when using today's connected technologies, with a goal to protect the privacy of a user when using Internet services and applications.

2.1.1 Bob's Got E-Mail

Let us take a look at an example; Bob wishes to create a new e-mail account at GrooveMail. When creating his new account he needs to enter, other than a username and password, his first and last name, the country he lives in, his postcode, gender and date of birth. The reason GrooveMail wants this much information is because they provide different services to Bob. On Bob's birthday they want to send out a birthday greeting from the staff at GrooveMail. They would also like to have his area of residence in order to provide e-mails with social events near him. It is unclear to Bob if he can later remove his postcode or date of birth and opt-out of their extra services. However, as Bob submits the form, one line at the bottom of the form states that, by submitting this information, Bob has accepted GrooveMail's privacy policy. Bob does not concern himself with this much because after all, he has done the same, many times, on other sites.

When looking at this example Bob might not be aware of the potential privacy-issues posed by the information he has provided. In order to protect users online it is necessary to create public awareness around the subject of privacy. Because of this reason the PrimeLife project has a goal to ensure that the majority of the community adopts privacy-enhancing technologies[3].

Another issue in this example is the fact that it would be difficult for Bob to figure out how his information is handled by GrooveMail, if he wanted to. Users should not have to spend a lot of time reading through the privacy policies when creating a new e-mail account, this would compromise the functionality of the service. Instead this site should clearly tell Bob how his information is handled, that is they should adopt better transparency. But it is not enough to tell the user how the information is handled, GrooveMail must also deliver on their promise.

The third issue when looking into this example is the amount of information Bob has to enter into the registration-form. Some of it, like his postcode, is not required for an e-mail account to function correctly. It is used for extra services that Bob might not want to use. This is why the PrimeLife project also want to incorporate data minimization into their solutions[3]. Data minimization means that only the information that is really needed should be requested and only for the amount of time it is needed for. This ensures the privacy of the user and also decreases the liabilities of the organization holding the information, in case an intruder would get access.

2.1.2 PRIME

PRIME, Privacy and Identity Management for Europe, is a project developing a prototype system to deal with the privacy-issues encountered by users online[2]. This kind of system is refereed to as IDM, Identity Management, which has the purpose of handling the identities of users and decides who has access to users' personal data at an organization. An enterprise IDM system usually focuses on the enterprise's needs and does not let users control their personal data. The prototype created by PRIME will show how an IDM system could work to provide users with control over their personal data.

The PRIME prototype is referred to as PRIME middleware because it resides between the service and the user's personal data[1]. Applications on an organization's server or a user's device will be able to use the PRIME middleware for tasks requiring user's personal data. The access to the PRIME middleware can follow one of two models, Transparent Interception or Explicit Access. The former can be useful for legacy applications which are not directly supporting PRIME since the communication between the applications and their databases are intercepted and handled by PRIME. The latter is support for new

PRIME-enabled applications. Users may also access the PRIME middleware by using the PRIME Console, which acts as the interface for the privacy-enhanced IDM system[2]. Some of the functionality available to users are[2]:

- Creating and managing multiple pseudonyms. A pseudonym is a handle used by a user to identify herself. By using different pseudonyms for different services there is less of a trail of a user's action online, instead there are smaller, unconnected trails.
- Gain a better understanding of privacy policies.
- Help users to negotiate and decide which information is to be sent to an organization and policies concerning how the information can be used, such as how long the information should be stored.
- Show a history of the user's transactions online.

With the use of PRIME middleware the registration of Bob's e-mail account at GrooveMail could have been altered to Bob's advantage. When the PRIME middleware is running on both Bob's machine and GrooveMail's servers they can negotiate how Bob's date of birth will be handled. He may choose to enter his date of birth for now and may later, by using the PRIME Console, choose to delete it from GrooveMail's database. If Bob would like to be even more private he can create a new pseudonym which he uses exclusively for the GrooveMail-service. That way his personal data is less likely to be connected to other personal data from other services. Bob can also verify that GrooveMail has used his personal data according to the policies set negotiated by GrooveMail and himself.

2.1.3 The PRIME Core

The PRIME middleware is divided into two main parts; the core and the optional. The core contains the required functionality for the IDM system to exchange personal data while enforcing the policies and preferences set. The optional part adds more advanced functionality. Figure 2.1 shows an abstract view of the PRIME middleware. The components surrounded by discontinued lines are the optional parts.

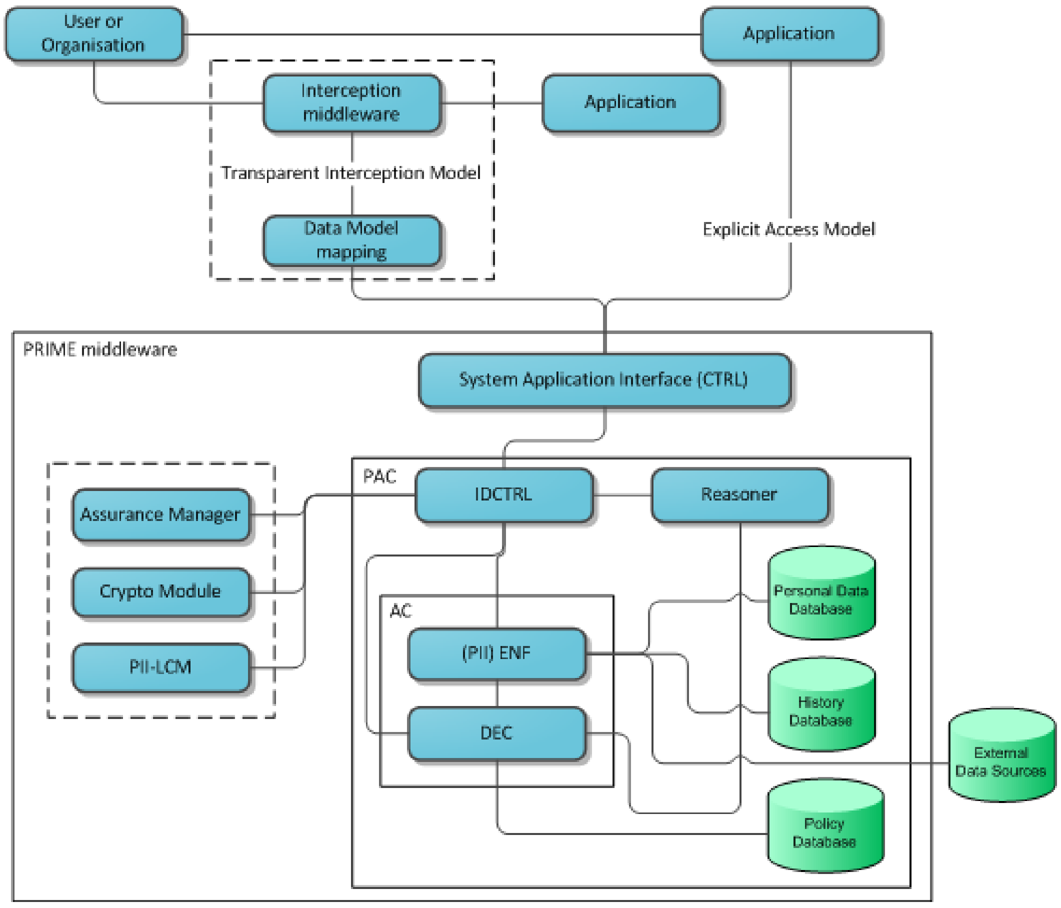


Figure 2.1: Figure showing an abstract view of the PRIME middleware. The components surrounded by discontinued lines are the optional parts of the PRIME middleware. Based on the figure in PRIME Architecture V2, 2007[1], page 54.

The Core Components

A brief description of the core components in figure2.1[1]:

- System Application Interface, is the component which provides access for applications to the PRIME middleware.
- PAC, Privacy Access Control, manages access to the protected data. Included are the following components:
  - IDCTRL, Identity Controller, and the Reasoner components handle requests for personal data by splitting them into smaller parts understood by the AC group.
  - AC, Access Control, handles access to the personal data through a more constraint API than PAC.
  - DEC, Policy Decision Point, decides whether to carry out the request or not. It can also request more information by the requester to fulfill additional requirements.
  - ENF, Policy Enforcement Point, accesses the personal data based on the decision made by the DEC.
- Personal Data Database, contains the personal data and accompanying policies.
- History Database, contains released personal data and accompanying policies to other users and organizations.
- Policy Database, contains policies and their uses.

The Optional Components

A brief description of the optional components in figure 2.1[1]:

- PII-LCM, Personal Data LifeCycle Management, handles the PRIME middleware's obligational policies on personal data, such as deletion.
- Assurance Manager, monitors the PRIME components to ensure that they have not been compromised, ensures users and organizations that the policies of personal data will be upheld and can also manage reputations of other users and organizations.
- Interception middleware and Data Model mapping, handles the communication between legacy applications and the PRIME middleware.
- Crypto Module, provides the other components with cryptographic functions.

2.1.4 Logging and Auditing

For users and organizations using PRIME it becomes necessary to provide means of logging exchanges of personal data. This is important in order to verify whether the policies on personal data has been upheld or not and if questions arise know who is responsible. The log needs to be secure and not in of itself become an invasion of privacy.

2.2 Cryptographic Primitives

Cryptographic primitives are used as building blocks when designing cryptographic systems[8]. Below follows a brief description of the primitives, with a focus on the relevant parts, used throughout this thesis. This entire section can safely be skipped by readers familiar with basic cryptographic primitives.

2.2.1 Hash, MAC and HMAC

According to the Wikipedia entry on hashes[7], a cryptographic hash function is a one-way function that takes an arbitrary-length message as input and produces a fixed-length output. As a minimum a hash must have the following properties[7]:

- **Preimage resistance**: given a hash h it should be hard to find any message m such that $h = \text{hash}(m)$.
- **Weak collision resistance**: given a message m_1 it should be hard to find another message m_2, where $m_1 \neq m_2$, such that $\text{hash}(m_1) = \text{hash}(m_2)$.
- **Strong collision resistance**: it should be hard to find two different messages $m_1$and m_2 such that $\text{hash}(m_1) = \text{hash}(m_2)$.

A hash is often compared to a digital fingerprint or a checksum[37]. One of the main applications of a hash is to verify the *integrity* of a message. An example would be the common practice in the open source community of providing the hash of an ISO file for various software distributions.

A Message Authentication Code (MAC) algorithm, like a hash function, takes an arbitrary-length message as input and produces a fixed-length output[18]. The difference is that a MAC algorithm, in addition to the arbitrary-length message, also takes as input a secret key[37]. This allows a MAC to verify both the *integrity* and *authenticity* of a message, where knowledge of the secret key serves as authentication. Another way to look at a MAC is to say that only someone who knows the secret key used to generate a MAC for a message can verify the integrity (and create a seemingly valid MAC for someone else in possession of the same secret key), compared to a hash where anyone can verify the integrity by generating the hash value[37]. A HMAC is simply a MAC that uses a hash algorithm internally[23].

2.2.2 Symmetric and Asymmetric Ciphers

A cipher is an algorithm that performs *encryption* or *decryption*[4]. The act of encrypting a message transforms it into a form that is unreadable to everyone except for those who knows the key that decrypts it[11]. For symmetric ciphers the same key, usually referred to as the private key or secret key, is used for encryption and decryption[37]. When it comes to asymmetric ciphers a keypair is used; one private and one public key. The private key is kept secret while the public key can safely be made public. A message encrypted with the public key can only be successfully decrypted using the corresponding private key and vice versa[37].

2.2.3 Digital Signatures

A digital signature is a way to validate the authenticity of a digital message[9]. A signature is formed by creating a hash of the message, encrypting the hash with the private key of the sender and appending the result at the end of the message. A recipient can then, by creating the hash of the message and comparing it the hash obtained after decrypting the attached signature using the public key of the sender, be reasonably sure that the message has not been modified since it was signed[37].

2.3 Secure Logs

The first thing to clarify is the ambiguity of the word “secure” in “secure logs”. A secure log, compared to a regular log, *protects* the *confidentiality* and *integrity* of the entries in the log[36]. The confidentiality is provided by encrypting the entries, which *prevents* an attacker from reading the contents. The integrity is provided by using hashes and MACs, which allows *detection* of any changes made to the entries. Furthermore, secure logs are concerned with the security of the log entries committed to the log *prior to* an attacker compromising the logging system; once compromised little can be done to secure future commits to the log.

The secure log described by Kelsey and Schneier[36] has been used as a foundation for our privacy-friendly secure log, as described in the next chapter, and in other related secure logging schemes by among others Holt[30], S. Sackmann et al[35] and Ma et al[33][34].

2.3.1 The Kelsey-Schneier Secure Log

Below follows the most important parts for providing confidentially and integrity in the Kelsey-Schneier secure log. Each field in the log entry layout used will be described.

An Inital Secret

A secret A_0 is used when a new log is initialized. This secret is then, as new entries are added to the log, hashed to produce an *authentication key* for each entry. For the entry with index i the authentication key is defined as $A_i = hash(A_{i-1})$. When a new authentication key is generated the old value is overwritten and irretrievably deleted. The authentication key generation takes place as soon as possible, so if the latest entry in the log has index i then the authentication key for entry $i + 1$ is stored in memory. This together with the preimage resistance property of the hash function, as described in section 2.2.1, means an attacker will be unable to gain knowledge of authentication keys used for entries committed to the log prior to the attacker compromising the logging system.

Encryption of the Data

The actual data to be logged, denoted as D, is stored in encrypted form using a symmetric cipher. The key used for encryption is derived from the authentication key and defined as $K_i = hash(W_i, A_i)$, for the entry with index i where ${W_i}^1$ is a value that is stored in plaintext as part of every log entry. The definition for the data field in a log entry is thus $E_{K_i}(D_i)$.

A Hash Chain

Each log entry contains an element in a hash chain that serves to authenticate the values of all previous log entries.[36]The hash chain, denoted with a Y, is defined as $Y_i = hash(Y_{i-1}, E_{K_i}(D_i), W_i)$ for the entry with index i.

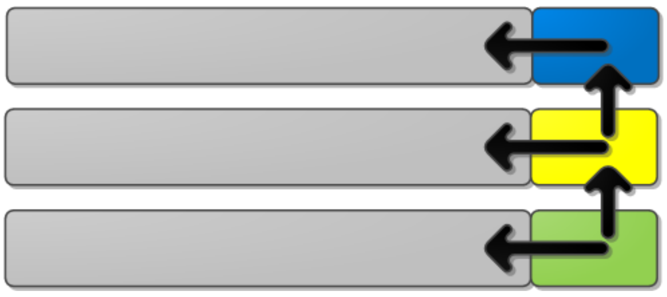


Figure 2.2: The hash chain concept used in the Kelsey-Schneier log.

The top entry in figure 2.2 represents the first entry in a log. Part of the first entry is a hash, represented by the box to the right, of all the other fields in the entry. The second entry also contains a hash, represented by the box to the right, of all the other fields in the entry *and the value of the hash in the previous entry* (top right box). In the same way the

[1]W is actually a permission mask used to determine who gains access to that particular log entry in the overall scheme described by Kelsey-Schneier[36]. It does however have no affect on the parts described in this paper other than being a known prefix used for key generation (also known as a salt).

third entry's hash field, the bottom right box, is a hash of all the other fields in the entry and the value of the hash in the previous entry (middle right box).

What the hash chain provides, given adequate protection, is *cumulative verification.* By verifying that the hash chain value is correct for one entry all entries committed to the log prior to the entry being checked is verified as well.

A MAC

The last field in the Kelsey-Schneier log entry is a MAC of the hash chain, $Z_i = MAC_{A_i}(Y_i)$. The MAC protects the integrity of the hash chain since only someone with knowledge of the authentication key for the entry can generate a valid MAC for the chain.

The MAC and hash chain are separate (instead of having the MAC form the chain) because, in the overall scheme by Kelsey-Schneier, they want to allow a partially-trusted verifier to be able to verify parts of the log without knowing any of the authentication keys used in the scheme. In addition to the extra field in each log entry a simple protocol between the verifier and a trusted party who knows the authentication keys is needed.

Overview

The authentication key is the core secret that provides all security in the Kelsey-Schneier secure log[36]. Figure 2.3 summarizes this subsection by showing the structure of an entry in the log with all the parts that have been described earlier.

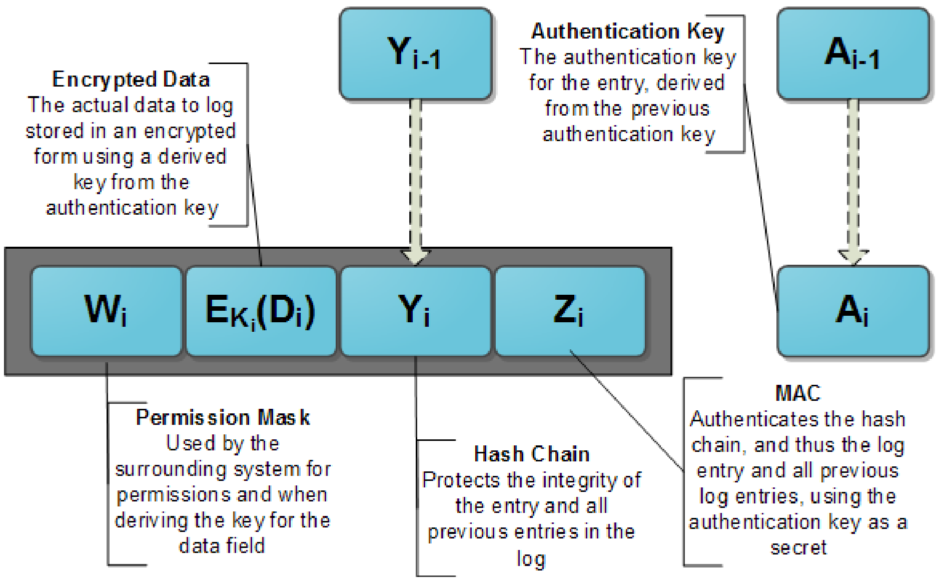

Y_{i-1}
A_{i-1}
Authentication Key
The authentication key for the entry, derived from the previous authentication key
Encrypted Data
The actual data to log stored in an encrypted form using a derived key from the authentication key
W_i
$E_{K_i}(D_i)$
Y_i
Z_i
A_i
Permission Mask
Used by the surrounding system for permissions and when deriving the key for the data field
Hash Chain
Protects the integrity of the entry and all previous entries in the log
MAC
Authenticates the hash chain, and thus the log entry and all previous log entries, using the authentication key as a secret

Figure 2.3: The Kelsey-Schneier log entry structure summarized. Based upon the log entry overview figure on page 4 in the Kelsey-Schneier paper[36].

2.4 Summary

In this chapter a brief introduction to PrimeLife and the PRIME middleware has been presented. Also basic cryptographic concepts and algorithms has been described to give an understanding on how to protect the confidentiality and integrity of personal data. Then we have seen how Kelsey and Schneier created a secure log using concepts like hash chaining. In the next chapter we will see these techniques being used in the creation of a secure log which also protects the privacy of its users.

Chapter 3

Prior Work

This thesis builds upon the work done by Peter Hjärtquist, Andreas Lavén, Tobias Pulls and their supervisor Hans Hedbom during the first half of 2009. As part of the Topics in Computer Security course at Karlstad University, during late spring 2009, an assignment was given to design and implement a prototype for a *privacy preserving secure log system* under the supervision of Hans Hedbom. The results of the design effort in the course was further developed and later presented at the IFIP International Summer School[28]. This chapter builds partly on the material presented then.

3.1 The Assignment

To get the students[1] started they received a number of references to academic work on secure logs, which are summarized in section 2.3, together with a document describing the requirements on the work to be done. Part of the requirements document was a figure similar to figure 3.1 which shows the general architecture of the log system and the different

[1]Tobias Pulls, one of the authors of this thesis, together with Peter Hjärtquist and Andreas Lavén.

components.

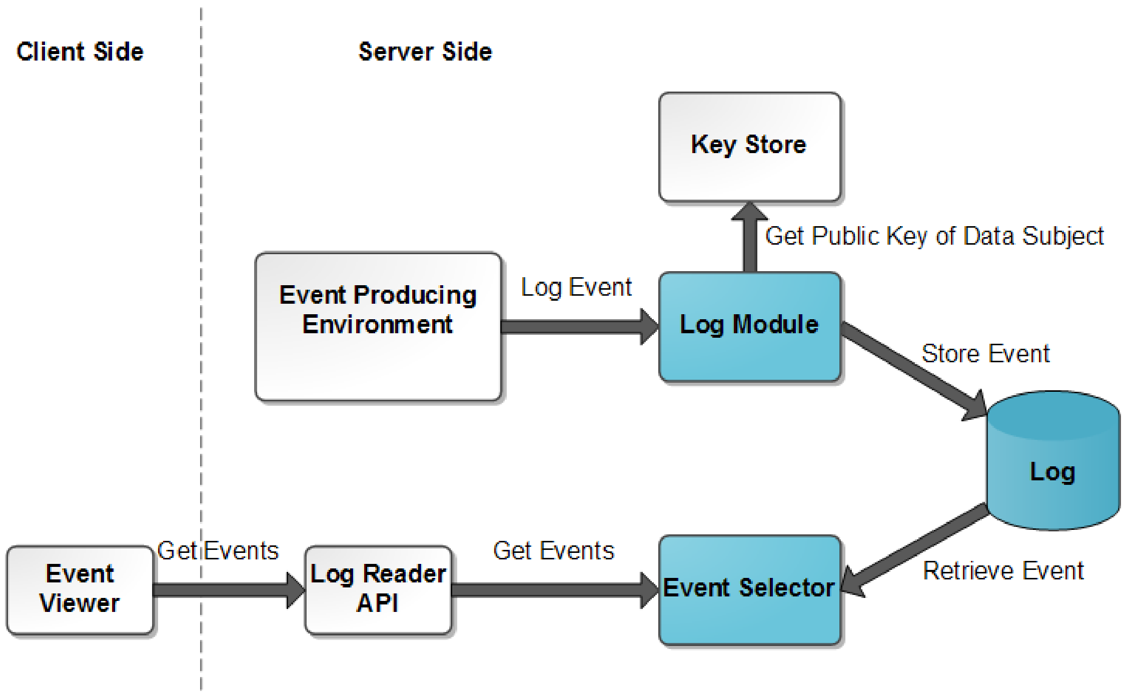


Figure 3.1: The components of the log system. Based upon the figure in the assignment handed out as part of the Topics in Computer Security course at KAU spring 2009.

A brief description of each component follows:

Event Producing Environment - The component that is under audit and produces events to be logged. Each event is defined by some data associated with a *data subject identifier*[2] .

Key Store - Contains the public key for each data subject in the system.

Log Module - Receives log events and transforms them into secure privacy preserving log entries, with the help of the public key of the corresponding data subject, and

[2]Data subject identifier means the identifier for the entity for whom the data concerns. This is a URI (Uniform Resource Identifier) in the system.

stores them in the log.

Log - The database storing the log entries.

Event Selector - Given an entry identifier the event selector retrieves the entry with the identifier from the log and returns it to the requestor.

Log Reader API - The API that provides anonymous or controlled access to the Event Selector.

Event Viewer - Downloads, decrypts and displays the log entries from the server in a user friendly fashion. Also contains functionality for validating the integrity of the log entries and to decide if any policy violation has occurred.

The students were tasked with developing the shaded components in figure 3.1 (Log Module, Log and Event Selector); for all other components simple placeholders were to be developed as needed.

A number of requirements were placed on both the design and prototype to be developed, ranging from the programming language to use to specific properties of the log design. These requirements will be described as the log design and the prototype is explained in the following sections.

3.2 The Design

What follows is an overview of our privacy-friendly secure log design with a focus on comparing it to the parts of the Kelsey-Schneier log described earlier in section 2.3.1. Details will be described as needed but interested readers are encouraged to read our full paper [28] for more information.

As was mentioned in the previous section, when describing the event producing environment that is under audit, our log stores events relating to data subjects. We want to allow users (known under one or more data subject identifiers by the event producing environment) who uses a client software (part of which is the event viewer described earlier) to read the log entries relating to the user.

3.2.1 Requirements

The following high-level requirements were put on our design[28]:

1. It should not be possible for anybody except the data subject to decrypt log entries once they are committed to the log.

2. It should not be possible to alter nor remove entries made prior to an attacker taking control of the logging system without detection.

3. There should be a high degree of *unlinkability* between data subjects and the log entries in the log.[3]In other words, it should be computationally hard for an attacker to determine to which data subject each entry in the log belongs to.

3.2.2 Secrets and Authentication Keys

Like Kelsey-Schneier we use secrets, that we generate authentication keys from, to ultimately provide all the security in the scheme. Since we want to allow each data subject, and the server, to validate the integrity of their respective log entries (in the case of the server the entire log) they all need secrets of their own.

[3]Initially the requirement was for full unlinkability between data subjects and log entries but it was changed early due to how prior academic research approached secure logs.

The secrets for the server are denoted SAS_0 and $ServerID_0$. SAS_0 is the *Server Authentication Secret* used to authenticate all entries in the log for the server. The role of the $ServerID_0$ will be explained in the following section. The secrets for each data subject are denoted DSS_0 and $EntryID_0$. DSS_0 is the *Data Subject Secret* used to authenticate all entries in the log for the data subject. The role of the $EntryID_0$ will be explained in the following section.

The SAS and DSS secrets are used like the secret in the Kelsey-Schneier log; for example the authentication key for the server for the first entry in the log is defined as $SAS_1 = hash(SAS_0)$.

3.2.3 Order of Entries In the Log and ID Generation

In the Kelsey-Schneier log the order of the entries is implicitly assumed to be chronological. This is not acceptable in our log because of our requirement for a high degree of unlinkability between log entries and data subjects. If an attacker was able to order all entries in the log in chronological order something as simple as an access log (like the Apache default access log) for a service using the PRIME Core (and thus ultimately the log) together with some statistical analysis would probably aid an attacker greatly in linking entries to data subjects.

Our log is simply a set[4] of log entries. There are, primarily, two ways to order some or all entries in the log in chronological order; through the server ID and entry ID generation. The ServerID field, for an entry with index i, is defined as $ServerID_i = hash(ServerID_{i-1}, SAS_i)$. For the first entry in the log, since there is no previous server ID, the $ServerID_0$ secret is used. By generating the server IDs the server, or anyone

[4]Perhaps a more accurate description is a multiset, where there can be multiple instances of the same value[19].

with knowledge of the server's secrets, can order all entries in chronological order. In the same way, for an entry with index j, the EntryID field is defined as $EntryID_j = hash(EntryID_{j-1}, DSS_j)$. For the first entry in the log *for the data subject*, the $EntryID_0$ secret is used. By generating the entry IDs the data subject, or anyone with knowledge of the data subject's secrets, can order all entries *belonging to that data subject* in chronological order. The approach we take to generating server IDs and entry IDs are similar to how Kelsey-Schneier generate the key used for encrypting the data field.

3.2.4 Data Field

The data field, which holds the actual data to log, is encrypted with an asymmetric encryption algorithm using the public key of the data subject. In addition to the data to log the entry also contains a signature of the data signed with the private key of the server. This allows the data subject to, after having retrieved and decrypted the data field, prove that the data was committed to the log by the server. The data is appended with a nonce[5] before encryption to make attempts to link the log entry to a specific data subject harder (with a common log entry an attacker could encrypt it using the public key of the data subject and try to match it to an entry stored in the log). The definition for the data field is thus $Data_i = ENC_{PU_{DS}}(SIGN_{PR_S}(data), data, nonce)$.

3.2.5 Chains

To allow the server and data subjects to validate the integrity of the log we use two chains; the ServerChain and the DataSubjectChain. The ServerChain allows the server to validate the integrity of all entries in the log while the DataSubjectChain allows each data subject

[5]A nonce is a random *n*umber used *once*.

to validate the integrity of their respective entries in the log. Our chains are a combination of the hash and MAC fields used by Kelsey-Schneier. For the i:th entry *in the entire log* the ServerChain is defined as $ServerChain_i = HMAC_{SAS_i}(ServerChain_{i-1}, DataSubjectChain_i, Data_i, EntryID_i, ServerID_i)$. For the j:th entry *in the log for the data subject* the DataSubjectChain is defined as $DataSubjectChain_j = HMAC_{DSS_j}(DataSubjectChain_{j-1}, EntryID_j, Data_j)$. Note that if we were to use a hash instead of a MAC for our chains anyone with access to the log could order all entries in the log in chronological order (through the ServerChain) and link entries belonging to data subjects (through the DataSubjectChain).

3.2.6 Structure and State

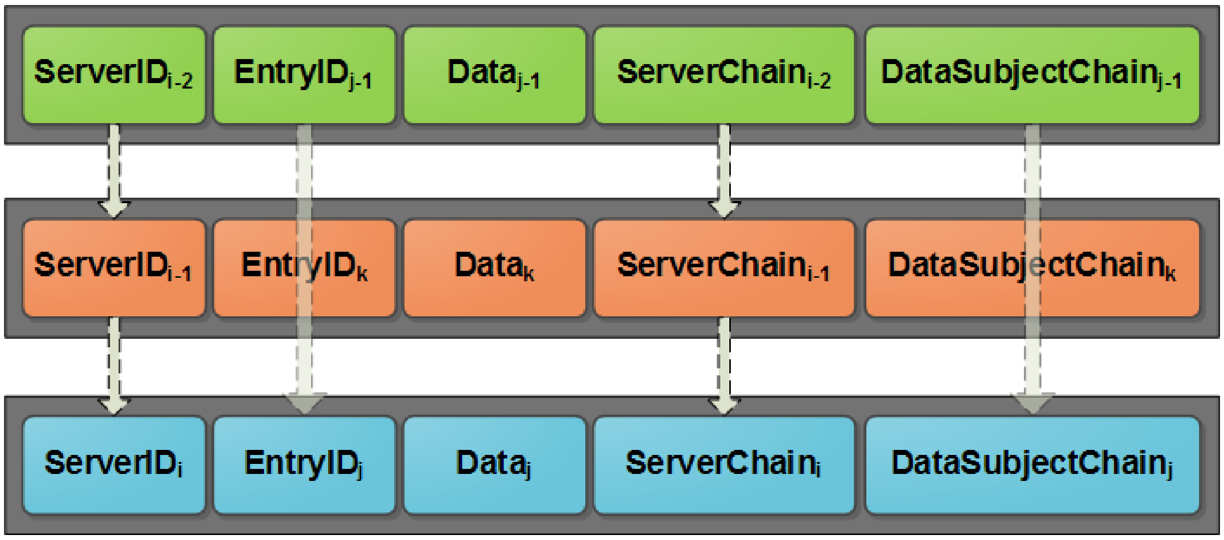


Figure 3.2: The structure of our logging scheme. The bottom entry is the new entry in the log and the top entry the previous entry in the log belonging to the same data subject (Bob). The middle entry is the previous entry committed to the log, but it belongs to another data subject (Alice).

We have now gone over enough to present the complete structure and state kept by the logging system in our scheme. Figure 3.2 shows the structure of our logging scheme and three different entries. The top entry belongs to the data subject Bob while the middle entry belongs to the data subject Alice. In the figure we have an additional entry, the bottom one, that is a new entry being added to the log for Bob.

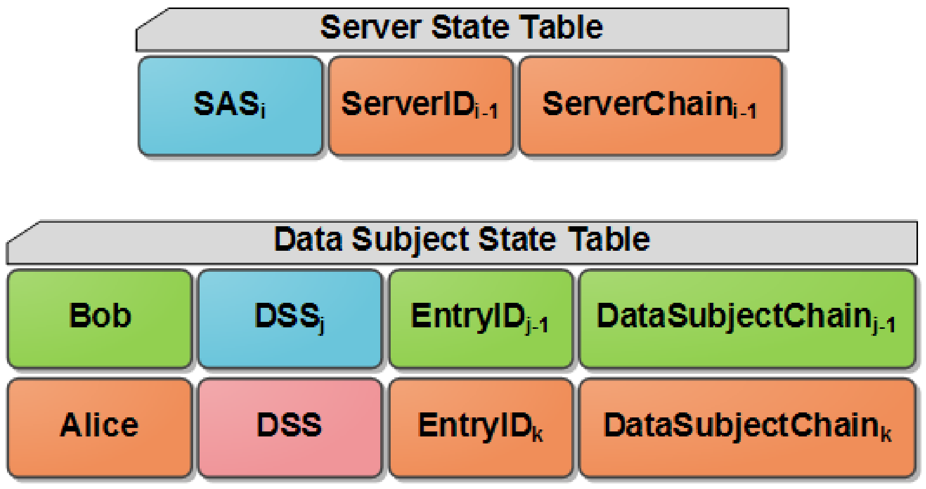


Figure 3.3: The structure of our log state tables. This example shows the state for two data subjects, Alice and Bob, together with the server state table. This is the state the log is in before another entry gets added for the data subject Bob, shown in figure 3.2.

Figure 3.3 shows the two state tables used by the server in the state they are before the new bottom entry is to be added to the log. The index i in figure 3.2 and 3.3 refers to the current entry in the log for the server. From the server's point of view the previous entry in the log is thus the middle entry which belongs to Alice. The index i, in figure 3.2 and 3.3, refers to the current entry in the log for the data subject Bob. From Bob's point of view the previous entry in the log is the top entry. The index k belongs to the data subject Alice. All of this information needs to be stored, and continuously updated as new entries are added, to the server's state tables together with the authentication key generation described earlier.

An attacker, after having compromised the logging system, can use the state table to link the latest entry in the log to each data subject in the system. Furthermore, the attacker can determine which of the entries were the last entry made in the server. This is a compromise we have done on the unlinkability requirement for the sake of getting cumulative verification of our log entries. The attacker does not learn how many entries in the log belong to a specific data subject and a simple assumption by any attacker would be that if a data subject exists in the system there are at least one entry in the log concerning that data subject.

3.2.7 The API

Anonymous access is provided to the log through the following API methods:

- **GetLogEntry(EntryID)** - returns the log entry (could theoretically be several in the case of a hash collision) with the provided entry ID. Since only a data subject with knowledge of the private key that decrypts the data field in an entry can read the contents of an entry this method can safely be made public.

- **GetLatestEntryID(DataSubjectIdentifier)** - returns the latest entry ID for the data subject together with a nonce, encrypted with the public key of the data subject. This method can be used by the data subject (but not fully relied upon) when generating all the entry IDs belonging to the data subject identifier. When the latest entry ID is requested for a data subject identifier that does not exist in the system a seemingly valid response should be returned to make it harder for an outside attacker to deduce valid data subject identifiers.

3.2.8 Validation

All communication between a client and server is assumed to be over an encrypted anonymous channel (for example TLS over the Tor network). This subsection greatly resembles the corresponding section in our paper[28].

Fetch all entries for a data subject identifier from the server

The client is assumed to have knowledge of the initial data subject identifier secrets DSS_0 and $EntryID_0$ for the data subject identifier.

1. Request getLatestEntryID() from the servers API for the data subject.
2. Generate and make a list of all entry IDs from $EntryID_1$ to the latest ID returned from the server in step 1.
3. Request all the log entries based on entry ID in random order from the server.

Integrity validation by the client for a data subject

1. Fetch all the log entries from the server by following the steps described earlier.
2. Generate the DataSubjectChain and compare it to the stored values in the entries. If it at any point does not match the validation fails.
3. Verify that the signature in each entry is valid.
4. Generate at least one more entry ID and request it from the server. If any entry is returned the validation fails.
5. Compare the recently downloaded entries in step 1 with the old entries (if any) stored in the client. If any entry differs or were not found on the server the validation fails.

Integrity validation by the server (or trusted third-party)

The server (or a trusted third-party) can validate the integrity of the entire log by knowing the initial server secrets (SAS_0 and $ServerID_0$).

1. Starting from $ServerID_0$:
   (a) Generate the next server ID and match it to an entry in the log. Each time you match an entry note it down on a list.
   (b) Generate the ServerChain and compare it to the stored value in the entry. If it does not match the validation fails.
   (c) Repeat until a generated server ID is not found in the log.
2. Compare the list from step 1 with the log. If there is any entry in the log that is not on the list the validation fails.
3. Examine the entry for the server in the server state table and verify that the correct server authentication key and previous entry are set.

3.2.9 Summary

The Kelsey-Schneier secure log was used as a foundation for our secure log. Our main contribution is adding a high degree of unlinkability between log entries and data subjects, making the log privacy friendly. The server and all data subjects can independently of each other validate the integrity of their respective log entries. Furthermore, each entry in the log can be downloaded one at a time through an API that does not need to perform any form of authentication.

3.3 The Prototype

The main goal for our prototype was to show that our design worked in practice and have it be reasonably easy to salvage parts later.

3.3.1 Requirements

The following high-level requirements were put on our prototype:

- Java 1.5[14] with no external libraries.
- HSQLDB[12] as the backend database.
- Good object-oriented programming and Java programming principles should be followed. The solution should be extendable.

3.3.2 Prototype Design

In appendix A the class diagram for the prototype can be found. It might be useful to take another peak at figure 3.1 before reading the description of the different classes that follows.

The following classes make up the interface for the log:

- **LogModule** - represents the logging module for the event producing environment.
- **EventSelector** - represents the logging module for the client. We had the client use the EventSelector directly instead of implementing an API to wrap around it as shown in figure 3.1.

Classes used by the LogModule and EventSelector to accomplish their respective tasks:

- **Crypto** - handles all cryptographic operations.
- **KeyStore** - stores the public key of the data subjects. Since this was only a prototype we hardcoded the keys for two data subjects to be read from a file.
- **LogStore and LogStoreHSQL** - LogStore is an interface that declares the methods used for the actual storage and retrieval of log entries. LogStoreHSQL is an implementation of the LogStore interface that uses HSQLDB for storing log entries.
- **LogState and LogStateHSQL** -LogState is an interface that declares the methods used for managing all the secrets and states in the scheme. LogStateHSQL is an implementation of the LogState interface that uses HSQLDB for storing the state.
- **LogFactory** - determines which implementation of the LogState and LogStore interfaces are used by the rest of the system and ensures that only one instance of each are created.

An entry in the log is represented by a number of classes that forms a hierarchy:

- **LogEntry** - represents a log entry. Contains the ServerID field, the ServerChain field and a ClientLogEntry object.
- **ClientLogEntry** - represents the client[6] part of an entry. Contains the EntryID field, the ClientChain field and the data field in encrypted form. The data field holds a serialized CryptoWrapper object.
- **CryptoWrapper** - contains a nonce, a serialized LogDataWrapper encrypted with AES (a symmetric encryption algorithm) together with the randomly generated key,

[6]What we mean here is the data subject part of the entry. This terminology was changed while writing our paper. In a similar fashion the DataSubjectChain used to be called the ClientChain.

used to encrypt the LogDataWrapper, encrypted with the public key of the data subject. This is not what was described in the log design, since we are only using an asymmetric encryption algorithm to encrypt a key instead of the entire thing. The reason behind this change is that RSA (the most widely adopted asymmetric encryption algorithm) does not support encrypting large amounts of data and is significantly slower than AES. Generating a random symmetric key used to encrypt the data and then encrypting the symmetric key with RSA is a common approach when dealing with large amounts of data to protect with an asymmetric key. For example, it is the approach taken by EFS in the NTFS file system[10].

- **LogDataWrapper** - contains a LogData object and a signature of the serialized LogData object using the private key of the server.

- **LogData** - contains the actual data to log as a byte array.

To be able to test the log module we developed a dummy client and event producing environment:

- **Client** - the client used by a user for retrieving the entries in the log relating to the user's data subject(s). Uses the EventSelector to retrieve entries from the log. Only basic functionality is provided and leaves a lot to desire when it comes to performing validation of the entries and how it behaves when requesting entries through the EventSelector.

- **Main** - the event producing environment. Takes simple command line arguments to add entries to the log for two hardcoded data subjects.

Exceptions and a debug class:

- **InvalidDataSubjectException** - an exception thrown by various parts of the system to indicate that the supplied data subject was invalid.
- **NoEntryFoundException** - an exception thrown by various parts of the system to indicate that no entry with the supplied entry ID exists.
- **Debug** - used to print debug messages by the system and provides an easy way to turn them off.

3.4 Summary

The design of our privacy-friendly secure log has been an ongoing process that continued after the Topics in Computer Security course. While the prototype serves its purpose as a proof of concept for the design it is far from ready for serious use in any system. Updating the prototype to follow the design in the published paper[28] and to integrate it into the PRIME Core is the main goal of this thesis.

Chapter 4

Implementation

We divided our work into several milestones each with a clear goal in mind. The biggest factor that determined how this split was made was the size of the PRIME Core codebase. At the start of this thesis the PRIME Core alone consisted of over 300 classes and 200 JUnit tests, with parts under active development. We wanted to ensure that our log module was fully functional, and reflected the changes made to the design after the initial prototype was constructed, before we started integrating it with the rest of the core.

4.1 Milestone 1 - Update Standalone

Our first milestone was to update our standalone logging module with all the changes made to the design and to incorporate a number of changes we identified during our research at the start of the thesis.

4.1.1 Naming and Coding Conventions

According to the Wikipedia entry on Naming Conventions[20], a naming convention consists of rules used to name identifiers such as variables and functions. This applies to both source code and documentation. For example, a rule could be to only use a lower case letter as the first character in any method name.

Also, according to Wikipedia's entry on Coding Conventions[6], a coding convention is a set of guidelines used when writing code. They can cover a lot of different areas, such as how files are named and can also include a naming convention. The coding conventions are language specific however there may exist different coding conventions for each language. As stated by Sun Microsystems[5], coding conventions are important because they improve readability which makes code easier to maintain.

In the development of the privacy-friendly secure log it is important to note that the maintenance of our piece of software will be performed by the other participants of the PRIME project, and thus making it understandable to others is a strong requirement. Since the other parts of the PRIME Core uses the coding conventions created by Sun Microsystems, and to follow them was a requirement (in section 3.2.1), we needed to update our previous code and documentation to follow the same guidelines as well as follow them in our future development of the privacy-friendly secure log.

Although we tried to follow all the guidelines throughout the implementation, one of the guidelines were ignored in some parts of the code. It was the placement of declarations, which should only be present in the beginning of a block of code. In some long methods, most notably the JUnit tests for the log, the declarations were instead placed before they were to be used. This change was made because it required less lines of code and made the tests easier to follow.

4.1.2 Centralized Configuration

The PRIME Core provides a class named Config to access the current configuration of the core. The configuration is represented by simple key-value string pairs. The Config class uses a three tier approach to determine the configuration of the core. The default values are hardcoded static values set directly in the class. The second tier of values are read from a config file found through a hardcoded path. Any settings in the config file will overwrite the default values. In a similar fashion, any command line arguments passed to the PRIME Core at launch will overwrite the default values and those set in any config file.

The task was to identify all the configuration values used throughout our log, update the code to use a Config class similar to that used in the PRIME Core (for easy integration in the next milestone) and to set sane defaults. The values for our log allow the configuration of:

- Which implementation of the LogState, LogStore and KeyHandler interfaces to use throughout the log.

- Specific settings for the HSQLDB implementations of the LogState, LogStore and KeyHandler interfaces.

- Cryptographic algorithms and related settings for all the cryptographic operations in the LogModule.

A list of all of configuration values for our log and their default values can be found in Appendix B.

4.1.3 Creating the KeyHandler

Since the initial prototype contained nothing more than hardcoded keys inside a KeyStore class we had to create a real KeyStore. To avoid a confusing naming conflict with the built in Java KeyStore we decided to rename our KeyStore class to KeyHandler.

The first thing we did was to implement our KeyHandler using a Java KeyStore internally to store all keys, hoping this would lead to an easy integration with the internal KeyStore used in the PRIME Core. However, after some digging around at a later date we came to the conclusion that using any of the keys already generated inside of the PRIME Core would potentially allow an attacker to link several data subject identifiers together by comparing keys obtained through other functionality[1] in the PRIME Core. With this in mind we decided to give our KeyHandler the same treatment as our LogStore and LogState classes; we turned it into an interface, let LogFactory manage it for the rest of the system and wrote an implementation using HSQLDB for storing the keys.

The fact that the KeyHandler is now an interface allows for our LogModule to be easily updated, should any central key management in the future be created for the PRIME Core. For now we are generating all of our own cryptographic keys, something which will be discussed further in the next milestone.

4.1.4 The Event Selector

In our standalone prototype, the Event Selector has two main methods: *getLogEntry* and *getLatestEntryID*. The method *getLogEntry* is used to retrieve a log entry with a given identifier, while *getLatestEntryID* returns the latest identifier for a given data subject.

Previously the *getLatestEntryID* method returned the latest entry ID of a data subject

[1]For example, the webservices that provides the API runs SSL. Connecting to this service returns the public key of the PRIME Core.

in clear text, which made it possible for an attacker to know the latest log entry of a data subject within the log. The method also threw an exception if a given data subject did not exist in the log. This would give an attacker opportunity to guess which data subjects existed in the system. The Event Selector needed to be updated to fix these problems.

Public key cryptography is now used in the *getLatestEntryID* method as well as giving fake responses to unknown data subjects. When the method is called, the entry ID and a nonce is encrypted together using the data subject's public key. It would thereby be unfeasible for an attacker to decipher the entry ID of a data subject's latest log entry.

In order to prevent a potential attacker from guessing the data subjects in the log, the *getLatestEntryID* method was also modified to return fake responses to invalid data subjects. These responses looks valid however they are encrypted using a special public key for which the private key has been discarded. For an attacker to tell the difference between a real response of a valid data subject and a fake response from an invalid data subject would be difficult due to the nature of public key cryptography.

The *getLogEntry* method was changed to return an entire log entry instead of only the data subject's part of the log entry. This gives a Client the ability to validate that the server parts of the log entry has not been changed since the previous download of the same log entry.

4.1.5 Updating Crypto

The direct use of cryptography is well compartmentalized to a few components in the PRIME Core, each with their own, at the time of writing, hardcoded algorithms and related configuration not using the PRIME Core's Config class. During the summer the Bouncy Castle library was included into the project, meaning that we now had access to more algorithms and easy to use cryptographic methods. We centralized the generation of random numbers using an instance of the digest random number generator from Bouncy Castle, enabling us to easily add more seeding material. Furthermore, throughout the development, we added a number of helper methods to Crypto allowing us to focus on the big picture in the rest of the code.

4.1.6 Log Integrity Validation

The integrity validation methods, both for the server and client, in our initial prototype lacked several steps that were outlined in the design as described in section 3.2.8. Changes to the validation process was also made after the development of the initial prototype.

The server validation was enhanced in two ways. First the check of the server's state was implemented, which validates that the expected SAS, ServerID and ServerChain values are set in LogState. The second change we did was to move the entire server validation out of the LogModule and into a new standalone class. This makes it easier for trusted third-parties, with knowledge of the server's secrets, to validate the log.

The client validation changed the most after our initial prototype was created. We realized that the behavior of the client when performing the validation could severely affect the linkability of entries to data subjects if an attacker, after having taken control of the server, watched the incoming requests to the API. Our first countermeasure was to request all the entries from the server in a random order, instead of in a sequence as was done before. While this change still allows an attacker to link the entries to a data subject (imagine a burst of requests for entries together with a call to the *getLatestEntryID* method for a subject), it makes it harder for an attacker to deduce more information since she does not know the actual order of the entries in the log.

The first countermeasure is far from adequate if the default behavior from the client is to always do a full validation where all entries in the log for a data subject are downloaded from the server. To allow for different forms of validation to be developed, for example where only a small subset of entries are re-downloaded together with the last five entries in the log for the data subject, we decided to split up the validation method into several parts.

- **LogEntry[] Client.getAllEntriesSecurely()** - Downloads all entries from the server in random order and returns them in an array.
- **boolean Client.validateDataSubjectChain(LogEntry current, LogEntry previous, byte[] key)** - Validates the DataSubjectChain in the current entry given the previous entry and the authentication key for the entry to be verified.
- **boolean Client.validateSignature(LogEntry entry)** - Validates that the signature in the entry is made by the server's private key. The server's public key is a mandatory argument when constructing the Client and set in a private variable.
- **boolean Client.validateGetLatestEntryID()** - Validates that the entry ID returned from the *getLatestEntryID* method on the server really is the latest entry ID in the log for the data subject.
- **boolean Client.validateEntryList(LogEntry[] original, LogEntry[] copy)** - Validate that a copy of a LogEntry array is equal to an original LogEntry array for each entry in the original array. The copy array can contain more entries than the original array, what matters is that the copy array is equal to the original array for each item in the original array.

Each part performs a given task and can be used for future versions of the log validation.

4.1.7 JUnit Tests

JUnit is a unit testing framework, that allows for test-driven development, for the Java programming language.[17] Our initial prototype had only a few small JUnit tests for the Crypto class. The testing of the LogModule was done through two methods representing a server and a client with two data subjects. Since the PRIME Core heavily relies upon automated JUnit tests, all of which are run each time new code is committed to the development SVN, we needed to replace our testing code with JUnit tests.

Since a big part of the classes used in our log are little more than a number of set and get methods, we decided to focus our tests on the overall functionality of the log together with specific tests targeting the Crypto class where the big transformations are done. Our two test classes are called LogModuleTest and CryptoTest.

LogModuleTest tests the overall functionality of the log. First it initializes a new LogModule and adds several entries to the log. In between entries being added both the server and client validates the integrity of the log. Towards the end of the main test several modifications are made to the log and we ensure that the validation processes detects the changes. Furthermore the public methods of the LogModule class is checked for all the expected exceptions, such as when an attempt is made to initialize the LogModule when it has already been initialized. CryptoTest, on the other hand, operates exclusively on the methods in the Crypto class. Each method that performs any of the underlying cryptographic primitives is tested.

During the rest of our work we continuously updated our tests, or implemented new, as was needed.

4.1.8 HSQLDB Mode

The initial prototype used the HSQLDB database in the two implementations of the LogState and LogStore interfaces. During this thesis we also made an implementation of our KeyHandler interface using HSQLDB, as discussed previously.

HSQLDB can be run in several modes, ranging from being completely isolated in memory of the process using the database to running as a server on its own much like MySQL. Initially we used a full HSQLDB server for development, something which the HSQLDB documentation encourages[13], because it is possible to connect to HSQLDB from outside the JVM using the supplied management tools in that mode. We now felt more confident that the development phase of the classes using HSQLDB was coming to an end, so we swapped the mode to in-process mode using cached tables (parts of the tables are cached in memory) with frequent writes to disk. In section 4.3, describing our HSQLDB milestone, further information about the different table types in HSQLDB will be discussed.

4.2 Milestone 2 - Implement Into the PRIME Core

Once comfortable with the standalone logging module the next step was to integrate it into the PRIME Core and implement the missing functionality. At the start of the thesis we did some basic research into the two major tasks in this milestone; creating the API and enabling sharing of secrets. However, it turned out that we had to commit significantly more time throughout this milestone to fully understand how the different parts of the PRIME Core actually worked.

4.2.1 Integrate the Configuration

The centralized configuration of our privacy-friendly secure log was created during the previous milestone (see section 4.1.2 Centralized Configuration). It now contained all the configuration values and defaults.

Because of the similarity of our centralized configuration to the Config class of the PRIME Core this task became easy. The first step involved moving all the configuration values and defaults to the Config class, then updating the references from the centralized configuration to the Config class.

The last step was to assign a different prefix to our configuration values. To enable other developers to find out where the values where being used easily, the prefix was assigned the same name as the package of our standalone prototype, namely ”kau.prime.servertransparency.logging”.

4.2.2 Creating LogSecrets

When dealing with the log, for data subjects and the server itself, a set of values are used, shared or saved as part of the scheme described in chapter 3. Instead of dealing with these values individually, we created the LogSecrets class to bunch them all together. The LogSecrets class is made up of the following fields:

- **Secret**, the core secret for the LogSecrets object. This would be DSS_0 or SAS_0.
- **Seed**, the seed used for ID generation. This would be $EntryID_0$ or $ServerID_0$.
- **PublicKey**, the public key of the data subject or server.
- **PrivateKey**, the private key of the data subject or server.

LogSecrets objects can be marshalled into XML. All the public methods in the LogModule, Client and LogValidator classes, where applicable, now takes a LogSecrets object expecting some or all fields to be filled. For example, the LogModule's initialize method takes a LogSecrets object representing the server's secrets and keypair. In the same way the method for initializing a data subject takes a LogSecrets object representing the data subject's secrets and public key.

4.2.3 Sharing Secrets and Keys

As part of integrating our log into the PRIME Core we had to enable two different cores, one acting as a client and one as a server, to share secrets and for each party to obtain the needed keys. Furthermore, we had to figure out where exactly to store all of these values.

Server Secrets and Keys

The server secrets and keys needs to be given to the LogModule to initialize it before anything can be added to the log. Therefore, we want to initialize the LogModule as soon as the PRIME Core starts and all necessary information we need is available.

The main class in the core is the PrimeCore class, which among other things starts all the webservices and, depending on settings, ensures that a valid password is set in the Config class for the key "keystore.password". An instance of the Java KeyStore is used by the PRIME Core to store the certificate used by the webservices for TLS. As part of the PrimeCore constructor, our LogModule is initialized with a LogSecrets object that is afterwards saved to an encrypted file and then discarded. The encrypted file with the LogSecrets is encrypted with password-based encryption, using the value of the "keystore.password" key in the Config class as the password.

The LogSecrets object used to initialize the LogModule first contains random values for the secret and the seed, together with a newly generated keypair, generated by our Crypto class. Before the LogSecrets object is sent to the LogModule to initialize it, the Config class is consulted if there are any set values for the secrets or keys. If any values are set there they will overwrite the randomly generated values in the LogSecrets object. This allows for the LogModule to be initialized with custom values, such as a reputable keypair.

Data Subject Secrets and Keys

A data subject's secret, seed and public key needs to be given to the LogModule to initialize the subject in the log. How, and where in the PRIME Core, these values are shared from a data subject to the server's LogModule is described in the following subsection. The same values, together with the corresponding private key, is used by the Client class to facilitate the client functionality. This means that we need to store the values somewhere for later use.

The PRIME Core has a session database called SessionDB, where Session objects are stored. A Session object describes the personally identifiable information disclosed to a party together with relevant information such as time, purpose and contact information for the party. Perhaps most notably a data subject identifier is associated with a Session, which is the same as the identifier of the data subject *on the server PRIME Core*. This lead us to the decision that the natural place to store the data subject's LogSecrets is as part of a Session inside of the SessionDB. All the information an instance of the Client class needs to function can now be obtained from the Session describing the information disclosure.

Sharing Data Subject Secrets

The LogSecrets for a subject is shared with a server PRIME Core, and sent to its LogModule to initialize the subject, as part of the exchange that creates a new subject in the server core. Figure 4.1 shows a scenario where a caller application, wishing to access a remote resource, causes the creation of a new subject (or ”handle”, ”session handle” or ”session ID” outside of the PRIME Core) on the server PRIME Core.

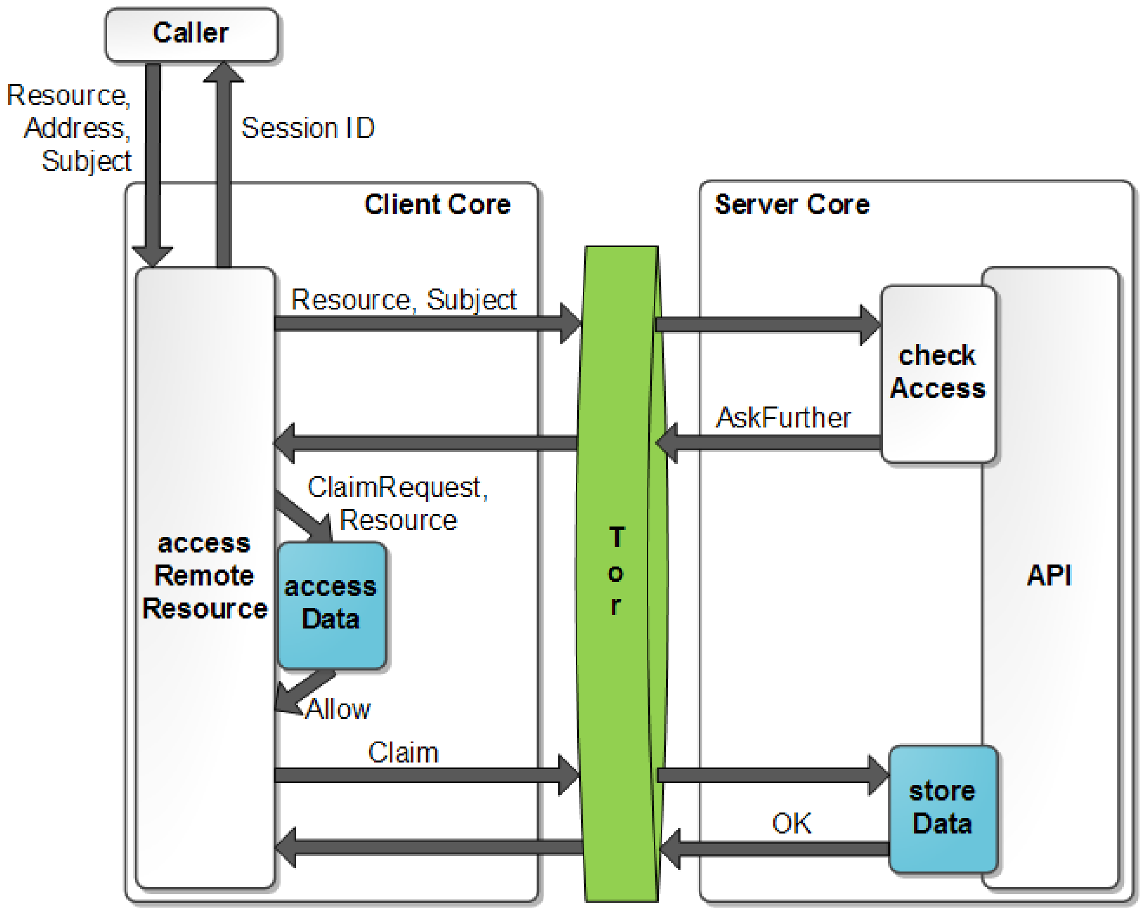


Figure 4.1: How a new subject is created in the server PRIME Core. As part of this exchange we share the data subject’s LogSecrets with the LogModule on the server. Based on the PRIME communication flow[21] and the Prime Architecture[1].

A call to the API method *accessRemoteResource* takes as arguments the resource which to access, the address to the server core where the resource is located and the subject under which to access the resource. An empty subject means that no information is stored on the server core for it to base its decision upon.

The first thing done by *accessRemoteResource* is to ask the server core for access to the resource. The possible responses are “Allow”, “Deny” and “AskFurther”. AskFurther, the only non-obvious response, asks for further information to be provided before access can be allowed. Inside an AskFurther object is a ClaimRequest object. A ClaimRequest contains a set of options, where each option represents a set of requests for statements about Personally Identifiable Information (PII) possibly together with criterias for evidence tokens[2]. Providing the server with a Claim object satisfying any of the options in the ClaimRequest will grant access to the requested resource. A Claim consists of a subject and an array of Groups, which in turn contains an array of PII together with an evidence token for the PII. Each Group can furthermore contain a data handling policy specifying how the PII should be handled.

The next step is to call *accessData* on the client to determine which, if any, of the options in the ClaimRequest can be disclosed to the server. The *accessData* method, depending on configuration, will either ask the user which option to disclose to the server or let the access control module make a decision based on current policy. If *accessData* allows the disclosure of the data requested in an option, a number of Claims[3] are returned to the *accessRemoteResource* method. The subject set in each Claim is randomly generated when created if no subject ID is given to the Claim.

[2]An example of an evidence token would be the proof of the PII stating that someone is over 18 years old.

[3]Currently only one Claim returned by *accessData*, in the case of allow, is supported. Future versions might support multiple Claim objects.

The last step is to store the Claim on the server by calling *storeData*. Once completed *accessRemoteResource* will return the subject ID of the Claim to the caller where it will be used as a session ID allowing access to the resource for which access was requested earlier.

We modified the Claim class together with the *accessData* and *storeData* methods to enable the sharing of secrets and keys for our log. The Claim class now has an extra field, named "logSecrets", which if set represents that transparency logging should be enabled on the receiving end for the disclosed data inside the Claim. Inside *accessData*, when disclosure of data is allowed, the Claim returned by *accessData* has a newly generated LogSecrets object set. Prior to the LogSecrets object being set in the Claim it is first saved to the SessionDB, together with the created Session, and its private key field is then nulled since we do not need nor wish to disclose it to anyone else. Once the Claim is sent to the server, through a call to its *storeData* method, the LogSecrets in the Claim is used to initialize the LogModule for the subject and then removed from the Claim to ensure that it is not saved anywhere on the server.

Obtaining the Public Key of the Server

The PRIME Core maintains a database of contact information, called ContactDB. The ContactDB stores Contact objects, which contains a key-value String map where a number of default contact properties are enumerated in the class. Furthermore, there exists an API method named *getContact* which returns the Contact object associated with the PRIME Core. To allow easy access to the public key of the server we simply added a key to the default contact properties containing a marshalled public key, in encoded form, wrapped in a PublicKeyWrapper. As was mentioned earlier in this section, as part of the Session stored in the SessionDB, contact information for the recipient of the information disclosure is saved. This contact information is in the form of the address to the recipient allowing a

Client to easily be able to obtain the Contact of the server, containing the server's public key.

4.2.4 Creating the API

Previously the Client class made direct calls to the Event Selector when requesting the latest entry ID or a log entry. However, in a real-world scenario the privacy-friendly secure log is located in another instance of the PRIME Core (most likely running on a server or computer other than the user's) and thus the user would be unable to retrieve the log entries in such a way. To enable the user to retrieve the correct log entries from the server the Log Reader API needs to be created (see figure 3.1 for an overview of the component's placement in the privacy-friendly secure log).

This task is comprised of the following:

- Explore how the PRIME Core client and server communicates.
- Understand how the Log Reader API could be created using the knowledge from 1.
- Implement the Log Reader API according to 2.
- Change the client to use the Log Reader API.

The PRIME Cores communicate using Web services and XML. The World Wide Web Consortium (W3C) defines a Web service as the following[22]:

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Basically a Web service allows for resources to be exchanged between client and server using XML-messages sent over an HTTP-connection. To achieve this effect in the PRIME Core the server part runs a web server (an open-source project named Jetty[16]) which relays the contents to the PRIME Core. All the services provided by the PRIME Core is used as if they were folders on that server. For example on a server running locally on port 6907, the *getLogEntry* method of the Common Web Service can be access through the web address: https://localhost:6907/common/getLogEntry. Note that the communication between PRIME Cores uses Transport Layer Security (TLS) or Secure Sockets Layer (SSL) by default.

The messages exchanged between the PRIME Core client and the PRIME Core server are of XML format that are created (marshalled) and interpreted (unmarshalled) with the use of Java Architecture for XML Binding (JAXB)[15]. The initialization and usage of JAXB is provided by the class JaxbUtils within the PRIME Core, however in order for a class to be marshalled it must be marked with XML-tags which indicates the elements of the class. There are many different types of messages within the core. The Log Reader API makes use of the messages shown in figure 4.2 as tilted boxes.

On a return message follows a response status code (like on an ordinary web page). The status code can be one of the following for the Log Reader API:

- 200 OK, which means the request was successful.
- 400 Bad Request, if the parameters were incorrect.
- 404 Not Found, occurs when an entry ID does not match an entry in the log. Following this code is also a status message in order for the Client to distinguish between an entry not found and the service is not available.

At first the Log Reader API was created using it is own Web Service but was later moved to an existing one called Common. The Common Web Service is usually not password protected and fits well with the implemented methods, *getLogEntry* and *getLatestEntryID*.

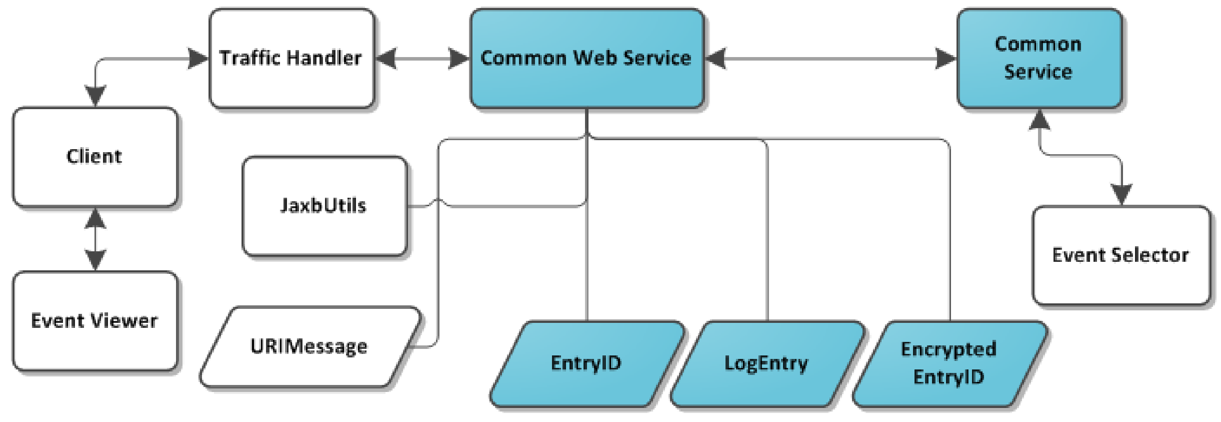


Figure 4.2: An overview of the involved parts of the Log Reader API. Tilted boxes are messages sent or received by the web service.

The boxes in figure 4.2 shows the involved parts of the Log Reader API. The shaded boxes are new or modified parts while the white boxes represent components using or being used by the API. Below is a short description of each of the components.

- **Event Viewer** and **Client**: The Event Viewer uses the Client which in turn makes requests to the server parts of the PRIME Core when needed. When the Client calls the Log Reader API, the Client marshalls either a URIMessage message containing the Client's URI or an EntryID message containing the entry ID of the desired log entry.

- **Traffic Handler**: The Jetty server catches the initial HTTP-call from the Client, however the processing of the call is done by the Traffic Handler. It confirms if the web service is available and the necessary parameters have been provided. It then calls the requested method in the web service with two arguments, one containing the parameters to the web service and the other is to let the web service send a response back.

- **Common Web Service**: This is the component that unmarshalls the message sent by the Client and calls the Common Service using the contents of the message. If the response from the Common Service is valid, a response with the status 200 OK and with the contents of either a LogEntry- or EncryptedEntryID-message. If the response from the Common Service is invalid an error response code is returned.
- **Common Service**: This component was modified to satisfy the requirements of a web service within the PRIME Core. It only relays the requests from the web service to the Event Selector.
- **Event Selector**: The component that fetches a log entry or entry ID from the log.
- **URIMessage**: A message containing the Uniform Resource Identifier for a data subject. The message is sent to the server when requesting the latest entry ID of the data subject.
- **EntryID**: This message is used when sending an unencrypted entry ID to the server when requesting a log entry.
- **EncryptedEntryID**: The same as an EntryID-message, except the contents of the message is encrypted with a data subject's public key. This is the response-message from a request to the *getLatestEntryID* method.
- **LogEntry**: An entry of the log is contained within this message, received by the Client after a *getLogEntry* request. This class has been modified in order to be

marshalled.

When the Log Reader API had been created the Client needed to be updated in order to use the new API. This resulted in errors in the JUnit Tests (see section 4.1.7) because these tests did not start the PRIME Core prior to calling the Client's methods which now used the API. The concerned tests were rewritten to start the PRIME Core prior to running. Also new tests were created to ensure that the API returned the same results as direct calls to the Event Selector.

4.2.5 Creating the Real LogData Class

The LogData is the data stored within a log entry and had previously only contained an array of bytes to represent the data. The goal of this task was to lessen the work of logging by creating a LogData class that handled the copying of a ClaimRequest, Claims and a text explaining the reason of the logging. Different constructors can be used dependent on what kind of data should be logged, for example a Claim and a reason can be used when initializing a new data subject.

It is possible that the kind of data contained within a LogData object will change. However the current class will store a timestamp when an object is created, which we feel is a requirement for all future logging. Another important parameter is the reason which will give the user information about why the data was accessed.

4.2.6 Second Form of Validation

In order to validate the privacy-friendly secure log the Client downloads all the entries from the server using *getAllEntriesSecurely* (see section 4.1.6 Log Integrity Validation). The full validation thereby uses a lot of bandwidth if a user has a lot of entries in the log. During this task a less bandwidth-intensive validation was created which only downloads the two latest entries of the log.

This second form of validation works very similar to the first, by using the methods created during the Log Integrity Validation-task. First the two latest entries are downloaded and then the methods *validateDataSubjectChain*, *validateSignature*, *validateGetLatestEntryID* and *validateEntryList* are used. It should be noted that if the Client have not previously downloaded either the last or second to last log entry, then the method *validateEntryList* will be unable to recognize a change in the server parts of those log entries.

The second to last log entry is downloaded by first fetching the last entry ID using the *getLatestEntryID* method. Then the client calculates the possible entry IDs in order up to the latest entry ID using its initial secret. When the latest entry ID is matched to the one fetched from the server, the second to last entry ID is used as a parameter to *getLogEntry*, which then returns the second to last log entry for the data subject.

4.3 Milestone 3 - Investigate HSQLDB

As was mentioned earlier, knowing the order of the entries in the log is sensitive since it could allow an attacker to correlate it with other information, such as the Apache access log, resulting in the attacker being able to link several entries to subjects. Another less sensitive example is if the entries in the KeyHandler or LogState tables were in the order they got inserted, allowing an attacker to deduce roughly when a subject was added to the system. The purpose of this task was to investigate what effects the choice of HSQLDB as the database has on the system from a security and privacy perspective.

The results presented in the following subsections were found through direct experimentation.

4.3.1 HSQLDB Modes and Types of Tables

With the exception of the HSQLDB mode for running a database in memory only (never saving to persistent storage), our findings show that it is the type of table that matters and not the mode HSQLDB is run in. Furthermore, we determined earlier in section 4.1.8 that we want HSQLDB to be run in-process mode. From here onwards, unless otherwise stated, assume that HSQLDB is running in in-process mode and that each table has its own database.

HSQLDB has four different types of tables: temp, memory, cached and text. Temp tables are temporary while the other three are persistent. This means we can rule out the temp tables, since we need persistent storage of our tables. Memory tables are entirely held in memory but have their data written to a "<database>.script" file as changes are made to it. For performance reasons, memory tables are inadequate for us. That leaves cached and text tables.

Cached tables have part of their data cached in memory with the bulk stored on disk. Changes made to a cached table are first done in memory and then written to a "<database>.log" file. The structure, and some settings, are written to a "<database>.script" file. The time it takes for changes made in memory to be synchronized with the file system is determined by the "WRITE_DELAY" property, which has a default value of 10 seconds. When the database is properly shut down, or a "CHECKPOINT" command is issued, the current state of the database is saved to a "<database>.data" file, a compressed backup of the data file is created named "<database>.backup" and the log file is cleared. A maximum size of the log file can also be set to trigger a checkpoint. The data file is a binary file using an unspecified format that has recently changed between HSQLDB versions[13]. When tables are modified the data file can become fragmented, leaving old values still in the data file until a fragmentation is triggered. This uncertainty with what is actually stored in the data file together with its relatively unknown and changing format made us rule out cached tables.

Text tables, like cached tables, have part of their data cached in memory with the bulk stored on disk. The main difference compared to cached tables is that the data file is replaced with a regular text file for each table in the database. In a text table file each line contains an entry in the log where each field of an entry is separated by some delimiter. Unlike cached tables, no backups of the text files used to store the data is created. While both log and script files are created for text tables, only structure and settings are written to the files, no actual data stored in the tables. Doing a checkpoint for a text table clears the log from all settings written. The write delay setting only determines the delay for settings being written to the log file, the actual changes to the data in a table is written directly to the text table source file regardless of settings. By inspecting a text table source file it is clear that HSQLDB adds new entries to the table by appending them to the end of the file.

4.3.2 Mitigation

While we have found that the text table type stores its data in a simple format and does not duplicate data in any other file, the entries in the source text files are still in chronological order. There does not seem to be any way, beyond modifying the source code, to make HSQLDB write entries in a different order. As was suggested by our advisor, one solution therefore is to randomize the order of the entries in the source files without involving HSQLDB.

Our mitigation for the weakness of HSQLDB to store entries in sequential order is the class Shuffler. Shuffler is used by all our classes that uses HSQLDB for storage in their respective methods that adds entries to a table. Each time a new entry is added there is a chance (specified by a config value) that a shuffle will be triggered. A shuffle swaps the positions of up to a configured value number of entries in a text table source file. The shuffle is done in the following steps:

1. We have two files; the original and a temporary one. Starting from the top of the original file, for each line toss a coin and do one of two things:

   (a) Write the line to the temporary file.

   (b) If there are still room in our buffer (the size is a config value) save the line to the buffer, otherwise write it to the temporary file.

2. Shuffle the order of the entries in the buffer.

3. Write the entries in the buffer to the end of the temporary file.

4. Replace the original file with the temporary file.

Naturally, HSQLDB does not like having its underlying data source be tampered with. Before a shuffle is made the source file for the table being shuffled is disconnected from the table by a SQL command. After the shuffle is done, the table is reconnected to the now shuffled source file and HSQLDB rebuilds its index and cache. Note that we also have to issue a checkpoint command after reconnecting the source to clear the log, since the SQL commands used to manipulate the source of the tables are written to the log.

4.4 Summary

This chapter described in detail our work. It was divided into three milestones; update standalone, implement into the PRIME Core and investigate HSQLDB. We started with updating the standalone logging module and then integrated it into the PRIME Core. Part of the integration was to implement the API and the sharing of secrets, the biggest individual tasks. We ended our work with investigating the affect of selecting HSQLDB on our logging module and came up with some mitigation for the problems found.

Chapter 5

Results and Evaluation

This chapter starts with a big example that highlights the components we have contributed to or modified in the PRIME Core. The next section evaluates each of the components highlighted in the example. The last two sections discuss some of the issues encountered when going from design to implementation and evaluates the performance of our logging module and JUnit tests.

5.1 Putting Our Work Into Context

Figure 5.1 shows an example of how the PRIME Core is used with a focus on the communication between a number of different components. The browser and application are PRIME aware with the help of plugins. The shaded components in the figure are modified or heavily contributed to by our work in this thesis. An explanation of all the steps in the figure follows.

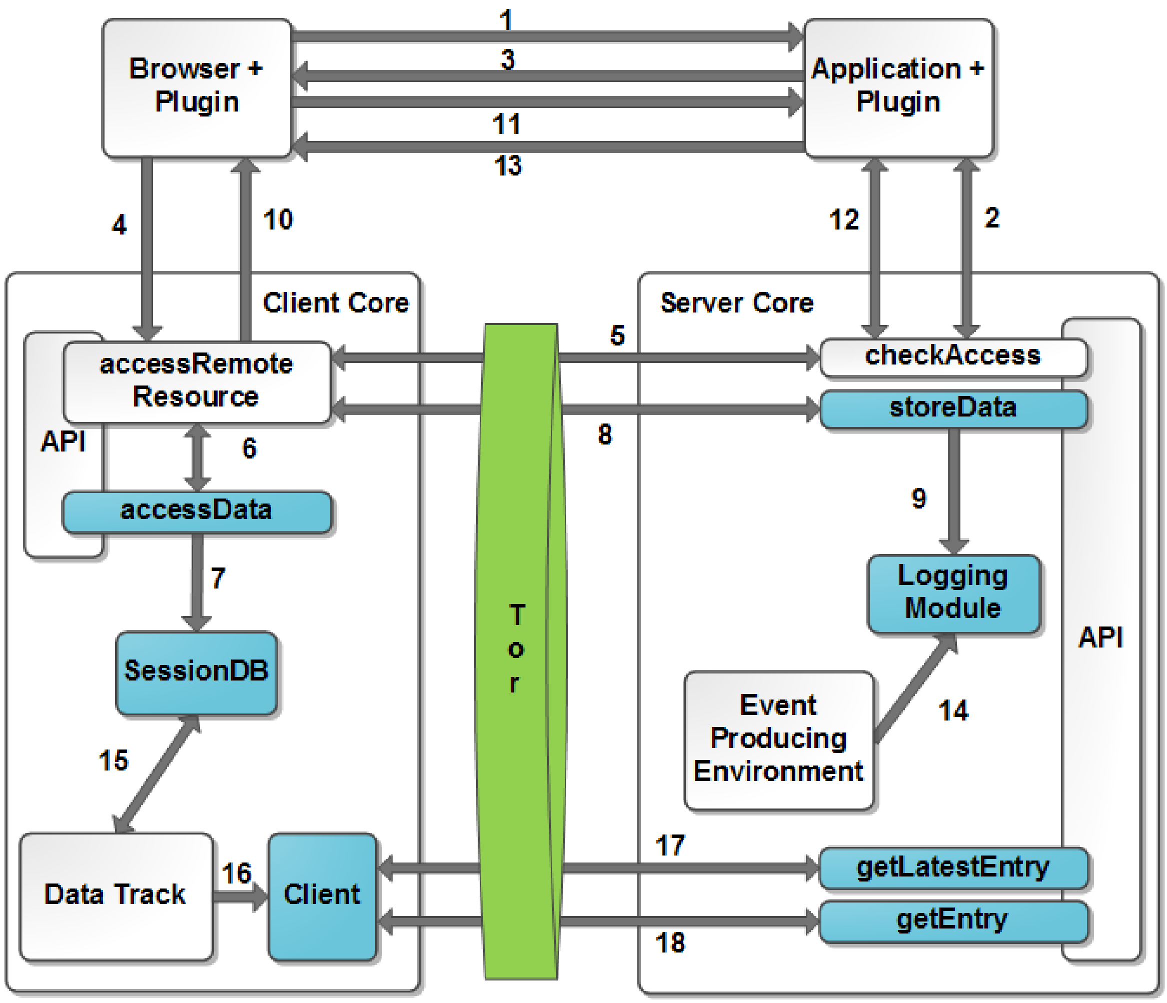


Figure 5.1: A big example of how parts of the PRIME Core is used with components we have contributed to or modified highlighted as shaded boxes. Based on the PRIME communication flow[21] and the Prime Architecture[1].

1. A browser sends a request to the remote application (for example, an Apache server hosting a website) to access some URL using a regular GET request.

2. The application's plugin maps the URL to a resource and asks the server's PRIME Core if the requester (an empty subject in this case) is allowed access to the re-

source. The *checkAccess* method responds that further information is required for the requester to gain access by returning an AskFurther object.

3. The plugin sends a response to the Browser containing the resource and the address of the server PRIME Core to contact.

4. The browser plugin intercepts the response and calls *accessRemoteResource* of the client PRIME Core.

5. The *accessRemoteResource* method calls the *checkAccess* method on the server PRIME Core and gets as a response an AskFurther object which contains a ClaimRequest.

6. The *accessRemoteResource* method calls *accessData* on the client PRIME Core sending the ClaimRequest to determine, depending on configuration with the help of the user or not, if any of the options in the ClaimRequest can be fulfilled and disclosed.

7. the *accessData* method determined that at least one of the options in the ClaimRequest could be fulfilled and stores the (soon to be) disclosed data as part of a new Session in the SessionDB. To enable transparency logging for the disclosed data *accessData* generates a new LogSecrets object and stores it as part of the Session. The private key in the LogSecrets is then removed and the LogSecrets is set as part of the Claim with the data that fulfills one of the options in the ClaimRequest.

8. The *accessData* method has returned the Claim fulfilling one of the options in the ClaimRequest to *accessRemoteResource* which in turn stores the Claim on the server's PRIME Core by a call to its *storeData* method.

9. The *storeData* method, on the server's PRIME Core, strips the LogSecrets from the Claim before it is stored and sends the LogSecrets to the LogModule to initialize the

new data subject.

10. The subject set in the Claim, for which there now contains the needed data to gain access to the resource for on the server side, is returned to the browser plugin.

11. The browser, with the help of the plugin, now makes a new request to the remote application to access the URL. As part of the request is a header with the subject returned from *accessRemoteResource*.

12. The application's plugin maps the URL to a resource and asks the server's PRIME Core if the requester is allowed access, this time sending the subject that was set in the header as an argument. The *checkAccess* method responds that the subject is now allowed access to the resource.

13. The resource is returned to the browser.

14. As data, that was previously disclosed to the server's PRIME Core in list item 9, is being processed by the server's PRIME Core and requested by the application, log events are created and sent to the logging module.

15. The Data Track provides an interface for the user (client side) to, among other things, manage and track their disclosed data. The Data Track reads the SessionDB to find what data has been disclosed and to whom.

16. A Client is initialized by the Data Track with the help of a Session obtained from the SessionDB. The Session contains the data subject, the LogSecrets and contact details for the remote PRIME server.

17. The Client requests the latest entry ID in the log for the data subject.

18. The Client downloads all the entries in the log for the data subject. The Client can now provide the Data Track with the contents of all the log entries and perform integrity validation on its behalf.

In the following section each shaded component will be presented and evaluated.

5.2 Evaluation of the Results

The previous section highlighted the different components we have contributed to, by creating new or modifying existing components, in the PRIME Core. The following subsections presents and evaluates each of these components.

5.2.1 The LogModule

The LogModule class represents the logging module to the rest of the system. With the exception of our Client and the API, the system deals exclusively with the LogModule class for interaction with the logging module.

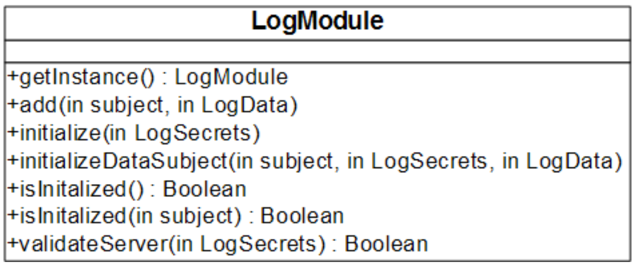

LogModule
+getInstance() : LogModule
+add(in subject, in LogData)
+initialize(in LogSecrets)
+initializeDataSubject(in subject, in LogSecrets, in LogData)
+isInitalized() : Boolean
+isInitalized(in subject) : Boolean
+validateServer(in LogSecrets) : Boolean

Figure 5.2: An UML diagram with the methods in the LogModule class.

- **getInstance()** returns the instance of the LogModule class, since it is a singleton. If there is no instance of the LogModule when the method is called an instance is created.

- **add(in subject, in LogData)** adds an entry, in the form of a LogData object, to the log for the supplied subject. The subject must have been initialized or the method will thrown an exception.
- **initialize(in LogSecrets)** initializes the LogModule itself, setting the initial server secret and seed together with the keypair to use for signing.
- **initialize(in subject, in LogSecrets, in LogData)** initializes the supplied subject in the LogModule and writes the first entry to the log for the subject.
- **isInitialized()** checks if the server has been initialized.
- **isInitialized(in subject)** checks if the supplied subject has been initialized.
- **validateServer(in LogSecrets)** validates the entire log using the supplied LogSecrets.

The LogModule itself is initialized, if needed, as soon as the PrimeCore is run. The initialization of data subjects is done as part of the sharing of secrets. This leaves little more than checking if a subject has been initialized before logging something in the log for a subject, something which can be accomplished in two lines of code. The initialization of the LogModule itself puts some requirements on the management of the LogSecrets given to the LogModule, which we encourage to not be kept on the system once saved to disk. Overall the LogModule presents a clean interface to the rest of the system with little consideration needed once the initialization is done.

5.2.2 The API

A Client may access the entries in the log by calling on the Log Reader API, which is located in the Common Web Service. The API consists of the following two methods:

- **GetLogEntry(EntryID)** takes as argument a marshalled EntryID message containing the entry ID of the entry. The method retrieves from the log a LogEntry-object corresponding to the entry ID and returns a marshalled copy of the object.
- **GetLatestEntryID(DataSubjectIdentifier)** accesses the log and retrieves the latest entry ID for the given data subject. The data subject identifier is contained within a marshalled URIMessage. The entry ID is encrypted, together with a nonce, using the data subject's public key and then returned as a marshalled EncryptedEntryID message. If the data subject does not exist a fake response which seems valid will be returned instead.

Both of these methods follows the design found in section 3.2.7 closely. However the anonymous access of the API is highly relied upon by the channel between the client and the server, as an example Figure 5.1 uses Tor to accomplish this.

One comment about the way the API and the Event Selector works is that an *EncryptedEntryID* object does not perform any kind of encryption or decryption, instead the Client has to first get the byte array from the object and further decrypt it using the *decryptDataSubjectEntryID* method (all of this is done in the Client's *getLatestEntryID* method). One alternative would be to move the contents of the *DataSubjectEntryIDWrapper*-class to the *EncryptedEntryID* class and update Crypto, Event Selector and the API accordingly.

5.2.3 The Client

The Client class provides all the functionality needed to download, obtain and validate all the entries in a server's logging module for a data subject. Note that one instance of the Client class can only manage one subject identifier. To manage multiple subjects several instances of the Client class needs to be used.

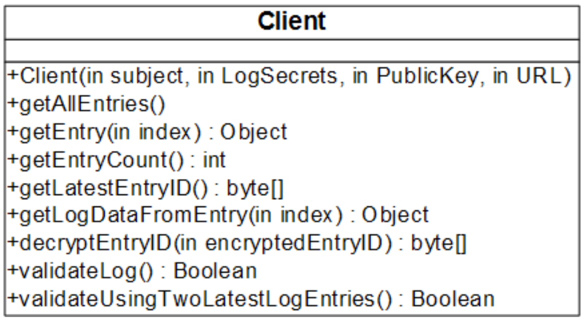

Client
+Client(in subject, in LogSecrets, in PublicKey, in URL)
+getAllEntries()
+getEntry(in index) : Object
+getEntryCount() : int
+getLatestEntryID() : byte[]
+getLogDataFromEntry(in index) : Object
+decryptEntryID(in encryptedEntryID) : byte[]
+validateLog() : Boolean
+validateUsingTwoLatestLogEntries() : Boolean

Figure 5.3: An UML diagram with the methods in the Client class.

- **Client(in subject, in LogSecrets, in PublicKey, in URL)** is the only constructor of the Client class. It initializes the Client with the data subject, the LogSecrets for the data subject, the public key of the server and the address to the server.
- **getAllEntries()** securely downloads all entries for the data subject from the server into the Client.
- **getEntry(in index)** returns the LogEntry with the supplied index. Entries in the log are indexed starting from 1.
- **getEntryCount()** returns the number of entries stored in the Client.
- **getLatestEntryID()** gets the latest entry ID for the subject from the server.
- **getLogDataFromEntry(in index)** returns the LogData from the entry with the supplied index.
- **decryptEntryID(in encryptedEntryID)** decrypts the supplied encrypted entry ID.
- **validateLog()** validates the log as outlined in section 3.2.8.

- **validateLogUsingTwoLatestLogEntries()** performs a validation of the two latest entries in the log for the data subject.[1]

While the Client has the basic functionality expected based upon what has been described in the design, it still lacks in several areas to be practically used by the Data Track. For example, there is no persistent storage of the entries downloaded. Furthermore, the validation procedure outlined in the design requires the entire log to be downloaded from the server; something which is impractical to run frequently. We started experimenting on a less resource-intense form of validation, that could be run frequently and provide decent assurance regarding the validity of the log. The Client, specifically when it comes to the interaction with the Data Track, is subject to further work.

5.2.4 Sharing Secrets

The sharing of secrets between a subject, represented by a PRIME Core in the client role, and a PRIME Core in the server role is done in the *accessData* and *storeData* methods. The secrets, on the client PRIME Core, is stored in the SessionDB as part of a Session.

- The *accessData* method is run on the PRIME Core in the client role. It was modified to, when a decision was made to disclose data, to generate a new LogSecrets object that is saved to the SessionDB and then attached to the Claim to be disclosed.

- The *storeData* method is run on the PRIME Core in the server role. It was modified to, when a LogSecrets object was attached to the Claim to be stored, enable transparency logging for the data subject by passing the LogSecrets to the logging module.

[1]This is an experimental method that was developed at the end of our work. Further work will take place.

- The SessionDB, and the Session class which objects are stored in the SessionDB, were modified to also be able to store a LogSecrets object as part of a Session.

Few methods and classes had to be modified to enable the sharing of secrets. There are however some potential problems with the approach taken. According to comments in the source code, *accessData* is at some point in the future suppose to be able to return multiple Claims containing data fulfilling a ClaimRequest. While this would have a significantly larger impact on other parts of the system, our exchange of secrets are also affected and would require changes to be made. For now little can be done by us, since the consequences of such a change to *accessData* would lead to a number of different methods changing in the PRIME Core, it is however noted as a potential problem in the future.

Another problem which is yet to be addressed is if multiple requests to *storeData* are made under the same subject. What if the second Claim contains a LogSecrets (representing a request for transparency logging to be enabled), but the first Claim did not? Right now this would enable transparency logging for the data exposed in the first Claim as well. In a similar manner, what if two Claims are sent to storeData where both Claims contains LogSecrets? Which LogSecrets object should storeData pick, assuming they differ, for transparency logging? These questions are subject to further research.

5.3 From Design to Implementation

From the design of our secure log there are, in particular, two things we rely upon that are problematic to implement:

1. That the entries in the log are stored in a multiset, and the only way to order them are through the generation of the values for some of the fields that make up an entry (primarily the EntryID and ServerID fields).

2. That authentication keys can be overwritten and irretrievably deleted after being used.

If the entries in the log could be ordered chronologically by an attacker, then he or she could correlate the log with other sources of information (like an access log). This would allow the attacker to gain further information about the log such as which subject each entry belongs to or perhaps part of entries contents. If authentication keys cannot be overwritten and irretrievably deleted then we cannot make any assumptions about the integrity of the entries committed to the log prior to an attacker compromising the log.

With our work on the Shuffler and HSQLDB, which will be evaluated later, we worked primarily on ensuring that the order of the entries (in the LogStore, LogState and KeyHandler) was not chronological but to a configurable degree shuffled. Furthermore, we found a configuration for HSQLDB that ensured that no data duplication was taking place on disk in the different files used by HSQLDB. While this is a good first step, there are further factors that complicate the realization of points 1 and 2.

Each time a shuffle is done by the Shuffler, a new file is created to replace the old file deleting the old file in the process. In several of the file systems commonly used by Windows, Linux and Mac OS X[2], deleting a file does not actually delete the data on disk, it merely removes some pointer (or equivalent) and flags the area on the disk as free leaving the old data in place until overwritten.

Generally, recovery of non-overwritten data from file systems is trivial and secure deletion is a common topic for research[32, 38]. Unfortunately for us the occurrence of data remanence is not limited to file systems but applies equally well to memory[27]. Even when a computer has been powered off DRAM retains its contents for several seconds, even if removed from the motherboard, as shown by a widely discussed paper titled "Lest We

[2]References: NTFS[24, 31], FAT32[31], Ext2[26] and HFS+[25].

Remember: Cold Boot Attacks on Encryption Keys”[29].

We have not looked into the implications of forensics on disk or memory. Primarily point 2, but also point 1 to a degree, are highly likely to be negatively affected. Mitigating the threat posed by disk and memory forensics is probably going to be further complicated by the fact that our system is written in Java, where all the code is executed in a JVM without direct access to the underlying system. Furthermore, the PRIME Core can right now be run on a number of different platforms. Further research is needed into forensics countermeasures to ensure the security of our implemented logging module.

5.3.1 HSQLDB

The logging module uses HSQLDB for data storage. To mitigate security problems due to how HSQLDB stores its data, as was discussed in section 4.3, we created the Shuffler class.

The Shuffler class introduces a performance versus security trade-off. From a security perspective you would want frequent shuffles of the source files together with a big buffer of entries to shuffle each time new entries are added to source tables. On the other hand, each shuffle is resource intense requiring the entire table to be read from and written to disk. Furthermore HSQLDB has to recreate its cache and indexes after each shuffle. From a performance perspective its clear that the number of shuffles should be kept to a minimum.

How effective the Shuffler is as mitigation against an attacker using the underlying data sources for deducing the order in which subjects were added into the system (LogStore and KeyHandler), or to correlate the entries in the log with some other source of information (LogStore), has not been looked into in detail. Further research is needed to determine the best way to implement the Shuffler, as a concept however we have shown that it is possible to use HSQLDB as the underlying database since text tables provide us with an easy way to modify the structure of the data stored on disk.

5.4 Performance and JUnit Tests

During the course of this thesis it was quickly realized that some of the functionality of the log would consume a lot of resources. In particular the generation of LogSecrets for each data subject. The creation of these secrets is done on the client side of the core to make sure that the secrets remain hidden and to keep the time-consuming task of generating the secrets away from the server.

5.4.1 Server Log Performance

A task that is performed on the server is writing to the privacy-friendly secure log. To test the performance of this, a new test was created which inserted 150 entries into the log and printed out the insertion-time for each log entry. Figure 5.4 shows a graph with the results from the test. The x-axis shows the number of entries inserted into the log while the y-axis shows the time in milliseconds for each new log entry inserted.

The graph shows that most entries are inserted relatively quickly into the log, compared to a few which seems to slow down the logging greatly. Those few are so much slower because of the shuffler created during the HSQLDB Milestone (see section 4.3) runs when those entries are inserted. During this test, the shuffler had a 20% chance of shuffling a maximum of 50 log entries within the log. It is clear from this test that further work is needed to improve the shuffler's performance.

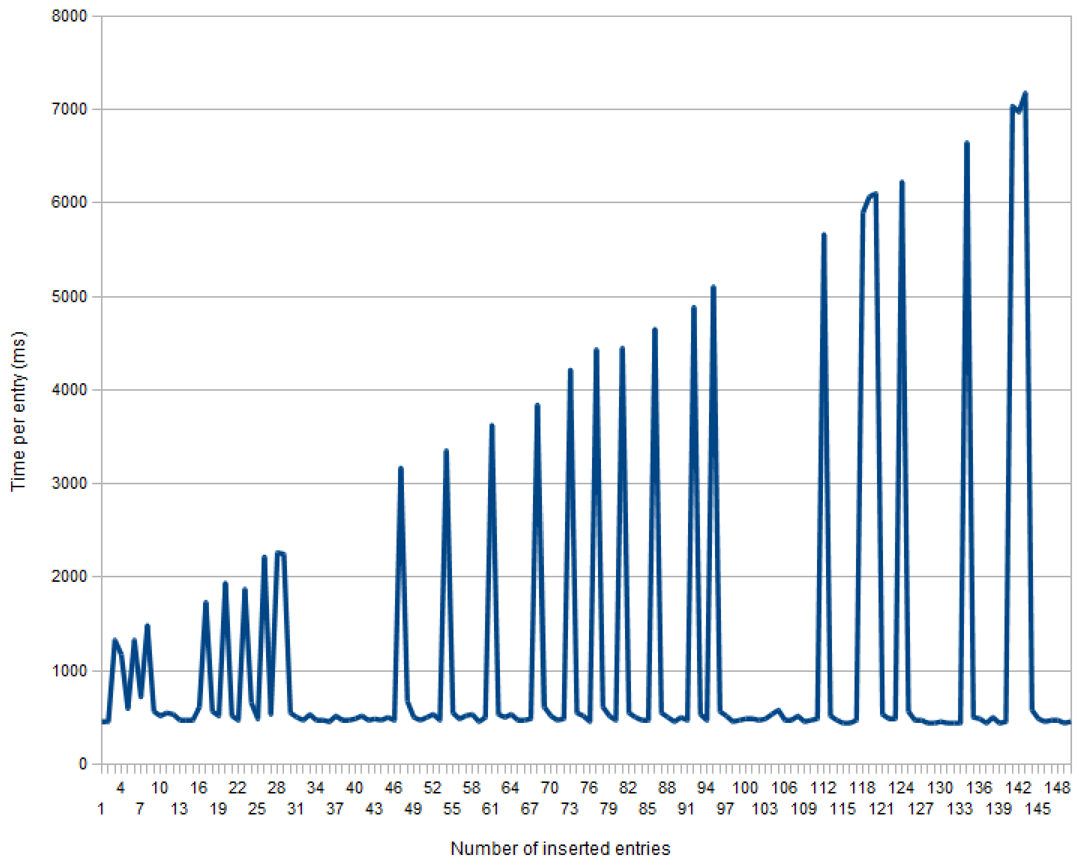


Figure 5.4: The graph shows the insertion-time for each of the 150 entries inserted into the log during this test. The shuffler for HSQLDB is set to have a 20% chance of running with a maximum shuffling of 50 entries.

It is possible to increase performance of the log by disabling the shuffler (setting the chance of a shuffle to 0%). Another test was run with the shuffler disabled, in which 500 log entries were inserted. The results are presented in Figure 5.5.

In this test the insertion-time for all entries is within the same range, meaning that the log does not slow down significantly when containing a lot of entries. However to remove the use of the shuffler does cause the HSQLDB to remember the order of inserted log entries, which potentially breaks the requirement of unlinkability (discussed in section

3.2.3 *Order of Entries In the Log and ID Generation*).

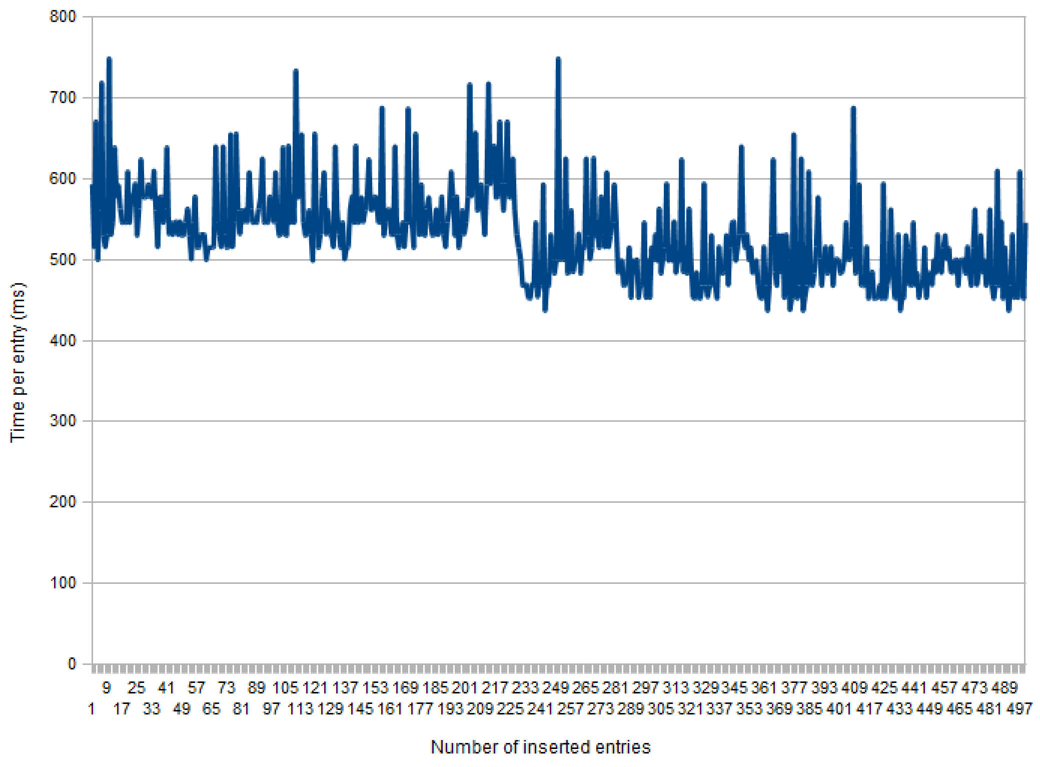


Figure 5.5: The graph shows the insertion-time for each of the 500 entries inserted into the log during this test. The shuffler for the HSQLDB is turned off for this test.

5.4.2 JUnit test performance

In additon to the logging module, several tests were created to ensure that the log was functioning correctly. The list below summarizes the tests:

- **LogModuleTest** is a group of tests that consists of a few minor tests and a major one. The minor tests controls that methods within the LogModule throws exceptions as intended. For example, if the LogModule is initialized a second time it should throw an AlreadyInitializedException because the log has already been initialized.

The major test ensures that the validation (both full validation and second form of validation) works and that logged data can be retrieved from the secure log and that it is correct.

- **CryptoTest** is a group of tests designed to test the Crypto-class. They check that the methods of Crypto is able to sign log data, create hashes, perform encryption and decryption, store and load secrets.
- **LogReaderTest** is a test to ensure that the Log Reader API is able to return log entries and also checks whether they are correct.

During the design of these tests we wanted to test as much as possible to ensure that the secure log was functioning correctly when we had implemented it into the PRIME Core. Because of this performance was not a big concern for the most part of this thesis. It was only after we had finished most of the implementation that we sped up these tests. This was done by not generating new secrets but loading them from a file instead.

Missing is a test for the sharing of secrets and keys between the client and the server (implemented during section 4.2.3). This is considered as future work (see section 6.2).

5.5 Summary

The chapter put our work into context and evaluated it, giving a clear picture of what we have contributed to the PRIME Core and what role it serves. We concluded that further research is needed into forensics countermeasures to ensure the security of our implemented logging module. The chapter ended with a look at the performance of our logging module and JUnit tests, showing that the Shuffler's performance leaves a lot to be desired.

Chapter 6

Conclusion

In this thesis we have described our work with implementing, and enhancing, a privacy-friendly secure logging module into the PRIME Core. Chapter 2 introduced the reader to the PrimeLife project and the PRIME Core. Furthermore the chapter provided a short overview of a number of different cryptographic primitives used when designing secure logs and ended with a look at the Kelsey-Schneier secure log. In chapter 3 the work done prior to this thesis was presented with a focus on the design. A short description of the developed prototype was also provided. Chapter 4 described our work in this thesis in detail, where the chapter was divided into three milestones each with a clear goal in mind. In chapter 5 we put our work into context highlighting, describing and evaluating the different components we had contributed to or modified in the PRIME Core. We ended the chapter with evaluating our tests and looking at the performance of our logging module. This chapter concludes the thesis.

6.1 Results

We successfully completed our primary and secondary goal. The logging module is now fully implemented and integrated with the PRIME Core. We came to the conclusion that it is possible to configure and manipulate HSQLDB in such a way that it can be used as a database for storing log entries without revealing the order in which the entries were inserted into the log. The performance of the Shuffler, used to manipulate HSQLDB's data source, is lacking and subject to future work.

6.2 Problems

At the beginning of Milestone 2 we set up our own server and downloaded the PRIME Core to it. We then worked on that version of the PRIME Core throughout most of our thesis. This decision was made to eliminate temporary problems with our privacy-friendly secure logging module while we were implementing it. This caused problems for us during our implementation because the version we downloaded had some issues concerning performance and errors in tests. It would have been a good idea to update the PRIME Core as we went along with our implementation.

6.3 Future Work

Here a few areas of future work are presented:

- An optimal logging strategy needs to be created to make sure all events of interest will be logged. Also new kinds of LogData might need to be created for future needs.
- Investigations into how well the authentication keys are overwritten in memory, since

the keys will at some point be there.

- The performance of the logging module is currently very poor due to the shuffling of log entries by the Shuffler class. Further development is needed to improve its performance.
- In order for the user to be able to track their disclosed data, the Client needs to be integrated with the Data Track.
- A test for sharing secrets and keys between client and server parts of the PRIME Core.

6.4 Final Words

The implementation of a privacy-friendly secure logging module into the PRIME Core makes it possible for tools such as the Data Track to help users verify that their personal data is being handled in accordance with agreed-upon policies and laws. Although the PRIME Core is a prototype it is our hope that the PrimeLife project will continue to inspire organizations and businesses to focus more on the privacy of its users.

Bibliography

[1] Prime architecture - version 2 (29 march, 2007) https://www.prime-project.eu/prime_products/reports/arch/pub_del_D14.2.c_ec_WP14.2_v1_Final.pdf.

[2] Prime white paper version 3 https://www.prime-project.eu/prime_products/whitepaper/PRIME-Whitepaper-V3.pdf.

[3] Primelife - privacy and identity management in europe for life http://www.primelife.eu/.

[4] Cipher - wikipedia, the free encyclopedia http://en.wikipedia.org/w/index.php?title=Cipher&oldid=317382555, 10 2009.

[5] Code conventions for the java programming language http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html, 11 2009.

[6] Coding conventions - wikipedia, the free encyclopedia http://en.wikipedia.org/w/index.php?title=Coding_conventions&oldid=324105279, 11 2009.

[7] Cryptographic hash function - wikipedia, the free encyclopedia http://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=318420134, 10 2009.

[8] Cryptographic primitive - wikipedia, the free encyclopedia http://en.wikipedia.org/w/index.php?title=Cryptographic_primitive&oldid=290307861, 10 2009.

[9] Digital signature - wikipedia, the free encyclopedia http://en.wikipedia.org/w/index.php?title=Digital_signature&oldid=319298362, 10 2009.

[10] Encrypting file system - wikipedia, the free encyclopedia http://en.wikipedia.org/w/index.php?title=Encrypting_File_System&oldid=331770835, 12 2009.

[11] Encryption - wikipedia, the free encyclopedia http://en.wikipedia.org/w/index.php?title=Encryption&oldid=317243048, 10 2009.

[12] Hsqldb http://hsqldb.org/, 12 2009.

[13] Hsqldb user guide `http://hsqldb.org/doc/guide/`, 11 2009.

[14] J2e 5.0 `http://java.sun.com/j2se/1.5.0/`, 12 2009.

[15] Java architecture for xml binding - wikipedia, the free encyclopedia `http://en.wikipedia.org/w/index.php?title=Java_Architecture_for_XML_Binding&oldid=325455928`, 11 2009.

[16] Jetty `http://www.eclipse.org/jetty/`, 11 2009.

[17] Junit - wikipedia, the free encyclopedia `http://en.wikipedia.org/w/index.php?title=JUnit&oldid=323859817`, 11 2009.

[18] Message authentication code - wikipedia, the free encyclopedia `http://en.wikipedia.org/w/index.php?title=Message_authentication_code&oldid=316624141`, 10 2009.

[19] Multiset - wikipedia, the free encyclopedia `http://en.wikipedia.org/w/index.php?title=Special:Cite&page=Multiset&id=310727642`, 12 2009.

[20] Naming conventions (programming) - wikipedia, the free encyclopedia `http://en.wikipedia.org/w/index.php?title=Naming_conventions_%28programming%29&oldid=319979206`, 11 2009.

[21] Prime communication flow with integrated prototype version 2 `http://blues.inf.tu-dresden.de/prime/AdvTv3/AdvTv3/Content/Prototypes/Integrated%20Prototype/prime.html`, 11 2009.

[22] Web services glossary `http://www.w3.org/TR/ws-gloss/`, 11 2009.

[23] Hmac - wikipedia, the free encyclopedia `http://en.wikipedia.org/w/index.php?title=HMAC&oldid=336183553`, 01 2010.

[24] Ntfs.com file recovery concepts & products `http://www.ntfs.com/file-recovery-concepts.htm`, 01 2010.

[25] Aaron Burghardt and Adam J. Feldman. Using the hfs+ journal for deleted file recovery. *Digital Investigation*, 5(SUPPL.):S76 – S82, 2008. And recovery;Current techniques;Deleted;Deleted files;File;File carving;File I/O;File systems;HFS+;Journal;Mac OS X;Recovery;Research and analysis;Tools and techniques;.

[26] Philip Craiger. Recovering digital evidence from linux systems. *IFIP International Federation for Information Processing, ISSN: 1571-5736 (Print) 1861-2288 (Online), ISBN 978-0-387-30012-2*, 2005.

[27] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. pages 77 – 89, Berkeley, CA, USA, 1996//. secure data deletion;encryption systems;sensitive data access;erased data recovery;security attacks;magnetic media;random-access memory;.

[28] P. Hjärtquist & A. Lavén H. Hedbom, T. Pulls. Adding secure transparency logging to the prime core. *In Post-Proceedings of the Fifth International Summer School: Privacy and Identity Management for Life, Nice, France, 7th ? 11th September, 2009. To Appear.*

[29] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91 – 98, 2009. Cooling technique;Cryptographic key;Encryption key;Main memory;Memory retention;Operating systems;Room temperature;Special devices;System memory;.

[30] J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian workshops on Grid computing and e-research - Volume 54*, volume 167 of *ACM International Conference Proceeding Series*, pages 203–211. Australian Computer Society, 2006.

[31] Yu Meisheng Huang Wei. The quickly solving method of file recovery in windows environment. *2008 International Conference on Computer Science and Software Engineering*, 2008.

[32] Nikolai Joukov, Harry Papaxenopoulos, and Erez Zadok. Secure deletion myths, issues, and solutions. pages 61 – 66, Alexandria, VA, United states, 2006. Data overwriting tools;File systems;Secure deletion;Unintended data recovery;.

[33] Di Ma and G. Tsudik. Forward-secure sequential aggregate authentication. *2007 IEEE Symposium on Security and Privacy (SP '07), p 95-100*, 2007.

[34] Di Ma and G. Tsudik. A new approach to secure logging. *Data and Applications Security XXII. 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security, p 48-6*, 2008.

[35] S. Sackmann, J. Strüker, and R. Accorsi. Personalization in privacy-aware highly dynamic systems. *COMMUNICATIONS OF THE ACM*, 49(9), September 2006.

[36] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. *The Seventh USENIX Security Symposium Proceedings, USENIX Press*, pages 53–62, January 1998.

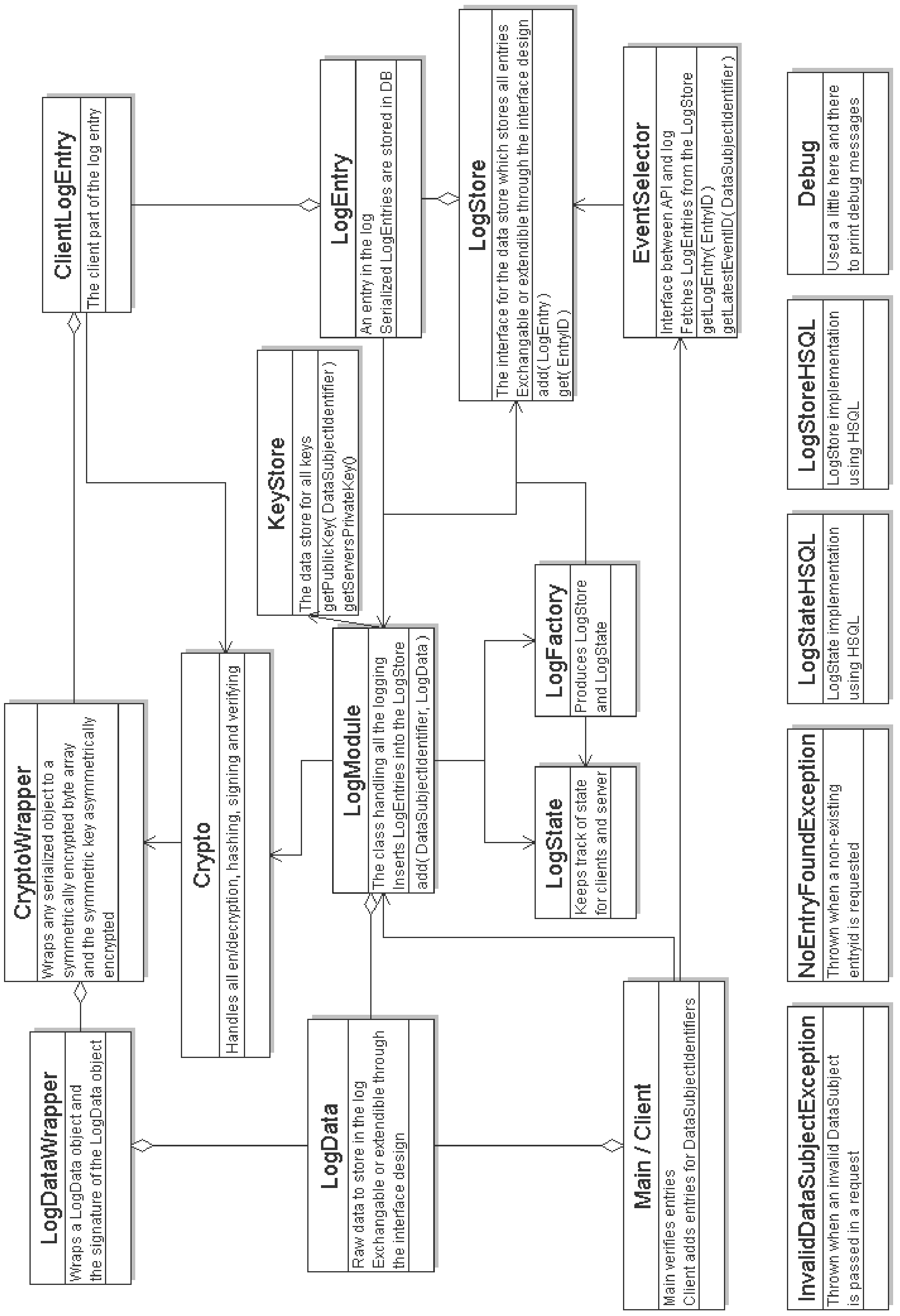
[37] Bruce Schneier. *Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C.* John Wiley & Sons, Inc, isbn 0471128457 edition, 01/01/96.

[38] Craig Wright, Dave Kleiman, and Shyaam Sundhar R.s. Overwriting hard drive data: The great wiping controversy. volume 5352 LNCS, pages 243 – 257, Hyderabad, India, 2008. Data Wipe;Digital Forensics;Format;Magnetic (CE);Secure Wipe;.

Appendix A

Prototype Class Diagram

The class diagram generated at the end of the prototype development in the Topics in Computer Security course at KAU spring 2009. Please note that some of the terminology has changed since its creation.


ClientLogEntry
The client part of the log entry
LogEntry
An entry in the log
Serialized LogEntries are stored in DB
LogStore
The interface for the data store which stores all entries
Exchangable or extendible through the interface design
add(LogEntry)
get(EntryID)
EventSelector
Interface between API and log
Fetches LogEntries from the LogStore
getLogEntry(EntryID)
getLatestEventID(DataSubjectIdentifier)
Debug
Used a little here and there
to print debug messages
KeyStore
The data store for all keys
getPublicKey(DataSubjectIdentifier)
getServersPrivateKey()
LogStoreHSQL
LogStore implementation
using HSQL
LogFactory
Produces LogStore
and LogState
LogStateHSQL
LogState implementation
using HSQL
CryptoWrapper
Wraps any serialized object to a
symmetrically encrypted byte array
and the symmetric key asymmetrically
encrypted
Crypto
Handles all en/decryption, hashing, signing and verifying
LogModule
The class handling all the logging
Inserts LogEntries into the LogStore
add(DataSubjectIdentifier, LogData)
LogState
Keeps track of state
for clients and server
NoEntryFoundException
Thrown when a non-existing
entryid is requested
LogDataWrapper
Wraps a LogData object and
the signature of the LogData object
LogData
Raw data to store in the log
Exchangable or extendible through
the interface design
Main / Client
Main verifies entries
Client adds entries for DataSubjectIdentifiers
InvalidDataSubjectException
Thrown when an invalid DataSubject
is passed in a request

Appendix B

Default Configuration

The default configuration for our log in the PRIME Core.

```
kau.prime.servertransparency.secrets.initialsecret = csdsd32423s232
kau.prime.servertransparency.secrets.serveridseed = sdfsdfsdfsdf
kau.prime.servertransparency.secrets.publickey =
kau.prime.servertransparency.secrets.privatekey =
kau.prime.servertransparency.secrets.file = build/serversecrets
kau.prime.servertransparency.logging.LogStateImplementation = kau.prime.servertransparency.logging.implementation.LogStateHSQL
kau.prime.servertransparency.logging.LogStoreImplementation = kau.prime.servertransparency.logging.implementation.LogStoreHSQL
kau.prime.servertransparency.logging.KeyHandlerImplementation = kau.prime.servertransparency.logging.implementation.KeyHandlerHSQL
kau.prime.servertransparency.logging.HSQL.shuffleChance = 20
kau.prime.servertransparency.logging.HSQL.shuffleSwaps = 50
kau.prime.servertransparency.logging.HSQL.cacheScale = 10
kau.prime.servertransparency.logging.HSQL.mode = jdbc:hsqldb:file:
kau.prime.servertransparency.logging.HSQL.dbpath = build/db/
kau.prime.servertransparency.logging.HSQL.username = sa
kau.prime.servertransparency.logging.HSQL.password =
kau.prime.servertransparency.logging.KeyHandlerHSQL.file = keyhandler
kau.prime.servertransparency.logging.LogStateHSQL.file = logstate
kau.prime.servertransparency.logging.LogStoreHSQL.file = logstore
kau.prime.servertransparency.logging.Crypto.symmetric.algorithm = AES
kau.prime.servertransparency.logging.Crypto.symmetric.mode = CBC
kau.prime.servertransparency.logging.Crypto.symmetric.padding = ISO10126PADDING
kau.prime.servertransparency.logging.Crypto.symmetric.keysize = 128
kau.prime.servertransparency.logging.Crypto.asymmetric.algorithm = RSA
kau.prime.servertransparency.logging.Crypto.asymmetric.mode = ECB
kau.prime.servertransparency.logging.Crypto.asymmetric.padding = OAEPWITHSHA-512ANDMGF1PADDING
kau.prime.servertransparency.logging.Crypto.asymmetric.keysize = 3072
kau.prime.servertransparency.logging.Crypto.signature.algorithm = SHA512withRSA
kau.prime.servertransparency.logging.Crypto.hash.algorithm = SHA-512
kau.prime.servertransparency.logging.Crypto.hmac.algorithm = HmacSHA512
kau.prime.servertransparency.logging.Crypto.nonce.min = 128
kau.prime.servertransparency.logging.Crypto.nonce.max = 1000
kau.prime.servertransparency.logging.Crypto.rng.seed = ZrSfsq40szyrtWmTsdlceaa6qUmz
kau.prime.servertransparency.logging.Crypto.rng.discard = 10000
kau.prime.servertransparency.logging.Crypto.PBE.algorithm = PBEWithMD5AndDES

kau.prime.servertransparency.logging.EventSelector.entryID.size = 50
```

Appendix C

Source Code

The source code can be requested from Hans Hedbom (Hans.Hedbom@kau.se). Please note that it is currently under some restrictions, the ultimate goal however is to release as much as possible under an open source license.

Appendix D

Glossary

Below follows a list of words, abbreviations, acronyms and names used throughout this thesis. In section 3.3 the different classes used in the proof of concept prototype, developed as part of the Topics in Computer Security course, are explained in detail.

AES - A symmetric cipher. See section 2.2.2.

API - Application Programming Interface.

DataSubjectChain - The data subject's MAC chain part of every entry in our privacy-friendly secure log. See section 3.2.5.

DSS - Data Subject Secret. See section 3.2.2.

EntryID - The identifier for the entry in our privacy-friendly secure log used by the data subject. See section 3.2.3.

GetLogEntry - The API method used to download an entry from our privacy-friendly secure log. See section 3.2.7.

GetLatestEntryID - The API method used to get the latest entry identifier for a data subject from our privacy-friendly secure log. See section 3.2.7.

HMAC - Hash-based Message Authentication Code. See section 2.2.1.

HSQLDB - A relational database written in Java.

ID - Identifier.

JUnit - A unit testing framework for Java, see section 4.1.7.

KeyHandler - The class that manages all the keys used in our privacy-friendly secure log. It now declares an interface and KeyHandlerHSQL implements it. Used to be called KeyStore.

LogSecrets - A class containing secrets in our logging scheme, see section 4.2.2.

MAC - Message Authentication Code. See section 2.2.1.

Nonce - A *n*umber used *once*.

PII - Personally Identifiable Information.

RSA - An asymmetric cipher. See section 2.2.2.

SAS - Server Authentication Secret. See section 3.2.2.

ServerChain - The server's MAC chain part of every entry in our privacy-friendly secure log. See section 3.2.5.

ServerID - The identifier for the entry in our privacy-friendly secure log used by the server. See section 3.2.3.

SVN - Short for Subversion, a version-control system used primarily for software development.

TLS/SSL - Transport Layer Security/Secure Sockets Layer, used interchangeably in this thesis. Provides secure communication between two endpoints.

Tor - The onion router, an anonymity network.

URI - Uniform Resource Identifier.