Christian Grahn

Implementing Transparency Logging for an Issue Tracking System

Computer Science
E-level thesis, 30 ECTS credit points

Date/Term: 2012-06-11
Supervisor: Simone Fischer-Hübner
Examiner: Donald F. Ross
Serial Number: E2012:01
Implementing Transparency Logging for an Issue Tracking System

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

Christian Grahn

Approved, June 11th, 2012

Opponent: Olimpia Censurato

Advisor: Simone Fischer-Hübner

Examiner: Donald F. Ross
Abstract

On the Internet today, users are accustomed to disclosing personal information when accessing a new service. When a user does so, there is rarely a system in place which allows the user to monitor how his or her information is actually shared or used by services. One proposed solution to this problem is to have services perform transparency logging on behalf of users, informing them how their data is processed as processing is taking place.

We have recently participated in a collaboration to develop a privacy-preserving secure logging scheme that can be used for the purpose of transparency logging. As part of that collaboration we created a proof of concept implementation. In this thesis, we elaborate on that implementation and integrate it with a minimalistic open source issue-tracking system. We evaluate the amount of work required to integrate the logging system and attempt to identify potential integration problems. Using this issue-tracking system we then design and implement a scenario that demonstrates the value of the logging system to the average user.
I want to thank my technical supervisor at Karlstad University, Tobias Pulls, for his exemplary support and guidance over the last year. I would also like to express my gratitude to Karel Wouters and Jo Vliegen for allowing me to take part in this interesting project, and for helping me to get started. Finally, I want to thank my advisor Simone Fischer-Hübner.
# Contents

1 Introduction 1

1.1 Motivation and goals ........................................ 2
1.2 Disposition .................................................. 3

2 Background 5

2.1 Distributed privacy-preserving secure log trails ............... 6
  2.1.1 Security requirements ...................................... 7
  2.1.2 Usage .................................................... 9
  2.1.3 Log entry meta data ....................................... 11
  2.1.4 Branching by cascading ................................... 12
  2.1.5 Auditing .................................................. 13
2.2 Logging scheme implementation ................................ 15
  2.2.1 Data subject .............................................. 16
  2.2.2 Data processor ........................................... 17
2.3 Demonstrative scenario ......................................... 19
  2.3.1 osTicket ................................................. 19
  2.3.2 Configuration ............................................. 20
  2.3.3 Procedure ............................................... 21
2.4 Summary .................................................... 23
3 Implementation

3.1 Audit components ................................................. 25
  3.1.1 The log server ............................................. 26
  3.1.2 The data processor ......................................... 27
  3.1.3 TimeStamping Authority ................................... 27
3.2 The data processor ............................................. 28
  3.2.1 Interconnection ............................................. 29
  3.2.2 osTicket .................................................. 30
  3.2.3 Data processor’s Java component ........................... 36
3.3 The data subject .............................................. 37
  3.3.1 Interconnection ............................................. 38
  3.3.2 Chrome extension .......................................... 39
  3.3.3 Java component ............................................ 43
3.4 Summary ..................................................... 44

4 Results

4.1 Setting up the scenario ...................................... 47
4.2 Performing the scenario ...................................... 48
4.3 Scenario coding summary .................................... 59
4.4 Logging system suggestions .................................. 60
  4.4.1 Data vault integration .................................... 60
  4.4.2 Cascade entry generation .................................. 61
4.5 Ticket deletion ............................................... 61
  4.5.1 Policy .................................................... 62
  4.5.2 Changes to osTicket ....................................... 62
  4.5.3 Changes to the logging system ............................ 63
4.6 Performance ................................................ 63
  4.6.1 Methodology ............................................... 64
4.6.2 Performance results ........................................... 65
4.7 Summary .............................................................. 68

5 Conclusion ............................................................... 69
5.1 Implementation ....................................................... 69
5.2 Scenario ............................................................... 70
5.2.1 Scenario choice .................................................. 70
5.2.2 Privacy ............................................................ 70
5.2.3 Transparency .................................................... 71
5.3 Performance .......................................................... 71
5.4 Project evaluation .................................................... 72
5.5 Key contributions .................................................... 74
5.6 Future work .......................................................... 74
5.7 Final words .......................................................... 76

References ................................................................. 77

A Glossary ................................................................. 79

B Data processor’s Java component WSDL ......................... 81

C open.js ................................................................. 85

D osTicket alterations ................................................... 87
D.1 include/staff/cascade.inc.php .................................... 87
D.2 include/class.processor.php ...................................... 88
D.3 client.inc.php ....................................................... 91
D.4 include/class.ticket.php .......................................... 91
D.5 include/class.validator.php ...................................... 91
D.6 include/client/open.inc.php ..................................... 91
E  Logging system class diagrams  

E.1  Pre-existing  
E.1.1  Data processor  
E.1.2  Data subject  
E.2  With audit components  
E.2.1  Data processor  
E.2.2  Log server  

F  Performance tests on log server state  

xii
List of Figures

2.1 The procedure that takes place when starting a process .................................. 10
2.2 The branching procedure .................................................................................. 11
2.3 Abstract illustration of the derivation of $\text{ExpDC}(\text{P}_a)_i$ .................................. 15
2.4 The user and the three data processor entities ...................................................... 21
2.5 The entities of the scenario with numbered arrows illustrating the chronologi-
    cal procedure for performing the scenario ......................................................... 22

3.1 The online form for creating tickets, modified to accept a public key ................. 31
3.2 The page shown to the user after a ticket has been successfully created .......... 33
3.3 A part of the ticket information page showing the controls available to staff
    members ............................................................................................................ 34
3.4 The process branching page that is available to staff members ......................... 35
3.5 The button added to Google Chrome’s toolbar by the data subject extension .... 40
3.6 The log trail visualisation page, logdisplay.html ............................................. 42
3.7 The log trail visualisation page, logdisplay.html displaying graphic logs ....... 43

4.1 The osTicket ticket creation page filled in by the customer ............................... 48
4.2 The ticket creation response page when using the data subject extension ....... 49
4.3 The staff member’s view of open tickets ......................................................... 50
4.4 Responding to the customer’s complaint ......................................................... 50
4.5 The log visualisation page after the first contact with support ....................... 51
4.6 The staff member’s cascade page .................................................. 51
4.7 The service entity’s ticket creation page filled in by support staff .......... 52
4.8 The staff member’s cascade page with initialisation data entered .......... 52
4.9 The log visualisation page after the logging process has been branched ... 53
4.10 Deleting the customer’s ticket at the support entity .......................... 54
4.11 The first meta entered as a normal ticket update ............................ 55
4.12 The log visualisation page after the first meta entry has been submitted .. 56
4.13 The log visualisation page after the specialist has received the hard drive . 56
4.14 The log visualisation page after the specialist has built a disk image ....... 57
4.15 The log visualisation page after the disk image has been sent to the service entity ................................................................. 58
4.16 The log visualisation page after a new hard drive has been installed .... 58
4.17 The log visualisation page at the end of the scenario ....................... 59
4.18 The distribution of execution time in the ticket creation procedure of the modified osTicket ......................................................... 66
4.19 The distribution of execution time in the procedure for replying to an open ticket in the modified osTicket ....................................... 67
4.20 The distribution of execution time in the ticket deletion procedure of the modified osTicket ......................................................... 67
Listings

3.1 Example of a JAX-WS annotated Java method ........................................ 30
3.2 Using an Endpoint to offer a @WebService over the network .................. 30
3.3 Excerpt from open.php showing the added code to open a log trail ....... 32
3.4 A typical method from the PHP class Processor ................................. 36
3.5 An example of a JSON object returned by GetPuk ............................... 38
3.6 A typical method from the PHP class Processor ................................. 39
3.7 Part of the Chrome extension manifest where open.js is registered as a       content script ................................................................. 40
3.8 Methods used to translate a log trail from Java objects to a JSON string . 45
wsdl.txt .................................................................................... 81
open.js ..................................................................................... 85
cascade.inc.php ........................................................................ 87
class.processor.php ..................................................................... 88
List of Tables

4.1 The number of lines added to each component . . . . . . . . . . . . . . . . 60
4.2 The PHP page generation time for creating, replying to and deleting tickets in osTicket, given in milliseconds . . . . . . . . . . . . . . . . . . . . . . 65
4.3 The processing time measured in the data processor’s Java component . . 66

F.1 Timing results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
Chapter 1

Introduction

In the current online climate it is common for users to share personal information with service providers. Unfortunately, it can be difficult or impossible for users to keep track later on of how that information is processed. On the Internet, popular websites often provide privacy policies that inform the users of how information about them may be collected and used. Our work in this thesis builds on another Privacy Enhancing Technology (PET): transparency logging. Where as a privacy policy may provide valuable information before the user contracts a service, transparency logging is meant to provide users with information about their individual case after the personal information has been shared. Transparency logging is fundamentally about transparency of data processing, where the processing is done by data processors and the recipient of the logging information is the data subject to whom the processed data relates. By providing data subjects with a log of how information about them has been altered and shared, service providers allow data subjects to get more directly involved in maintaining their privacy. Getting involved can mean changing privacy settings and observing the effects in the log, or simply being aware of what information is available to whom.

We previously co-authored a technical report defining a logging scheme with a focus on user privacy [10]. Such a logging scheme is ideal for transparency logging as logging
systems with no consideration for privacy can cause serious privacy issues themselves. A log may contain more information about users than would otherwise be retained by the service provider. Some information may even be possible to infer from the encrypted data of the log. If that data can then be linked to a user by examining the data itself, or by observing the log retrieval process, the logging system will have caused a breach of privacy. A privacy aware logging scheme can prevent this scenario in several ways. For example, by decoupling the entries that form one user’s log, the attacker is prevented from gaining any information that can not be inferred from a single log entry on its own. In Chapter 2, we describe how the logging scheme we work with in this thesis provides this and other privacy features. In the technical report that defines the logging scheme, a proof of concept implementation is described; in this thesis, we add audit functionality to that implementation and then we use the implementation to provide transparency logging from an open source product.

1.1 Motivation and goals

The report by Pulls et al. [10] mentions a number of suggestions for future work on the proof of concept implementation. We have chosen to follow up on two of these suggestions. The first is the audit functionality that was left out of the implementation due to time limitations. We chose to implement this suggestion first, so that we could then view the logging components as complete, and work with them only through their APIs. The second idea for future work that we chose to realise was to integrate the logging scheme implementation with a real world product. The purpose of this integration was to find friction points in the logging scheme and its implementation in order to make suggestions for improvements. Another reason to integrate with a real world system is that the concept of transparency logging and its benefits can be difficult to get across to users. Therefore, it was our aim to create a scenario involving transparency logging that, from the users point
of view, is both simple and useful. This led us to choose issue-tracking software aimed at
customer support early on as the ideal software type for the project as it offers a service
most users will have encountered in some form.

1.2 Disposition

The remainder of this thesis is structured as follows: In Chapter 2, we present the logging
scheme that is the foundation of our work in this thesis and its proof of concept imple-
mentation. We also introduce the issue-tracking system we have chosen to integrate with
the logging system and describe a scenario that can be executed after the integration. We
proceed to describe the changes to the issue-tracking system as well as the “glue” code
between the issue-tracking and logging systems in Chapter 3. In addition to this, we de-
scribe the implementation of an application for the user that helps the user to interact
with the logging system. In Chapter 4, we show the results of the implementation in terms
of performance and the amount of work required for integration. We also show how the
scenario presented in Chapter 2 can be executed with the components described in Chapter
3. Finally, we provide some final thoughts and suggestions for future work in Chapter 5.
Chapter 2

Background

The work in this thesis builds upon the privacy preserving secure logging scheme of Pulls et al. [10]. That scheme in turn has its roots in a long history of research in secure logging where the works of Bellare and Yee [1] as well as Schneier and Kelsey [11], from 1997 and 1998 respectively, are considered seminal, though Futoransky and Kargieman [5] laid the foundation for the use of a hash chain as a means to secure a log a few years earlier.

Bellare and Yee define the forward integrity (FI) property for logs to mean that the data in an FI secure log that is written prior to the logging entity being compromised should not be susceptible to undetectable modification. They also propose a deletion detecting-FI secure logging system where the log is verified by a separate verifier.

Schneier and Kelsey define a logging scheme involving an untrusted log server, a trusted party and a set of verifiers that can review, but not change a subset of the existing log entries. The untrusted log server continually updates a local log that is occasionally transferred to, and verified by, the trusted party.

A common element of the schemes mentioned above is that they rely on an initial sharing of secrets between an untrusted and trusted entity that takes place at a point when the untrusted entity is assumed to be uncompromised. The untrusted entity then uses the shared secrets in the process of logging, occasionally evolving them in an irrevocable way.
that can be replicated by the trusted entity as it verifies the log at a later time.

The logging scheme defined by Pulls et al. is greatly influenced by two earlier secure logging schemes that both had a focus on the privacy of data subjects. These earlier schemes were defined by Wouters et al. [13] and Hedbom et al. [8, 4, 7] and the authors have since collaborated to combine the privacy features of the two schemes. The result of this collaboration is [10].

In this thesis, we have expanded upon the proof-of-concept implementation described by Pulls et al. that we performed and documented as part of the Research Project in Computer Science course at Karlstad University. The practical work of this thesis consists of (i) implementing the missing audit functionality for the proof of concept logging system, (ii) adapting an open-source issue tracker logging system and (iii) to evaluate the amount of work required to do so, in terms of modifying the software and in terms of performance of the resulting implementation.

This chapter summarises the scheme described by Pulls et al., the existing proof-of-concept implementation of the scheme and it ends with a description of the open-source issue-tracking software that we have modified as part of this project.

2.1 Distributed privacy-preserving secure log trails

As previously mentioned, our work builds on the logging scheme described in [10]. This logging scheme is designed to log information for the benefit of data subjects; in particular to increase the transparency of the process from which information is being logged. To increase the transparency of a process, the log can contain information about how data subjects’ personal information is being processed. Because the log can contain sensitive information and the act of retrieving log entries can in itself reveal further information, the scheme includes measures to protect the privacy of data subjects.

\(^1\)As defined in [10], data subjects are individuals whose data are being processed by data processors.  
2.1. DISTRIBUTED PRIVACY-PRESERVING SECURE LOG TRAILS

The scheme consists of three principal components: (i) the data processor, (ii) the log server and (iii) the data subject components. The data processor is the component that carries out a process, which can be viewed as a series of operations on a data subject’s information. Processes are often distributed, i.e. they may involve multiple data processors. The scheme facilitates such processes by providing a security- and privacy-aware solution for branching processes. Processes can branch out dynamically, depending on, for example, input data or current server load. The log server generates and stores log entries based on messages from one or more data processors. As part of generating log entries, the log server performs as much of the associated cryptographic work as possible without violating the security requirements of the scheme, to be described. The data subject component works on behalf of the data subject and is responsible for storing meta data about the data subject’s (active and finished) processes and for retrieving all available log entries pertaining to those processes upon request.

All log entries pertaining to a certain process can be organised into a log trail. A log trail is a set of log entries that can be spread out over several log servers and cover the actions of several data processors. This mirrors how a process can branch from one data processor to another. When a process branches a special type of entry is written to the log, called a cascade entry (described in Section 2.1.4). This log entry contains all the information the data subject needs to contact the log server that was chosen to generate log entries for the new branch. In this way the data subject can rebuild the full log trail without contacting any data processor, and without knowing the path of the process in advance.

2.1.1 Security requirements

In this section, we will restate an abridged version of the requirements for the logging scheme, given by Pulls et al. [10]. The requirements state a number of properties that the scheme should have, either permanently or until such a time that an entity in the
system becomes compromised. These properties concern aspects like the privacy of the data subject, the integrity of the log and the availability of the log for data subjects.

Functional requirements

For every process a data subject owns, the data subject should be able to retrieve all log entries generated as a side effect of that process. The ability to identify all the log entries belonging to a trail and decrypting those entries should be exclusive to the data subject.

Verifiable authenticity and integrity

When a log server is compromised all previously stored log entries should be safe from manipulation, i.e. it should not be possible for an attacker to drop entries or modify their payloads without being detected. To this end, a data subject should be able to verify the integrity of his log trail. Similarly, a data processor should be able to identify, retrieve and verify the integrity of all log entries it has submitted. Finally, an external auditor should, in collaboration with all data processors using a log server, be able to verify the integrity of the full log database at that log server.

Privacy

A common security requirement for a log is that it should be impossible (or computationally infeasible) for anyone but the intended recipient to read the contents of the log, i.e. it should be encrypted. This requirement protects information explicitly stored in the log. However, each log entry in the scheme also contains log meta data from which information could potentially be inferred. Therefore, this scheme takes additional steps to ensure the privacy of data subjects. It should not be possible for an attacker, even with full access to one or more log servers, to derive information about processes or parts of processes that took place before the compromisation of the log servers. Explicit requirements include
the unlinkability between log entries and data subjects, between any two log entries and between the different identifiers of a data subject.

**Auditability and accountability**

Because of the separation of log server and data processor, there can arise a need for accountability when the logging process is not working as intended. Therefore, it is required that both data processors and log servers are able to prove, towards entities in the logging scheme as well as trusted third parties, that they are working according to the specification. Section 2.1.5 describes how these requirements are fulfilled.

### 2.1.2 Usage

Before a data processor can perform logging for any process it must register with a log server, at which point the log server will initialise a state for the data processor. This state contains a secret that is shared with the data processor. After one or more messages have been submitted to the log server by the data processor, this secret can be used by the data processor to identify and retrieve the resulting log entries. The shared secret also plays a critical role in verifying the integrity of log entries.

Once a data processor has registered with a log server, processes can be started at that data processor. This can be done either directly by a data subject or by another processor on behalf of a data subject. Figure 2.1 shows a data subject \((S)\) a data processor \((P_1)\) and a log server \((L_1)\), and it illustrates the procedure of starting a new process. To complete the first step of the procedure, the data subject will generate a new key pair unique to that process. The public key, \(PuK_1\), is then included in the request sent to the first data processor. The purpose of the public key is two-fold: Aside from enabling public key cryptography to be used, it also serves as a unique identifier for the data subject at that data processor. Internally the data processor stores the public key in its data vault to be retrieved whenever further logging operations are performed.
When a data processor agrees to start a process for a data subject it must perform the second and third steps of the procedure. This is done by first contacting a log server to initialise a state for the given data subject identifier. After initialising the state the log server will return a secret, encrypted under the identifier. The data processor then enters a log entry indicating that it is committing to executing a specific process for that identifier. When the commitment message has been logged, the data processor performs the fourth and final step by returning a set of data referred to as the, $\text{InitData}$, containing among other things the encrypted secret from the log server and contact information for the log server to the initiator. An initiator can be a data subject or another data processor. In Figure 2.1 the initiator is a data subject. When the data subject receives the $\text{InitData}$ directly from the data processor it stores this information in its data vault. From the data vault the information can then be used to retrieve all log entries from that process and any branches thereof. In Figure 2.2 the same process is branched to another data processor ($P_2$) that is using another log server ($L_2$). In this situation, the first data processor, $P_1$, is the initiator of the process at $P_2$. When a process is moved from one data processor to another like this, a new public key is created through the process of cascading (described in Section 2.1.4). The new public key is then provided to the second processor in place.
of the original public key. The cascade value \( c \) is generated in the process of cascading and must be returned to the data subject in some way. In Figure 2.2, we can see the new public key, \( PuK_2 \), is transmitted to the second processor in step 1, and the cascade value \( c \) is stored in the log together with the initialisation data, \( InitData_2 \), in step 5.

Once the log server has successfully initialised a state for the data subject and the initialisation data has been transmitted to the initiator, the data processor can go on executing its process, logging information as needed.

### 2.1.3 Log entry meta data

In the log server’s database log entries consist of more than just the submitted message. Attached to each message there are two blocks of meta data; one block for the data subject and one for the data processor. Each block can in turn be divided into two fields; the *index chain* (IC) and the *data chain* (DC). The index chain allows the data subject or data processor to identify a log entry as part of their chain. The purpose of the data chain is to verify the integrity of a series of log entries. Because the data chain value is derived from the data chain of the previous entry in the series, as well as the index chain and message from the entry to which it is attached, it recursively depends on the index chains.
and messages of all preceding entries. For data subjects the series of log entries verified by
the data chain consists of all the log entries for the process to which it belongs. For data
processors it consists of all the log entries generated on that data processor’s behalf at that
log server. To maintain these chains the log servers keep a state for each data processor
and data subject it serves. This state consists of intermediate values rather than any IC
or DC value that has been used in a log entry, so that the state cannot be linked to any
existing log entry. When the message received from the data processor is combined with
the meta data blocks, this is how the log entry looks:

\[ \text{LogEntry} = (\text{IC}(S), \text{DC}(S), \text{IC}(P), \text{DC}(P), \text{Data}) \]

For details on log entry meta data and the initialisation, evolution and use of the state,
see the technical report by Pulls et al. [10].

2.1.4 Branching by cascading

As previously mentioned, a process can take place over multiple data processors. However,
if one unique identifier was passed from processor to processor, then an attacker could
potentially determine something about the data subject by examining when and where
the process branched. To address this threat, the scheme uses a method called cascading
where the identifier and public key of a data subject is combined with a \textit{cascade value} \( c \)
to create a new identifier that is a valid public key. The value \( c \) can then be stored in
the log, and by combining this value with the original private key the data subject can
derive the new private key corresponding to the new public key. The process of cascading
is described in greater detail in [10].
2.1.5 Auditing

The separation of the process from the duties of storing and providing the log to users is a feature of the system intended to minimise the overhead of logging on the data processor. This feature has the side effect that when an entity in the system is compromised and that entity’s behaviour is modified it is not clear to outside entities where the corruption has occurred. If there are missing log entries then the data processor may have neglected to ask for them to be stored or the log server may have received the log entries but only feigned storing them. To resolve situations like this, that can potentially involve multiple companies, e.g. in a situation where one company is outsourcing the log servers to the other, there is an audit scheme in place.

The audit scheme periodically triggers an exchange of signatures between the log server and data processor. These signatures can be used by the data processor to show towards a trusted third party (referred to as an auditor) that the log server has received and committed to store all log entries up to that point in time. Inversely the audit information can be used by the log server to show towards data processors, data subjects or external auditors that its log database has not been altered in any way.

The signature exchange

To trigger the auditing system all log servers maintain a round counter for each one of its associated data processors. The counter is incremented for each log message submitted by that processor and when the counter reaches a certain threshold a signature exchange is performed. The log server, \( L_\alpha \), first signs the final log entry that triggered the audit including all its meta data with its private key, \( PrK_{L_\alpha} \), and sends the signature, \( \text{Sig}_{PrK_{L_\alpha}} \), and the log entry, \( \text{LogEntry} \), to the data processor, \( P_\alpha \).

\[
L_\alpha \rightarrow P_\alpha : \text{Sig}_{PrK_{L_\alpha}} (\text{LogEntry})
\]
The data processor then verifies the log server’s signature using the log entry and the log
server’s public key before countersigning, i.e. signing the signature, with its private key,
PrK_{P_{\alpha}}, and sending this countersignature to the log server.

\[ P_{\alpha} \rightarrow L_{\alpha} : \text{Sig}_{\text{PrK}_{P_{\alpha}}}(\text{Sig}_{\text{PrK}_{L_{\alpha}}}(\text{LogEntry})) \]

When the log server has verified this countersignature it must contact a Time-Stamping
Authority (TSA)[2] and get a signed timestamp on the countersignature. The timestamp is
then shared with the data processor and both parties store the necessary data until such a
time that a new audit round is successfully completed. The timestamp binds the signatures
to a moment in time, assuring that the audit data cannot be replaced at a later time even if
both the involved data processor and log server are compromised and are colluding. After
storing the audit data the log server resets the round counter for that data processor.

For the log server the necessary data stored in the audit component is the last log
entry before the audit took place, the log servers own signature, the data processor’s
countersignature and the timestamp issued by the TSA. The data processor stores the
same data and one additional log entry: the second to last entry before the signature
exchange occurred.

Verification

The two log entries that the data processor stores with its audit information are designated
\( l_i \) and \( l_{i-1} \) and consist of the following parts:

\[ l_i = (\text{IC}(S)_i, \text{DC}(S)_i, \text{IC}(P_{\alpha})_i, \text{DC}(P_{\alpha})_i, Data_i) \]
\[ l_{i-1} = (\text{IC}(S)_{i-1}, \text{DC}(S)_{i-1}, \text{IC}(P_{\alpha})_{i-1}, \text{DC}(P_{\alpha})_{i-1}, Data_{i-1}) \]

The data processor also has the secret that was returned when the data processor registered
with the log server. Using parts of \( l_i \), \( l_{i-1} \) and the shared secret the data processor can
generate the expected data chain for $l_i$, $ExpDC(P_\alpha)_i$, as illustrated in Figure 2.3, and compare that value to the given $DC(P_\alpha)_i$ to verify it. The transformation applied in Figure 2.3 to generate $ExpDC(P_\alpha)_i$ is described in [10]. Verifying the data chain implicitly verifies the data chains of all previous log entries in the chain. Given that the log server has signed the log entry it should be able to reproduce valid log entries for all preceding index chain values. The log server in turn can show the data processor’s countersignature which proves that the data processor has seen the log entry and verified its data chain value, ensuring that all previous log entries are valid and that their generation was requested by the data processor.

2.2 Logging scheme implementation

For the work in this thesis we are using an existing proof-of-concept implementation of the logging scheme. This implementation is written in Java and in addition to the basic components of the logging scheme, the existing codebase also provides an example implementation.
of the interface used by the log server to provide its API. This example implementation encapsulates communication in HTTP and the class `LogServerRemoteWHardware`, seen in the class diagrams in Appendix E, is used on the client-side to transparently handle the network communication. With this addition the log server is autonomous and needs no alteration within the scope of this thesis. The rest of this section details how the logging components for the data subject and data processor are made available to applications.

### 2.2.1 Data subject

The data subject component provides the data subject application with all the unique functionality needed to make use of the logging system. This includes generating new asymmetric key pairs, interpreting and storing logging meta data given by the data processor upon starting a process and finally rebuilding a log trail based on that information.

#### The data vault

The data vault assists the data subject application in storing meta data about its processes. The `SubjectDataVault` class itself is a simple mapping from an integer key to a `SubjectDataVaultEntry` instance which contains all available information about a process. It uses the configuration file for persistent storage and data vault entries are loaded automatically upon instantiation of the `SubjectDataVault` class. One item in the data vault consists of:

- A public/private key pair.
- The URI and public key for the initial data processor.
- The URI and public key for the initial log server.
- The shared secret between the log server and data subject, called the Authentication Key.
When a data vault entry is created for a new process a key pair is automatically generated, which can then be used for the new process or be overwritten.

**Rebuilding a log trail**

The process of retrieving log entries is made available in the `LogTrail` class through the method `buildLogTrail()`. The `LogTrail` class uses the information available in the data vault to rebuild and verify the log trail for a specific process. The log trail is then available through the public member `Entries` in the `LogTrail` instance. `Entries` is a list type that contains plain text log entries and new lists of the same type where cascading has taken place.

### 2.2.2 Data processor

The data processor logging component assists data processors in logging-related matters such as (i) registering and deregistering data subjects with log servers, (ii) submitting messages to log servers, (iii) cascading and (iv) translating the internal identifiers of the data processor to the public key identifiers used in the logging system. The internal identifiers are mapped to their public key counterparts in the data vault. Using the data vault simplifies adoption of the logging system for existing applications that already have internal identifiers, and future applications that wish to use internal identifiers for the sake of simplicity or to abstract the use of the logging system. The data processor API is well defined in [10] apart from the interfaces of the data vault and cascading components.

**StartProcessLogging**

The API method `StartProcessLogging()` takes a public key as a parameter and will return a blob of data that should be forwarded to the data subject. Before it returns, the method will have contacted the log server to register the data subject so that logging for that data subject can take place. The data processor will also have submitted the
first message to the log server to indicate to the data subject that the data processor has committed to execute the requested process.

**LogData**

The `LogData()` API method takes as its parameters a data subject identifier and a message to be logged. The message is submitted to the log server and the log server responds with a log entry complete with meta data. The data processor’s meta data block is verified immediately. The log server may also return a signature on the log entry, which signals the data processor that it is time to perform the auditing procedure. If so, the auditing procedure is performed before the API call returns.

**EndProcessLogging**

The API method `EndProcessLogging()` deletes all information about a data subject at the data processor, such as what log server has been chosen for that data subject. The log server is also asked to close the log. This is important as it allows the log server to delete meta data it maintains about the process. Deleting meta data has some positive effects on the privacy of data subjects in case that the log server is compromised, and on the performance of the log server. The method takes a data subject identifier as a parameter and returns a single byte value which indicates whether the subordinate log server call has succeeded or failed.

**The data vault**

As previously mentioned the data vault maps the internal identifiers used by the data processor to ones used by the logging system. The data vault is abstracted by the `IDataVault` interface and the only available implementation is `SimpleDataVault`. The interface defines the method `addIdentifierPair()` which adds a pair of identifiers to the data vault and
the method `getGlobalIdentifier()` to retrieve a public key identifier given an internal identifier.

Cascading

When a process is to be branched, i.e. continued at another data processor, the identifier for that process must be cascaded, as described in Section 2.1.4. A method is provided that cascades an identifier and returns the new identifier together with the cascade value \( c \), but it is up to the data processor application to use this method and to store the resulting cascade value in the log, together with the data returned from the `StartProcessLogging` call at the other data processor.

2.3 Demonstrative scenario

The majority of this thesis deals with the application of the logging system to a real world usage scenario. This scenario is designed to show the capabilities of the logging system in a setting where the average user will find it useful. The chosen scenario begins with a customer contacting customer support regarding a faulty computer. The customer is then the data subject and the ticket opened in the customer support system plays the role of the process. This means that the issue-tracking platform where the ticket is first opened is the initial data processor.

This section presents the issue-tracking software, the configuration of data processors and log servers in the scenario and a step-by-step documentation of the user’s process from first contact to the successful resolution of the issue.

2.3.1 osTicket

When choosing an open source issue-tracking system to use in our scenario we took into consideration the size of the implementation, the programming language and whether it was
specialised in customer service applications or software development (e.g. bug tracking). Java implementations were considered first, to ease integration with the logging system, but most were out of date and ultimately there were no suitable candidates among them. In the end we chose osTicket\(^3\) for its minimal size and ease of use. It is written in PHP and uses a MySQL database for persistent storage. The version used in this work is 1.6.0, which is the most recent version at the time of writing (2012-05-06), and it is available at http://osticket.com/downloads.php. With osTicket, tickets are created from an online form or via email, either directly by customers or by staff members acting on behalf of customers. Tickets can then be assigned to departments and to specific staff members. Tickets have a priority level which is visible to staff members and they can have a deadline assigned if needed. Staff members can post updates to tickets that are visible to customers, or they can write notes visible only to the staff. The customer can also post updates to a ticket by first logging in using a ticket ID and the email associated with the ticket ID.

\[\text{Configuration}\]

In our scenario there are three osTicket instances, as illustrated in Figure 2.4. Two belong to the same theoretical company, one at a customer service front-end and the other at a service center. The third osTicket instance belongs to a company with expertise in data recovery that is consulted as part of our scenario. The osTicket instances are connected to local instances of the data processor’s Java component running as web services. Each data processor in turn makes use of a local log server. Each set of osTicket instance, data processor Java component and log server is maintained by a separate entity. The three entities in this scenario are named support, service and specialist in reference to their respective responsibilities.

\(^3\)http://www.osticket.com/, last accessed 2012-04-18.
2.3. DEMONSTRATIVE SCENARIO

2.3.3 Procedure

The scenario can be broken down to a series of ten steps beginning with the initial complaint and ending with the repaired computer being returned to the user. Once started the process will flow through the three entities as illustrated in Figure 2.5, and further described below:

S1 The customer opens a ticket regarding a faulty computer at the osTicket instance of the support entity by filling out an online form.

S2 The support determines that the customer’s faulty computer must be forwarded to the service center, i.e. the service entity. The customer is informed and responds by sending the computer.

S3 A new ticket is created by the support at the service’s osTicket instance.

S4 The support entity closes its local ticket.
Figure 2.5: The entities of the scenario with numbered arrows illustrating the chronological procedure for performing the scenario.
2.4. SUMMARY

S5 The service entity receives the computer and performs a number of tests, posting results to the ticket. It is determined that the hard drive has incurred physical damage that cannot be repaired.

S6 A company with expertise in data recovery is contacted, i.e. the specialist entity. A new ticket is created by the service entity at the specialist’s osTicket instance. The faulty hard drive is sent from the service entity to the specialist.

S7 At the specialist all data from the hard drive is successfully recovered and is stored in an image file.

S8 The image file is returned to the service entity where it can be installed to a new hard drive. With this the ticket is closed at the specialist.

S9 The image returned from the specialist is installed to a new hard drive which in turn is installed to the customer’s computer.

S10 The service entity returns the computer to the customer and closes the ticket. All tickets are now closed and the process is completed.

In Chapter 3 we discuss the implementation of the software components involved in this scenario and in Chapter 4 we show how the scenario can be performed using these components.

2.4 Summary

In this chapter, we have described a logging scheme aimed at increasing transparency for the benefit of data subjects as well as the existing proof-of-concept implementation of that scheme. We discussed in particular detail the auditing system of the logging scheme which was not included in the proof-of-concept implementation. Lastly, we described the issue tracking system, osTicket, that we have modified to utilise transparency logging.
for a scenario that suitably demonstrates the capabilities of the altered osTicket and the underlying logging system.
Chapter 3

Implementation

The following chapter details the practical work performed to realise the scenario laid out in Section 2.3. We also present our implementation of the audit functionality for the proof-of-concept logging system implementation. The implementation of the scenario includes the modification of the issue-tracking software osTicket (described in 2.3.1), and the implementation of a data subject application. The purpose of the data subject application is to assist the user by generating and storing important information like cryptographic keys, and to rebuild and visualise the log trail. Visualising the log trail allows the data subject to follow the progression of the scenario as it unfolds over multiple osTicket instances.

3.1 Audit components

The audit components are parts of the data processor and log server that execute the audit procedure described in Section 2.1.5. The purpose of the audit procedure is to assure that for each coupled data processor and log server mutual and provable assurances are exchanged periodically. The frequency of these exchanges is determined by the length of the audit round. An audit round consists of a set number of log entries and at the end of each round the audit mechanism is triggered. This mechanism can be described in four
steps:

1. An audit is triggered at the log server. A signature is generated and sent to the data processor.

2. The data processor receives the signature, verifies it and responds with a counter-signature.

3. The log server verifies the counter-signature, acquires a timestamp for it from a TSA and sends the timestamp to the data processor.

4. The data processor receives the timestamp and verifies it.

Next we will describe the finer points of how these steps are executed in the Java proof-of-concept implementation.

3.1.1 The log server

The log server audit component consists of a java.util.Map of data processor identifiers to a data structure, LAuditDataEntry, seen in the class diagrams in Appendix E, that keeps the most recent audit data stored as well as a round counter and another data object that can hold an extra signature. When a new data processor is registered a new LAuditDataEntry object is created and stored in the Map. Each time a new log entry is generated the audit component is contacted so that it can increase the round counter for the relevant data processor and, if the round counter reaches the threshold, trigger the audit mechanism.

When the audit mechanism is triggered the log server will in Step 1 store the signature generated and the last log entry of the audit round in the LAuditDataEntry without overwriting any data of the most recent successful audit. When the counter-signature is then received from the data processor in Step 3 it can be verified against the original signature. The required timestamp is generated by the custom class TSA and its static
method getTimestamp(). When the timestamp has been generated and sent to the data processor the log server overwrites the last audit data for that data processor with the new signature, counter-signature, timestamp and log entry.

3.1.2 The data processor

The data processor can make use of multiple log servers, which means that the data processor audit component contains a map similar to the one in the log server audit component. The map uses a log server’s public key as the identifier and the value is an instance of the class PAuditDataEntry. The data processor audit component, like its log server equivalent, is updated each time a log entry is successfully committed to the log. The data processor needs to store last two log entries of an audit round to be able to verify its audit information. Each log server that is used by a data processor has an array of two log entries in its PAuditDataEntry that is updated for each new log entry generated by that log server. After completion of an audit round the two entries in that array are copied along with the rest of the audit data to another array where it is stored until it is overwritten after the next audit round is completed.

In Step 2 of the audit mechanism the data processor receives a signature along with the final log entry of an audit round. The data processor reacts by verifying the signature using the log entry and the log server’s public key which must be given in the configuration of the data processor. The data processor then signs the signature using its private key which is the same for all log servers it uses. When the timestamp is received from the log server in Step 4 it is verified with the TSA using the validateTimestamp() method of the class TSA before the new audit data overwrites that of the last successful audit.

3.1.3 TimeStamping Authority

The functionality of the TSA is invoked through two static methods of the class TSA: getTimestamp() and validateTimestamp(). In this proof of concept these methods per-
form the duties of a TSA locally at each entity using a shared key pair. In the future the current implementation should be replaced by calls to an external TSA.

The method `getTimestamp()` takes an array of bytes as its only parameter and returns a signed timestamp. The timestamp is a string representation of the local date and time. After a timestamp is generated it is copied to a new byte array. The data provided as a parameter is then hashed and placed in the new byte array following the timestamp. The result is signed and then stored in an instance of a custom `Timestamp` class together with the original timestamp data.

The purpose of the `validateTimestamp()` method is to validate that a given timestamp signature pertains to the given data and that the timestamp has not been altered. It takes a `Timestamp` and an array of bytes, `data`, as parameters, and arranges the timestamp data from the `Timestamp` object with the `data` in the same way as `getTimestamp()` before verifying the signature. The public key of the TSA is assumed to be public.

### 3.2 The data processor

The data processor consists of the open source software issue-tracking software osTicket, as described in Section 2.3.1. Our goal was to add the following features to osTicket:

- A new logging process should be created for each new ticket.
- Information should be written to the log for significant events, starting with messages written to the ticket by staff and by the user.
- When a ticket is closed, the associated process should be closed as well.
- osTicket should facilitate the branching of a process upon request from authorised staff.

From these requirements we can infer a number of practical goals:
3.2. THE DATA PROCESSOR

G1 The osTicket software must be interconnected with the Java data processor API.

G2 When creating a ticket in the system, a data subject must provide a unique public key for the logging process.

G3 After a ticket has been created the initialisation data must be provided to the data subject.

G4 To assist in branching, osTicket should provide cascaded identifiers upon request from authorised personnel.

G5 osTicket should provide an interface where personnel provide the necessary raw data for branching and it should format and commit that data in the form of a branching message in the log.

The rest of this section describes how these goals were accomplished.

3.2.1 Interconnection

To accomplish G1 we decided to use a SOAP[6] interface over HTTP to connect osTicket, which is written in PHP, with the data processor API, written in Java. We chose SOAP because it is simple and it has support in Java EE via JAX-WS¹ (Java API for XML Web Services) as well as native support in PHP. The interface is similar to the data processor API but has been altered to minimise the changes required in osTicket.

To use JAX-WS a normal Java class is annotated² to specify what parts should be offered as a web service and how. We have used JAX-WS to annotate a class named SOAPProcessor. A typical method definition in SOAPProcessor is annotated like the LogData() method, shown in Listing 3.1.

We can provide the annotated Java class as a web service using a javax.xml.ws.Endpoint shown in Listing 3.2. When providing a web service like this JAX-WS will automatically

---
²Java annotations are a form of meta data that can be included with Java source code.
generate a description of the SOAP interface in the Web Services Definition Language (WSDL). The WSDL definition is then made available via HTTP at "url?wsdl", where url is the value given as a parameter to the Endpoint.publish() method in Listing 3.2. Using the WSDL definition, a client, like PHP’s native SoapClient class, can use the service with minimal configuration. The WSDL for this interface can be found in Appendix B.

### 3.2.2 osTicket

In osTicket the procedures for creating, deleting and posting messages to tickets have been altered to make use of the logging system and taking into account the response values therefrom. The process of creating a ticket, for example, has been altered to include a call to the method StartProcessLogging() on the data processor’s Java component described in Section 3.2.3, and if that call returns an error then the ticket is not created and an error message is displayed to the user.
3.2. THE DATA PROCESSOR

Figure 3.1: The online form for creating tickets, modified to accept a public key

Interface changes

Without modification osTicket provides multiple ways for a ticket to be created; via email, an online form filled out by the customer or a separate online form filled out by staff. When implementing G2 we removed support for creating tickets via e-mail to save time as the online forms offered a quick and intuitive solution. The online forms were altered to incorporate an `<input>` field in the HTML form where the public key can be input in its hexadecimal representation. The HTML form with the additional field is shown in Figure 3.1. When the form is submitted it is processed by `open.php`, shown partially in Listing 3.3, where a logging process is started and a log message containing the message given by the data subject is submitted.
// Ticket::create... checks for errors..
if (($ticket=Ticket::create($_POST, $errors, SOURCE))) {
    $msg='Support ticket request created';

    // LOGGING: Initialise data subject with processor
    $initData = Processor::startProcessLogging($ticket->getExtId(),
        $POST['puk']);

    $result = Processor::logMessage($ticket->getExtId(),
        date("Ymd H:i:s") .
        " - Your ticket was opened with the following " .
        "information provided: <b>" .
        $POST['message'] . "</b>";

    if ($result != "13") {
        error_log("LogData failed for process " . $ticket->getExtId()
            . " with status " . $result);
    }

    if ($thisclient && $thisclient->isValid())
        @header('Location: tickets.php?id=' . $ticket->getExtId());

    // Thank the user and promise speedy resolution!
    $inc='thankyou.inc.php';
} else {
    $errors['err'] = $errors['err'] ? $errors['err'] : 'Unable to create a ticket. Please correct errors below and try again!';
}

Listing 3.3: Excerpt from open.php showing the added code to open a log trail
Figure 3.2: The page shown to the user after a ticket has been successfully created
Figure 3.3: A part of the ticket information page showing the controls available to staff members

After a new ticket has been created the initialisation data is returned to the data subject, as required by G3, on the standard response page. The initialisation data is returned from the data processor’s Java component in its hexadecimal representation and is presented to the data subject by putting the value of the variable $initData (from Listing 3.3) in a $textarea which is marked read-only, shown in Figure 3.2.

When a ticket has been created staff members can view the ticket and make changes to it. On the page vieviewticket.php, shown partially in Figure 3.3, information about a ticket is displayed together with small forms for performing common actions such as posting a public reply or a private note for staff only. There is also a drop-down list for tasks such as deleting, closing or setting the priority of a ticket. This is where we have added the option to branch the process associated with that ticket, fulfilling G4 and G5 in one page. Choosing this option brings the staff member to an entirely new page named cascade.inc.php, listed in Appendix D and pictured in Figure 3.4. On this page a cascaded public key is provided in a read only field, and the cascading value is in a hidden
3.2. THE DATA PROCESSOR

Figure 3.4: The process branching page that is available to staff members

field so that it is available to be logged when the form is submitted by the user. There is also an empty `<textarea>` for the initialisation data returned by the new data processor. Upon submitting the form a cascade entry is written to the log for the original process.

**Behind the scenes**

Aside from the changes that have a visual representation we have also slightly altered the process of posting messages to tickets. When posting a message a call is made to the data processor API to first write the message to the log. If an error occurs during this API call an error message is shown to the user, but the message will still go through. Similarly, when a ticket is deleted an API call closes the process belonging to that ticket, printing potential error messages to the server log file with the native PHP method `error_log()`.

To communicate with the data processor’s Java component we have created the PHP class `Processor` which provides a set of static methods. The static methods typically use PHP’s native SOAP support to call a method from the data processor’s Java component and return the results from that, or an error value. A simple case that works this way is
The method `startProcessLogging()` shown in Listing 3.4.

### 3.2.3 Data processor’s Java component

The `SOAPProcessor` Java class offers a set of methods similar to the data processor API as described in Section 2.2.2. In fact, for the methods `StartProcessLogging()`, `LogData()` and `EndProcessLogging()` this Java class provides close matches with a few changes to the parameters. The most important difference between this helper class and the data processor API is that the helper class uses a data vault internally. This means `StartProcessLogging()` takes an additional integer parameter, as the identifiers used internally by osTicket are integers, and subsequent calls to `LogData()` and `EndProcessLogging()` need only provide that same internal identifier. These methods return string values which are hexadecimal representations of the values returned by their data processor API counterparts.

Three methods are provided for the purpose of branching processes. They are `GetCValue()`, `CascadeKey()` and `WriteCascadeEntry()`. `GetCValue()` will generate an appropriate value for cascading. A call to this method is usually followed by a call to `CascadeKey()` which takes as its parameters an internal identifier and a cascade value, and returns the a
cascaded version of the public key related to that internal process identifier. The method WriteCascadeEntry() is called by osTicket after the branching form has been submitted by a staff member. This method takes a cascading value, an internal process identifier and a block of initialisation data for its parameters and arranges this information in a format that is understandable to the data subject before writing it to the log.

3.3 The data subject

In the scenario we have presented the role of the data subject is to (i) create the initial ticket at the support entity’s osTicket instance, (ii) to respond to any questions asked by staff in relation to that ticket and (iii) when asked to do so, submit the faulty laptop to the service entity. In addition to this the data subject needs some way to rebuild and view his log trail. We have created an interface where the data subject’s log trail is visualised as it is branched to the support, service and specialist entities. This visualisation consists of abstract representations of the entities and the resources involved (the faulty laptop, etc.) and a set of textual log entries at each entity. The purpose of the visualisation is to give the data subject a better understanding of the current status of the ticket as well as what information has been shared and with whom.

When modifying osTicket to make use of the logging system an input field was added to the online form for creating tickets. In this field the data subject must input an appropriate public key in hexadecimal notation. Additionally, when a ticket has been created initialisation data is returned to the user that must be forwarded to the data subject application in some way. These additional requirements make the process of creating a ticket more cumbersome. Therefore we have chosen to integrate the data subject application with the Google Chrome web browser\(^3\) to automate these extra steps. This is accomplished with an extension that automatically inputs a new public key where it is required and auto-

\(^3\)https://www.google.com/chrome, last accessed 2012-05-01.
matically stores the initialisation data in the data vault. The log trail visualisation is also packaged with this extension and can be viewed directly in Chrome.

The data subject’s Chrome extension consists of a set of JavaScript files that can access and manipulate the Document Object Model (DOM)\(^4\) of the relevant osTicket pages. The extension also contains JavaScript, CSS and HTML files for the log trail visualisation page. Since much of the required functionality of the data subject’s application, such as log trail rebuilding and the data vault, was already implemented in Java we decided, for the sake of scope, to make use of a Java component. This Java component needs to be run separately from the extension. It offers an HTTP interface that can be accessed from the extension’s JavaScript code and returns objects encoded in JavaScript Object Notation (JSON)[3].

### 3.3.1 Interconnection

The HTTP interface offered by the Java component is made up of three methods: `GetPuk()`, `GetTrail()` and `PostInit()`. These methods are identified in the HTTP GET request by their URL paths; “/getpuk”, “/gettrail” and “/postinit”.

`GetPuk()` generates a new public key and returns it in hexadecimal notation as an element in a JSON object. There is also a boolean element named `status` in the returned object that indicates if the call was completed successfully. An example of a `GetPuk()` result is shown in Listing 3.5.

The method `PostInit()` requires initialisation data to be passed as a parameter in the query string of the requested URL. The method deconstructs the initialisation data and stores the resulting data in the data vault together with the key generated in the last call.

\(^4\)http://www.w3.org/DOM/, last accessed 2012-05-01.

```
{
    "status":true,
    "puk":"36732d8e537e4674..."
}
```

Listing 3.5: An example of a JSON object returned by `GetPuk`
3.3. THE DATA SUBJECT

Listing 3.6: A typical method from the PHP class Processor

```json
{
  "dsURI": "192.168.56.101:11133",
  "lsURI": "http://192.168.56.101:11132/",
  "branch": true,
  "entries": [
    {
      "text": "Process starting...",
      "branch": false
    },
    ...
  ]
}
```

to GetPuk().

The visualisation component of the Chrome extension uses the GetTrail() method to retrieve the log trail of a process. It is returned as a JSON object with an array of log entries, where cascade entries contain their own array of log entries. A brief example of the format is shown in Listing 3.6.

### 3.3.2 Chrome extension

The bulk of the data subject’s Chrome extension is written in JavaScript and uses the jQuery\(^5\) JavaScript library. The responsibility of automatically injecting a public key in the ticket creation form and retrieving the initialisation data after the form has been submitted lies with the file open.js, included in full in Appendix C. Using jQuery.ajax() an HTTP GET request is sent to the Java component and after a successful exchange the JSON data is automatically parsed into a JavaScript object. Given this connection it is a simple matter to fill the input field or retrieve and send the initialisation data as evidenced by the mere 34 lines of code in open.js. This part of the Chrome extension is registered in the extension manifest, shown in Listing 3.7, as a content script. This means that open.js is executed in the context of a page with access to its DOM although it cannot communicate with other scripts running on the page.

Listing 3.7: Part of the Chrome extension manifest where open.js is registered as a content script

```json
{
  "background": {
    "scripts": ["background.js"]
  },
  "browser_action": {
    "default_icon": "icon.png"
  },
  "content_scripts": [
    {
      "matches": ["http://*/osticket/open.php", "http://*/osticket/open.php?"],
      "js": ["jquery-1.7.2.min.js", "open.js"],
      "run_at": "document_idle"
    }
  ],
  "permissions": ["*", "tabs"]
}
```

Figure 3.5: The button added to Google Chrome’s toolbar by the data subject extension
The extension also has an associated browser action which means that when it is loaded it adds a button to Google Chrome’s toolbar next to the address field, pictured in Figure 3.5. The script `background.js` is run automatically when Chrome loads the extension and when run it registers a callback method that will be called when the toolbar button is pressed. This callback method will cause another file included with the extension, `logdisplay.html`, to open in a new tab. `logdisplay.html` is where the log trail is retrieved from the Java component and displayed to the user. The included JavaScript file `logdisplay.js` includes both jQuery and the jsPlumb\(^6\) JavaScript library which is used to “plumb” together visual elements, giving the page the appearance of a dynamic flowchart. On the `logdisplay.html` page there are a number of hidden images that are made visible when appropriate, based on the content of the log. If the process has been started at all then the image representing the support entity will be made visible and after that branching the process will reveal the service and specialist entities. Green connections will appear to illustrate the initial connection from the data subject to the support entity as well as process branches between the entities. Connections of other colours represent a transmission of a physical item or data, i.e. the laptop, hard drive or the recovered hard drive image. In Figure 3.6 we see `logdisplay.html` after the laptop has arrived at the service entity.

When the mouse cursor hovers over either the support, service or specialist entity on `logdisplay.html` a list of log entries called the graphic log for that entity will appear next to the cursor. Double-clicking the entity will cause the graphic log to be affixed to the bottom of the entity until it is double-clicked again. Normal ticket update entries from osTicket begin with a timestamp and ends with the update message in bold. Staff at the logging entities can also enter meta-entries that denote events that concern the ticket, such as the laptop arriving at the service entity. This is done by submitting a ticket update that begins with a hash and is one of (i) #computer-arrived, (ii) #harddrive-arrived, (iii)

\(^6\)http://jsplumb.org/, last accessed 2012-05-01.
Figure 3.6: The log trail visualisation page, logdisplay.html
3.3. THE DATA SUBJECT

Figure 3.7: The log trail visualisation page, logdisplay.html displaying graphic logs

#image-built, (iv) #image-received, (v) #image-installed and (vi) #sent-laptop. Values (ii) and (iii) are to be entered from the specialist entity and the rest from the service entity. All of these values when encountered in logdisplay.js trigger a visual change on the page and a special entry is written in the graphic log with a background colour that matches any connection triggered by that entry. Figure 3.7 is the same as Figure 3.6 but with the graphic logs displayed.

3.3.3 Java component

The data subject Java component uses the Jetty\(^7\) library to provide an HTTP interface to the Chrome extension. It provides three methods; GetPuk(), InitPost() and GetTrail(). The first two are used by the Chrome extension to help with creating new tickets. When the user opens the ticket creation page of the modified osTicket the Java component is

\(^7\)http://jetty.codehaus.org/jetty/, last accessed 2012-05-02.
contacted and `GetPuk()` is run. `GetPuk()` uses an existing method in the logging system Java implementation to generate an appropriate key pair. The pair is saved in a data object marked `public`, and the public key is saved to a string in hexadecimal notation which is then returned. When `PostInit()` is then called it will read the key pair from the `public` data object and use it while decoding the initialisation data that is passed as a parameter. When the initialisation data has been deconstructed and decrypted it is saved in the data vault, which is part of the pre-existent data subject Java API, together with the key pair. Finally, an integer identifier representing the position of the data in the data vault is returned.

The `GetTrail()` method takes a data vault index as a parameter and will, based on the information found in the data vault, rebuild a log trail and arrange it in a JSON structure before returning it. The data subject API offers functionality for rebuilding log trails from data vault entries which greatly simplifies the implementation of `GetTrail()`. A data vault entry is read from the data vault and passed to the log trail rebuilding component after which the log trail is reorganised by `GetTrail()` into a data structure that can be translated to JSON by `org.eclipse.jetty.util.ajax.JSON.toString()`. In this structure textual entries and branching entries are represented by generic instances of `java.util.Map` which can be translated to JSON objects. In each `Map` information is stored in `String-Object` pairs where the `String` is the identifier and the `Object` can be a `String`, a `boolean`, a `Map` or an array of `Objects`. Each of these Java types has a JSON equivalent to which they are translated by `JSON.toString()`. This restructuring is performed by the `jsonify()`, method shown in Listing 3.8.

### 3.4 Summary

In this chapter, we described how the audit components laid out in Chapter 2 were implemented in the existing proof-of-concept logging system implementation. We also described
```java
public String jsonify(LogTrail trail) {
    if (trail.Entries.size() == 0) return null;
    return JSON.toString(jsonifyProcessEntry(trail));
}

public Map jsonifyProcessEntry(LogTrail trail) {
    Map jsonMap = new HashMap();
    jsonMap.put("branch", true);
    jsonMap.put("dsURI", trail.getProcessor().toString());
    jsonMap.put("lsURI", trail.getLogServer().toString());
    jsonMap.put("list", jsonifyList(trail.Entries));
    return jsonMap;
}

private Map jsonifyTextEntry(TextEntry entry) {
    Map jsonMap = new HashMap();
    jsonMap.put("branch", false);
    jsonMap.put("text", entry.toString());
    return jsonMap;
}

public Map[] jsonifyList(LinkedList<ILogEntry> entries) {
    Map[] processedEntries = new Map[entries.size()];
    Iterator<ILogEntry> iterator = entries.iterator();
    for (int i = 0; iterator.hasNext(); i++) {
        ILogEntry entry = iterator.next();
        if (entry.getType() == ILogEntry.TEXT_ENTRY) {
            processedEntries[i] = jsonifyTextEntry((TextEntry) entry);
        } else if (entry.getType() == ILogEntry.TRAIL) {
            processedEntries[i] = jsonifyProcessEntry((LogTrail) entry);
        }
    }
    return processedEntries;
}
```

Listing 3.8: Methods used to translate a log trail from Java objects to a JSON string
how the issue tracking software osTicket has been augmented to use the logging system and provide tools for process branching between osTicket instances. Lastly we presented and described the data subject application that maintains a data vault and rebuilds and displays a data subject’s log trails. Using the data processor and data subject software described above we can execute and visualise, from the data subject’s end, the scenario laid out in Section 2.3.
Chapter 4

Results

This chapter demonstrates how the scenario laid out in Section 2.3 is performed from the point of view of the imaginary customer and staff members. When performing the scenario we will refer to the steps S1–S10 from Section 2.3.3 as we describe how they are executed. Furthermore, we summarise the implementation work by evaluating its performance impact on osTicket and identifying areas where the logging system can be improved.

4.1 Setting up the scenario

As described in Section 2.3, the scenario requires three entities, each with separate instances of osTicket, log server and the data processor’s Java component. We refer to these three entities as the support, service and specialist entities. The data subject needs to have access to the support’s osTicket instance where it can create the initial ticket. In order to rebuild the log trail the data subject needs connectivity to all three log servers. The data subject must use Google Chrome to utilise the data subject extension which allows the user to visualise the log trail.
CHAPTER 4. RESULTS

4.2 Performing the scenario

With all entities online the scenario is started by the customer at the osTicket instance of the support entity. Following the steps from 2.3.3 the scenario is performed as follows:

S1 The customer fills out the online form, as shown in Figure 4.1, with the help of the extension. When the form is submitted the extension will store the returned initialisation data in its data vault and provide the user with a unique identifier for the entry in the data vault, as shown in Figure 4.2.

S2 When the ticket has been created staff members will be able to see it via the staff panel by navigating to the open tickets page, as shown in Figure 4.3 on page 50. Clicking the ticket subject will bring the staff member to a page where the ticket can be replied to. The staff member enters a message informing the customer that the laptop must be submitted to the company’s service center, as shown in Figure 4.4 on page 50. In Figure
4.2. PERFORMING THE SCENARIO

Figure 4.2: The ticket creation response page when using the data subject extension

4.5 on page 51 we see what the log visualisation looks like for the customer at this point.

S3  The staff member at the support entity responsible for the ticket chooses to branch the underlying logging process and is brought to the branching page, shown in Figure 4.6 on page 51. Using the given public key the staff member fills out the ticket creation page at the service’s osTicket instance, shown in Figure 4.7 on page 52. The returned initialisation data is input in the branching form, as shown in Figure 4.8 on page 52, and upon submitting the branching form the service entity will appear on the customer’s log visualisation page, shown in Figure 4.9 on page 53.

S4  Figure 4.10 on page 54 shows one of several ways to delete a ticket. When the ticket is deleted the underlying logging process is closed at that entity. This does not have a visual representation for the customer on the log visualisation page.
Figure 4.3: The staff member’s view of open tickets

Figure 4.4: Responding to the customer’s complaint
4.2. PERFORMING THE SCENARIO

Figure 4.5: The log visualisation page after the first contact with support

Figure 4.6: The staff member’s cascade page
Figure 4.7: The service entity’s ticket creation page filled in by support staff

Figure 4.8: The staff member’s cascade page with initialisation data entered
Figure 4.9: The log visualisation page after the logging process has been branched
CHAPTER 4. RESULTS

Figure 4.10: Deleting the customer’s ticket at the support entity

S5 When the faulty laptop arrives at the service entity the first meta entry is written to the log by a staff member. As can be seen in Figure 4.11, this is no different from submitting a normal entry to a ticket. In Figure 4.12 on page 56 we see the customer’s log visualisation page after the laptop has arrived.

S6 When the laptop has been diagnosed the service staff branch the logging process to the specialist entity. The hard drive is then taken from the laptop and sent to the specialist. When the hard drive arrives the staff at the specialist entity will submit the second meta entry, at which point the customer’s log visualisation page will look like Figure 4.13 on page 56.

S7 At the specialist all data is recovered successfully from the faulty hard drive. A staff member at the specialist entity then submits the third meta entry to the ticket. This meta entry indicates that a full disk image has been created from the hard drive. A normal ticket update is also submitted with any additional details. Figure 4.14 on page 57 shows how the customer’s log visualisation page will look at this point.
Figure 4.11: The first meta entered as a normal ticket update
Figure 4.12: The log visualisation page after the first meta entry has been submitted

Figure 4.13: The log visualisation page after the specialist has received the hard drive
4.2. PERFORMING THE SCENARIO

A new meta entry is submitted at the service entity when the image file has been received from the specialist and work can continue on the laptop. In Figure 4.15 we can see what the customer’s log visualisation page looks like after this.

Another meta entry is written, this time indicating that the image has successfully been installed to the laptop and that it is now in working condition. This is represented in the log visualisation page by replacing the image and broken laptop symbols at the service entity with a working laptop symbol, as shown in Figure 4.16.

In the last step the last meta entry is submitted by the staff at the service entity and as the last ticket is deleted the last branch of the process is closed. The final layout of the log visualisation page without the graphic logs closed is shown in Figure 4.17 on page 59.
Figure 4.15: The log visualisation page after the disk image has been sent to the service entity

Figure 4.16: The log visualisation page after a new hard drive has been installed
4.3 Scenario coding summary

To realise the scenario we laid out in Section 2.3 we have altered or implemented the following components from scratch:

- osTicket.
- The data processor’s Java component.
- A browser extension.
- The data subject’s Java component.

Table 4.1 shows how many lines of code were written for each of these components, as reported by the utility clc\textsuperscript{1}. For the browser extension the number of lines is the combined total of the HTML, CSS and JavaScript elements. Of the 182 lines added to the osTicket

\textsuperscript{1}http://cloc.sourceforge.net/, last accessed 2012-05-21.
60 CHAPTER 4. RESULTS

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>osTicket</td>
<td>182</td>
</tr>
<tr>
<td>Data processor’s Java component</td>
<td>139</td>
</tr>
<tr>
<td>Browser extension</td>
<td>366</td>
</tr>
<tr>
<td>Data subject’s Java component</td>
<td>266</td>
</tr>
</tbody>
</table>

Table 4.1: The number of lines added to each component

project, 69 belongs to the Processor class in class.processor.php, shown in Appendix D.2. The Processor class connects osTicket to the data processor’s Java component.

4.4 Logging system suggestions

As we have worked on integrating the proof of concept logging scheme implementation with osTicket we have noted a few places where adapting has been unnecessarily problematic. In this section we suggest some minor changes to the logging system that could simplify future integration work.

4.4.1 Data vault integration

When implementing the the data processor’s Java component we chose to integrate the data vault functionality with the StartProcessLogging(), LogData() and EndProcessLogging() API calls. We did this by adding a parameter to StartProcessLogging() that specified the internal identifier that osTicket was using for the process and then supplying this identifier to subsequent LogData() and EndProcessLogging() calls. We believe this is a simpler interface than the current API, and that this interface would be preferred in most, if not all, applications.

In general terms the internal identifier is (from the logging scheme’s point of view) an unspecified data object. This has the effect that the data vault has to be implemented in a generic way so that it can handle any type of identifier, or there needs to exist a number of data vault implementations for the most common identifier types. This issue may be
the reason the data vault is treated as a separate component. We, however, believe that a
generic data vault would be possible to implement in any reasonably modern programming
language, e.g. C using void- and function pointers, and therefore it would not be a problem
to integrate the generic data vault with the data processor’s API. Doing so would also affect
the implementation of the cascade component as it also needs access to the logging system
identifiers for data subjects.

4.4.2 Cascade entry generation

The cascade entry is the special type of log entry that is committed to the log when branch-
ing takes place in a process. It contains a cascade value and a block of initialisation data
from the new data processor. The cascade entry is committed by the data processor and is
then read by the data subject where it must be recognised as a special entry and disassem-
bled accordingly. The implementation of the data processor’s cascading component offers
methods for cascading data subject identifiers but does not provide a specific method for
generating and committing a cascading entry. Instead cascading entries can be manually
formatted and committed via a normal LogData() API call. This means that the software
that wishes to make use of the data processor API must synchronise the cascading entry
format with the data subject API. If the data processor’s cascading component included
a method for this it would simplify the integration process. The method would accept
a cascade value and a block of initialisation data as its parameters, format and commit
a cascading message using the LogData() API call and finally return the value that was
returned from said API call.

4.5 Ticket deletion

In osTicket, tickets can be closed and they can be deleted. Normally, when an issue is
resolved in a system like osTicket, the ticket is closed, but not deleted. This causes a
problem for the logging system because closed tickets can be reopened; closed logging processes, however, cannot. Therefore, the logging process associated with each ticket is closed first when the ticket is deleted. We wanted to reflect this in the scenario to show the life of the logging processes from beginning to end.

When a logging process is closed some meta information about the process is deleted in the data processor API and at the log server. It is necessary to close logging processes, because doing so reduces the resource usage of both the data processor and log server. It also assures that if either entity is compromised by an attacker no new log entries can be generated for the closed processes.

In the rest of this section we suggest some ways this problem can be dealt with in systems like osTicket.

4.5.1 Policy

A company using this version of osTicket could employ a policy where tickets are closed when they are resolved and remain closed for a set amount of time. This period should be long enough that it is unlikely that the ticket would be reopened after its expiration, therefore the ticket can be deleted. This is also a form of data minimisation.

4.5.2 Changes to osTicket

A new ticket state could be introduced into osTicket where the ticket is closed and cannot be reopened but is also not deleted. This way there is no information loss, but processes can still be closed in the logging system. An alternate approach would be to allow customers to reopen tickets by providing a new logging system identifier that can be used by the data processor to start a new logging process. Such a solution would necessitate some changes to the data subject application as well.
4.5.3 Changes to the logging system

Though the logging scheme does not currently provide a way for processes to be reopened after they are closed, it may not be impossible. When a logging process is closed, the log server deletes the meta information it maintains about that process, without which it cannot generate new entries. However, the data subject can regenerate that meta data from the information in its data vault. This means that in this logging scheme there may be a way for a data subject to trigger a process to be reopened. The trouble with this approach is that even if the mechanism can be worked out it is likely to add complexity to the logging scheme; complexity that may eclipse the positive effects of the feature it is providing. In most cases, even where reopening functionality is wanted, it may be easier to apply the aforementioned technique of creating a new logging process. The data subject application would then assemble and present multiple log trails as a whole.

4.6 Performance

As part of our project evaluation we have performed some rudimentary performance tests on our implementation. We do this to provide a basis for discussions on future work and the feasibility of the logging system for potential applications. The numbers presented in this section are approximate and are based on a proof of concept implementation for which there may exist opportunities for significant optimisations. The performance measurements were taken on a Virtual Machine (VM) using the free software Oracle VM VirtualBox\(^2\). The VM was configured to use 512 MiB of primary memory and was running Arch Linux\(^3\) with the Xfce desktop environment. The following software was used in the tests: Apache HTTP Server\(^4\) - 2.2.22, PHP 5.3.10 and Java 1.7.0.03-icedtea. The host machine runs Windows

\(^3\)http://www.archlinux.org/, last accessed 2012-05-26. The kernel, as given by `uname -rm`: 3.2.9-1-ARCH i686.
7 on an Intel i5 2500K processor running at 3.3 GHz, of which the virtual machine is configured to utilise up to one hundred percent of one core.

4.6.1 Methodology

There are three primary functions in osTicket that have been altered to incorporate the logging system: (i) opening a ticket, (ii) replying to a ticket and (iii) deleting a ticket. We have timed each of these procedures in both an unaltered instance of osTicket, and in an instance of osTicket that used our logging system, to measure the performance impact of the logging system.

To time the PHP scripts that execute the procedures identified above we have used the PHP method `microtime()`. Given the value `true` for its first parameter, `microtime()` returns the number of seconds since the Unix epoch\(^5\) in a float format. According to [http://www.php.net/](http://www.php.net/) this method is accurate to within one microsecond\(^6\). To time each PHP script we used the `microtime()` method at the beginning and end of the execution of the script. The time delta was then written to a file.

We executed PHP scripts through a normal HTTP request to a local Apache HTTP Server, where the scripts are interpreted by the module `mod_php`. An alternative approach for executing the scripts in a test environment would be to run them through a PHP interpreter directly, without the need for HTTP communication or the involvement of a webserver.

For a more in depth look at where optimisations are possible, we decided to time the data processor’s Java component as it executes its part of the open, reply and deletion procedures. In Java we make use of `java.lang.System.nanoTime()` to measure the execution time of the methods `StartProcessLogging()`, `LogData()` and `EndProcessLogging()`. The measurements of the Java components were performed separately from the osTicket

---

\(^5\)00:00:00 UTC on 1 January 1970.

measurements so that the timing overhead from the Java components would not influence
the results of the osTicket measurements.

Having measured both osTicket and the Java component, we can now infer a third
measurement by subtracting the time required by the Java component and normal osTicket
execution time from the total execution time of the modified osTicket. This measurement
roughly represents the additional time required to contact the Java component from the
PHP code in our system. It also includes some error handling code that should not be
considered as overhead. We assume that the execution time of the error handling code is
comparatively minor to the overhead from using the Java web service. In the results this
measurement will be referred to as overhead.

4.6.2 Performance results

In Table 4.2, we present the results of the measurements on osTicket, both in its unaltered
form and with the logging system integration. The average values stated are the arithmetic
mean of one hundred iterations of each test. We also give the standard deviation, \( \sigma \), of the
test results.

<table>
<thead>
<tr>
<th></th>
<th>Open</th>
<th>Reply</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>without logging</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>25.8</td>
<td>65.0</td>
<td>43.0</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>4.5</td>
<td>6.5</td>
<td>4.2</td>
</tr>
<tr>
<td>with logging</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>262.5</td>
<td>142.8</td>
<td>63.1</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>28.9</td>
<td>10.0</td>
<td>8.2</td>
</tr>
</tbody>
</table>

Table 4.2: The PHP page generation time for creating, replying to and deleting tickets in
osTicket, given in milliseconds

In Table 4.3, we present the same values for the Java measurements, once again using
a sample size of one hundred.
Table 4.3: The processing time measured in the data processor’s Java component

<table>
<thead>
<tr>
<th>Method</th>
<th>Average</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartProcessLogging</td>
<td>128.8</td>
<td>5.1</td>
</tr>
<tr>
<td>LogData</td>
<td>64.2</td>
<td>3.8</td>
</tr>
<tr>
<td>EndProcessLogging</td>
<td>3.6</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Using the values from Table 4.2 and 4.3, we have generated some pie charts to illustrate how the execution time of the modified osTicket can be broken down into a few constituent parts. In Figure 4.18, we show the execution time distribution of the ticket creation process, which involves one call to the `StartProcessLogging()` method and one call to the `LogData()` method. Figure 4.19 shows the execution time for the procedure of replying to an open ticket. This procedure involves a single call to the `LogData()` method. In the ticket deletion process, whose execution time distribution is illustrated in Figure 4.20, there is only one call to the relatively fast method `EndProcessLogging()`. 
4.6. PERFORMANCE

Figure 4.19: The distribution of execution time in the procedure for replying to an open ticket in the modified osTicket

Figure 4.20: The distribution of execution time in the ticket deletion procedure of the modified osTicket


4.7 Summary

In this chapter, we have shown how the components described in Chapter 3 can be used to execute the scenario laid out in Section 2.3. We show this process from point of view of both the customer and the staff members at the three involved entities. We have also summarised the osTicket implementation work in terms of the amount of new code and the performance of the final product. Based on our experience working with the logging system we have also suggested some improvements that could simplify the logging APIs.
Chapter 5

Conclusion

This project has produced an improved proof-of-concept logging system implementation as well as logging support for a piece of real world open source software. We also have an intuitive usage scenario that demonstrates how the logging system can be utilised in a way that is valuable to the data subject. We have done all this because we wanted to evaluate the logging system implementation from the point of view of an independent developer wishing to make use of the system. We also wanted to demonstrate the potential of the logging system to the average user in a visual format.

5.1 Implementation

The data processor and data subject implementations both consist of a more or less user friendly front-end communicating with a Java back-end. The data processor is the osTicket software which is written in PHP and communicates with its Java component via SOAP over HTTP. The data subject application front-end is a browser extension which is written in JavaScript and communicates with the back-end via JSON encoded objects transmitted over HTTP. Using these components we are able to stage the scenario we designed in Chapter 2, wherein the data subject creates a ticket regarding a computer problem, that
plays out over three separate data processors. As the scenario is performed, the data subject can at any time see the updated status of the process via a visualisation page that is included with the browser extension.

5.2 Scenario

When designing the scenario we took many things into consideration. The foremost of these was that the scenario should show how the logging system can be easy to use and provide value for the data subject. For a customer using the data subject application we have implemented, no extra work is required to make use of the modified osTicket version. Using the data subject application the customer can keep up to date with the state of an issue as it unfolds over multiple tickets.

5.2.1 Scenario choice

Issue tracking systems in general, and osTicket in particular, have a number of qualities that made them suitable for our scenario. Primarily, we wanted the resulting demonstration to be relateable to the average user, and we wanted it to be driven by user interaction, to facilitate a step by step approach. For transparency logging of potentially privacy sensitive information, ideal areas (that are often brought up as suitable areas in literature [10]) are eHealth and eGovernment services. Unfortunately, we were not able to find any good examples of open source services in these areas. Consequently, we chose to move forward with osTicket as it is well suited for the purpose of demonstration, although it does not present clear opportunities to highlight privacy or transparency issues.

5.2.2 Privacy

As previously mentioned, our chosen scenario does not show the benefit of the privacy preserving aspect of the logging system in a prominent way. However, there are some
benefits to using a privacy preserving log system in this setting that may not be immediately apparent. For example, the customer retrieves log entries from the specialist’s log server in an anonymous way. If there is no identifying information on the hard drive (e.g. if the disk is encrypted) and no such information is passed from the service entity to the specialist, the customer could potentially be fully anonymous toward the specialist entity. The same anonymity could be provided at the service entity as long as there was a solution for returning the submitted hardware to the customer anonymously. The privacy preserving aspect of the logging system ensures that it will not interfere with any future efforts by the companies to make their services more privacy friendly.

5.2.3 Transparency

Another concession that we made in choosing an issue tracking system as the basis of our scenario is that the customer’s personal data is not processed in any significant way by the data processors. The logs that are kept now do not carry transparency information but more practical information that is accessed in a privacy friendly manner. Potential information that can be logged within our scenario for the purposes of transparency would be (i) what staff members have viewed a ticket and (ii) what information is shared when the ticket is continued at another entity.

5.3 Performance

In Section 4.6 we presented the results of a set of performance tests. We also reasoned that we would be able to use these results, although they are rough and based on a proof of concept implementation, when considering the suitability of the logging system for potential applications.

When examining the results of the performance tests, perhaps the first thing that one notices is the significant increase in execution time of the ticket creation process. Including
two calls to the Java component, to create a logging process and to log an initial message, the execution time is increased to more than ten times its normal value. This may be unsuitable for some applications where performance is vital, and where there would be quick turnover of processes. For other applications, however, the performance of the LogData() method at the Java component may be of greater importance. Although the execution time is around half that of StartProcessLogging() it would in most applications be executed far more often. It is important to note that the execution time of both calls, but StartProcessLogging() in particular, takes place in part on the log server. This means that the full processing load does not need to lie with the data processor.

One of the reasons for the long execution time of StartProcessLogging() is that it involves writing a log message with the API call LogData(). This accounts for close to half its execution time. This first log message is written by the data processor as an assurance that it is committing to executing a certain process for a the data subject to whom the log pertains. In our scenario the process for creating a ticket involves starting a new logging process, and storing an initial log message, separately from the commitment message. It is conceivable that the data processor API could be extended to allow applications to merge such an initial log message with the commitment message. In our osTicket implementation this would result in a reduction in execution time of around 25% for the ticket creation process. Other applications may benefit similarly from such a mechanism.

### 5.4 Project evaluation

At the start of this project we set out to investigate how well the proof of concept logging system we had implemented would work with a real world product. We also wanted to produce an application for the data subject that was more user friendly than the one we had made so far, which was strictly text-based. We have managed to do both of these things, to an extent that is consistent with what we initially hoped for, within the given
time frame. The open source product we chose to integrate the logging system with has rudimentary logging support, and to extend it further, e.g. by logging other data, would be trivial. Furthermore, the browser extension that we implemented as the data subject application is a tremendous improvement over its predecessor in terms of illustrating what the logging system actually does. A personal challenge for us, in this project, has been to work with PHP, JavaScript, HTML and CSS; languages with which we had only basic proficiency beforehand.

When planning the project we also considered an evaluation of deploying log servers in Amazon’s Elastic Compute Cloud\(^1\) in terms of security and data subject privacy. This idea went so far as to be written into the original time plan but eventually it had to be cut.

When implementing the data subject’s Java component we ran into an interesting issue in translating our data structures to JSON. As the return value for the method \texttt{GetTrail()}\(^1\), we used a fully recursive structure similar to a linked list where each log entry contained the next, from the first to last. To translate this structure to JSON we used a call to the translation method for each level of recursion. When a JSON object was translated a second time (as its containing object was being translated) any quotation marks ("), which are abundant in JSON, would be escaped by prepending them with a backslash (\). For each subsequent translation pass both the quotation mark and its preceding backslashes would be escaped again by prepending another backslash to each. This caused the amount of backslashes to increase exponentially, and the length of the return value from the \texttt{GetTrail()} method to grow out of hand after only a handful of log entries. This logical error was not found until it finally caused the data subject application to crash. The final version uses another data structure which is fully constructed in Java before it is translated in a single pass, i.e. without escaped characters.

5.5 Key contributions

In this thesis we have presented the first integration of a real world service with the logging scheme presented by Pulls et al. [10], and its proof of concept implementation. We have made a number of suggestions for areas of improvement for said logging system, such as integrating the data vault in the API and adding another API method for generating cascade entries. We have presented some performance results from our implementation that may be useful for future development and discussion. In addition to the performance tests presented in Section 4.6, we have also carried out more accurate tests on a subcomponent of the log server API. The results of these tests were included by Vliegen et al. in [12], and are available in Appendix F.

5.6 Future work

One possible avenue of future development would be to assess what information other than what is currently logged from osTicket may be interesting. For example, in highly sensitive usage scenarios it may be relevant to log every time a staff member views a ticket. Access to the data processor and data subject Java components should be restricted as they are currently accepting all requests which is wholly unsuitable for real world usage.

The data subject could be improved by translating the Java component to JavaScript and integrate it with the browser extension. This would simplify the design of the application as a whole and make it easier to deploy. If the log visualisation page is to be used beyond the given scenario then it would require some work to generalise the code, especially when it comes to the meta entries as they have special visual representations on the page.

The audit components that were added to the proof of concept logging system have a few critical problems, mostly regarding security. As noted in Section 2.1.5, where the auditing implementation is described, the timestamping system used cannot be considered
secure. This is because the private key used in creating the timestamp must be available to the log server as timestamping is performed locally. The implementation should instead contact an external TSA. Furthermore, the current TSA implementation does not perform linked timestamping, which is recommended by Pulls et al. [10]. Another issue with the audit components is that they do not have any way to output the audit data to auditors. The audit components now store one version of the audit data and overwrite it when a new audit round is completed and all values are verified. In a real system, the audit data should be stored in such a way that it is inaccessible to the data processor after it has been written, and all audit rounds should be stored indefinitely. Until such a solution is in place, we have deemed it unnecessary to create a procedure for outputting audit data.

Another possibility for future development is to integrate the logging system with other potential use cases. Because we chose to make network interfaces for the Java components it should be possible to re-use them in other projects, regardless of the programming language used. Furthermore, with class.processor.php (listed in Appendix D.2), the time required to implement prototype logging support for PHP projects should be minimal. This notion is backed by the osTicket logging implementation, which was accomplished with a total of 113 lines of added PHP and HTML code (excluding class.processor.php).

The logging system implementation used in this project also has some future work listed by Pulls et al.. For example, the logging system currently is not persistent, meaning that when a data processor or log server is restarted for any reason all data is lost, including log entries.
5.7 Final words

The issue of privacy on the Internet is one many care about, but can be overlooked by the average Internet user for the sake of convenience. Hopefully, this project has been one step toward privacy enhancing technologies that are both technologically advanced and user friendly.
References


Appendix A

Glossary

**API**  Application Programming Interface, the well defined interface consisting of methods, classes and data structures with which one software component interacts with another.

**CSS**  Cascading Style Sheets, a style sheet language that is most notably used for defining the look of HTML web pages.

**HTML**  Hypertext Markup Language, a language used for creating web pages.

**HTTP**  Hypertext Transport Protocol, an application protocol most notably used for accessing web pages on the World Wide Web.

**JSON**  JavaScript Object Notation, a human readable format for data representation originating from JavaScript.

**MySQL**  A widely used relational database management system.

**PHP**  PHP: Hypertext Preprocessor (recursive acronym), a scripting language used primarily in web development.
**SOAP**  A protocol for exchanging structured information used by Web Services.

**TSA**  Time-Stamping Authority, a service where a central authority will issue timestamps with as strong guarantees of validity and accuracy as possible.

**Unlinkability**  Unlinkability, as it is used in this thesis, means that an attacker cannot, given only information from our logging system, determine whether two items of interest are related or not. This definition is based on the unlinkability definition by Pfitzmann and Hansen [9].

**URI**  Uniform Resource Identifier, a string of characters uniquely identifying a resource.

**URL**  Uniform Resource Locator, a type of URI.
Appendix B

Data processor’s Java component

WSDL

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI’s version is JAX-WS RI 2.2.4-01. -->
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI’s version is JAX-WS RI 2.2.4-01. -->
<definitions
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://www.w3.org/2001/XMLSchema/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema/
    xmlns:wsdl="http://soapProcessor.dataProcessor/"
    targetNamespace="http://soapProcessor.dataProcessor/"
    name="SOAPProcessorService">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://soapProcessor.dataProcessor/"
                          schemaLocation="http://127.0.0.1:9876/example?xsd=1"/>
        </xsd:schema>
    </types>
    <message name="StartProcessLogging">
        <part name="parameters" element="tns:StartProcessLogging"/>
    </message>
    <message name="StartProcessLoggingResponse">
        <part name="parameters" element="tns:StartProcessLoggingResponse"/>
    </message>
</definitions>
```
<message name="LogData">
    <part name="parameters" element="tns:LogData"></part>
</message>

<message name="LogDataResponse">
    <part name="parameters" element="tns:LogDataResponse"></part>
</message>

<message name="EndProcessLogging">
    <part name="parameters" element="tns:EndProcessLogging"></part>
</message>

<message name="EndProcessLoggingResponse">
    <part name="parameters" element="tns:EndProcessLoggingResponse"></part>
</message>

<message name="GetCValue">
    <part name="parameters" element="tns:GetCValue"></part>
</message>

<message name="GetCValueResponse">
    <part name="parameters" element="tns:GetCValueResponse"></part>
</message>

<message name="CascadeKey">
    <part name="parameters" element="tns:CascadeKey"></part>
</message>

<message name="CascadeKeyResponse">
    <part name="parameters" element="tns:CascadeKeyResponse"></part>
</message>

<message name="WriteCascadeEntry">
    <part name="parameters" element="tns:WriteCascadeEntry"></part>
</message>

<message name="WriteCascadeEntryResponse">
    <part name="parameters" element="tns:WriteCascadeEntryResponse"></part>
</message>

<portType name="SOAPProcessor">
    <operation name="StartProcessLogging">
    </operation>
    
    <operation name="LogData">
        <input wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/LogDataRequest" message="tns:LogData"></input>
        <output wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/LogDataResponse" message="tns:LogDataResponse"></output>
    </operation>
    
    <operation name="EndProcessLogging">
        <input wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/EndProcessLoggingRequest" message="
</input>

</operation>

<operation name="GetCValue">
<input wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/GetCValueRequest" message="tns:GetCValue"></input>

<output wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/GetCValueResponse" message="tns:GetCValueResponse"></output>
</operation>

<operation name="CascadeKey">
<input wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/CascadeKeyRequest" message="tns:CascadeKey"></input>

<output wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/CascadeKeyResponse" message="tns:CascadeKeyResponse"></output>
</operation>

<operation name="WriteCascadeEntry">
<input wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/WriteCascadeEntryRequest" message="tns:WriteCascadeEntry"></input>

<output wsam:Action="http://soapProcessor.dataProcessor/SOAPProcessor/WriteCascadeEntryResponse" message="tns:WriteCascadeEntryResponse"></output>
</operation>

<portType>
</portType>

</binding name="SOAPProcessorPortBinding" type="tns:SOAPProcessor">
</soap:binding>
</operation name="StartProcessLogging">
</soap:operation>
</input>

<soap:body use="literal"></soap:body>
</input>

<output>
<soap:body use="literal"></soap:body>
</output>

<operation name="LogData">
<soap:operation soapAction=""></soap:operation>
</input>

<soap:body use="literal"></soap:body>
</input>

<output>
<soap:body use="literal"></soap:body>
</output>

<operation name="LogData">
<soap:operation soapAction=""></soap:operation>
</input>

<soap:body use="literal"></soap:body>
</input>

<output>
<soap:body use="literal"></soap:body>
</output>

<operation>
<operation name="EndProcessLogging">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>

<operation name="GetCValue">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>

<operation name="CascadeKey">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>

<operation name="WriteCascadeEntry">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>

<service name="SOAPProcessorService">
<port name="SOAPProcessorPort" binding="tns:SOAPProcessorPortBinding">
<soap:address location="http://127.0.0.1:9876/example"/>
</port>
</service>
</definitions>
Appendix C

open.js

```javascript
var initData;
if (document.forms.open) {
    document.forms.open.puk.readOnly = true;

    $.ajax(
        {
            type: "GET",
            url: "http://localhost:11140/getpuk",
            dataType: "json",
            success: function(data) {
                if(data.status)
                    document.forms.open.puk.value = data.puk;
                else
                    document.forms.open.puk.value = data.status;
        }
    });
} else if (document.forms.initdata) {
    initData = document.forms.initdata.data.value;
    $.ajax(
        {
            type: "GET",
            data: { "initdata": initData },
            dataType: "json",
            success: function(data) {
                if(data.status)
                    document.forms.initdata.data.value = "initData stored at index " + data.id;
                else
                    document.forms.initdata.data.value = "Error";
        }
    });
}
```
Appendix D

osTicket alterations

What follows are the two new files introduced to the osTicket project followed by a complete listing of changes made to the original osTicket files as generated by the program \textit{diff} of GNU diffutils\textsuperscript{1}.

D.1 include/staff/cascade.inc.php

```php
<?php
// Note that ticket is initiated in tickets.php.
if (!defined('OSTSCPINC') || !$thisuser->isStaff() || !is_object($ticket)) die('Invalid path');
if (!$ticket->getId() or (!$thisuser->canAccessDept($ticket->getDeptId()) and $thisuser->getId()!=$ticket->getStaffId())) die('Access Denied');

$cascadeData = Processor::cascade($ticket->getExtId());
?>
<form action="tickets.php?id=<?= $id ?>" method="post" enctype="multipart/form-data">
<input type='hidden' name='ticket_id' value="<?= $id ?>"/>
<input type='hidden' name='a' value="cascade"/>
<table align="left" cellpadding=2 cellspacing=1 width="90%">
  <tr><td align="left">
```

\textsuperscript{1}\url{http://www.gnu.org/software/diffutils/}, last accessed 2012-05-12.
```
D.2 include/class.processor.php

```php
<?php

class Processor {
    static $wsdl = 'http://127.0.0.1:9876/example?wsdl';

    public static function startProcessLogging($id, $puk) {
        $retval = 'ERROR';
    }
}
```
try {
    $processor = new SoapClient(Processor::$wsdl);
    $res = $processor->StartProcessLogging(array('ost_id'=>$id, 'puk' =>'puk'));
    $retval = $res->return;
} catch (Exception $e) {
    $retval = 'Data processor API connection failed: ' . $e->getMessage();
}
return $retval;

public static function endProcessLogging($id) {
    $retval = 'ERROR';
    try {
        $processor = new SoapClient(Processor::$wsdl);
        $res = $processor->EndProcessLogging(array('ost_id'=>$id));
        $retval = $res->return;
    } catch (Exception $e) {
        $retval = 'Data processor API connection failed: ' . $e->getMessage();
    }
    return $retval;
}

public static function logMessage($id, $message) {
    $retval = 'ERROR'
    try {
        $processor = new SoapClient(Processor::$wsdl);
        $res = $processor->LogData(array('ost_id'=>$id, 'message' =>$message));
        $retval = $res->return;
    } catch (Exception $e) {
        $retval = 'Data processor API connection failed: ' . $e->getMessage();
    }
    return $retval;
}

public static function cascade($id) {
    $retval = array();
try {
    $processor = new SoapClient(Processor::$wsdl);
    $res = $processor->GetCValue();
    $retval['c'] = $res->return;
    $res = $processor->CascadeKey(array('ost_id' => $id, 'c' => $retval['c']));
    $retval['puk'] = $res->return;
} catch (Exception $e) {
    $retval['err'] = 'Data processor API connection failed: ' . $e->getMessage();
}
return $retval;

public static function writeCascade($id, $postData) {
    if(!isset($_POST['c']) || !isset($_POST['initData'])) {
        return array('err'=>'Not all information available');
    }
    $c = $_POST['c'];
    $initData = $_POST['initData'];
    try {
        $processor = new SoapClient(Processor::$wsdl);
        $res = $processor->WriteCascadeEntry(array('ost_id' => $id, 'c' => $c, 'initData' => $initData));
        if($res->return == "Cascading commited to log")
            $retval = array('msg'=>$res->return);
        else
            $retval = array('err'=>$res->return);
    } catch (Exception $e) {
        $retval = array('err'=>'Data processor API connection failed: ' . $e->getMessage());
    }
return $retval;
}
D.3  client.inc.php

```php
> require_once(INCLUDE_DIR.'class.processor.php');
```

D.4  include/class.ticket.php

```php
$result = Processor::endProcessLogging($this->getExtId());
if($result != "13") {
    error_log("EndProcessLogging failed for process ".
    $this->getExtId()." with status ". $result);
}
```

D.5  include/class.validator.php

```php
case 'puk':
    if(strlen($this->input[$k]) != 128)
        $this->errors[$k]=$field['error'];
    break;
```

D.6  include/client/open.inc.php

```html
<form action="open.php" method="POST"
    enctype="multipart/form-data">
    <form name="open" action="open.php" method="POST"
    enctype="multipart/form-data">
    </form>
    <tr>
        <td>
        </td>
    </tr>
    <tr>
        <th>Public key:</th>
        <td>
            <input type="text" name="puk" size="35"
                value="<?= $info['puk']?>">
                &nbsp;<font
                class="error">*&nbsp;<?= $errors['puk']?></font>
        </td>
    </tr>
```

D.7  include/client/thankyou.inc.php

```php
$n = newintendent();
$n->start($arg1, $arg2); // LOGGING: Initialise data subject with processor
$initData = Processor::startProcessLogging($ticket->getExtId(), $_POST['puk']);
$result = Processor::logMessage($ticket->getExtId(), date("Ymd H:i:s") . " - Your ticket was opened with the following information provided: <b>" . $_POST['message'] . "</b>"; if($result != "13") {
  error_log("LogData failed for process ". $ticket->getExtId() . " with status ". $result);
}
```

D.8  include/staff/viewticket.inc.php

```php
$option = "cascade".
  $info['do'] = 'cascade'? 'selected': '';?
  > Branch Process</option>
```

D.9  open.php

```php
$n = newintendent();
$n->start($arg1, $arg2); // LOGGING: Initialise data subject with processor
$initData = Processor::startProcessLogging($ticket->getExtId(), $_POST['puk']);
$result = Processor::logMessage($ticket->getExtId(), date("Ymd H:i:s") . " - Your ticket was opened with the following information provided: <b>" . $_POST['message'] . "</b>"; if($result != "13") {
  error_log("LogData failed for process ". $ticket->getExtId() . " with status ". $result);
}
```
D.10  scp/tickets.php

D.10  scp/tickets.php

21a22
> require_once(INCLUDE_DIR.'class.processor.php');
86a88,95
>     // Write to the log
>     $result =
Processor::logMessage($ticket->getExtId(), date("Ymd H:i:s") . " - <i>" . $thisuser->getName() . "</i> posted a response:
<b>" . $_POST['response'] . "</b>");
>     if($result != "13") {
>         error_log("LogData failed for process ". $ticket->getExtId() . " with status ". $result);
>         $msg='Response Posted - Failed to submit log message!';
>     }
>
204a214,226
>     case 'cascade':
>         $page='viewticket.inc.php';
>         if(!$ticket || !$thisuser->canEditTickets()) {
>             $errors['err']='Perm. Denied. You are not allowed to edit tickets';
>         } else {
>             $status =
Processor::writeCascade($ticket->getExtId(), $_POST);
>             if(isset($status['msg'])) {
>                 $msg=$status['msg'];
>             } else {
>                 $errors['err']=$status['err'];
>             }
>         } else {
>             $errors['err']='Perm. Denied. You do not have the necessary permissions to branch processes.';
>         } else {
>             if($ticket->isOpen()) {
>                 $page='cascade.inc.php';
>             }
>         }
>         break;

320a343,351
>     }
>     break;
>     case 'cascade':
>         if(!$thisuser->canManageTickets() && !$thisuser->isManager()) {
>             $errors['err']='Perm. Denied. You do not have the necessary permissions to branch processes.';
>         } else {
>             if($ticket->isOpen()) {
>                 $page='cascade.inc.php';
>             }
>         }
>     break;
Appendix E

Logging system class diagrams

The following diagrams were generated using ObjectAid UML Explorer\(^1\) for Eclipse.

E.1 Pre-existing

The class diagrams in this section show only classes that existed before the start of this project.

\(^1\)http://www.objectaid.com/, last accessed 2012-05-07.
E.1.1 Data processor
E.1.2 Data subject

The class diagrams in this section show the audit components that were added to the proof of concept logging system as part of this project, and how they fit into the data processor.
and log server implementations.

E.2.1 Data processor
E.2.2 Log server

- <<Java Class>>
  - LogServerWithHardware
    - LogState: TrustedState
    - mStorage: Storage

- <<Java Class>>
  - SimpleAudit
    - mPrK: ECPrivateKeyParameters
      - initialise(ECPrivateKeyParameters): void
      - isRoundFinished(ECPrivateKeyParameters): boolean
      - newEntry(ECPrivateKeyParameters, LogEntry): void
      - endRound(ECPrivateKeyParameters): byte[]
      - finish(ECPrivateKeyParameters, byte[]): byte[]

- <<Java Class>>
  - AuditDataEntry
    - roundCounter: int
    - mySig: byte[]
    - dsSig: byte[]
    - waitingForResponse: boolean

- <<Java Class>>
  - Timestamp
    - date: byte[]
    - signature: byte[]
    - Timestamp(byte[]): void
    - Timestamp(String)

- <<Java Class>>
  - LogEntry
    - data: byte[]
    - DCS: byte[]
    - CS: byte[]
    - Cp: byte[]
    - DCP: byte[]
    - LogEntry(byte[]): void
    - LogEntry(byte[]): void
    - LogEntry()
Appendix F

Performance tests on log server state

We have carried out a number of performance tests on the state subcomponent of the log server. In [12] Vliegen et al. present a hardware component that performs the duties of the state. As part of this presentation, hardware performance is compared to our results from the existing Java implementation. The following are the results, exactly as presented by Vliegen et al. The measurements marked HW are taken from the hardware and SW, correspondingly, means software. The two measurement for the hardware measure the execution time when the procedure involves the first entry in the state component (at index 0), and when it involves the last (at index 511, as the hardware state holds at most 512 entries).

<table>
<thead>
<tr>
<th></th>
<th>HW Target at 0 [ms]</th>
<th>HW Target at 511 [ms]</th>
<th>SW [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>createProcessorState</td>
<td>68.86</td>
<td>80.05</td>
<td>59</td>
</tr>
<tr>
<td>createSubjectState</td>
<td>69.11</td>
<td>91.49</td>
<td>60</td>
</tr>
<tr>
<td>createEntry</td>
<td>1.07</td>
<td>23.45</td>
<td>0.062</td>
</tr>
<tr>
<td>deleteSubject</td>
<td>9.71</td>
<td>20.90</td>
<td>0.020</td>
</tr>
<tr>
<td>getStateIC</td>
<td>0.37</td>
<td>22.75</td>
<td>41</td>
</tr>
</tbody>
</table>

Table F.1: Timing results