Adding Secure Transparency Logging to the PRIME Core

Hans Hedbom, Tobias Pulls, Peter Hjärtquist, Andreas Lavén

Adding Secure Transparency Logging to the PRIME Core *

Hans Hedbom, Tobias Pulls, Peter Hjärtquist and Andreas Lavén

Department of Computer Science, Karlstad University, Karlstad, Sweden
E-mail: Hans.Hedbom@kau.se, tobias@pulls.name, peter@hjartquist.se and a.laven@gmail.com

Abstract. This paper presents a secure privacy preserving log. These types of logs are useful (if not necessary) when constructing transparency services for privacy enhancement. The solution builds on and extends previous work within the area and tries to address the shortcomings of previous solutions regarding privacy issues.

1 Introduction

PrimeLife [6] is aiming at understanding the privacy implications for a user[1] in a networked world and at constructing concepts and tools that can help a user to regain control over her personal sphere. One goal is to increase the possibilities that a person has to know what really happens with her personal data, i.e., what data about her are collected and how they are further processed, by whom, and for what purposes. This is important in order to judge if the data are processed in a legal manner and whether they are correct. The concept usually used to describe these properties is the notion of transparency. Consequently, one of our goals within PrimeLife is to develop tools and concepts for increased transparency.

In order to audit or verify that custodians of personal information (usually called data controllers) are behaving according to agreed policies, some form of event log is needed to track the processing and access of data at the data controller's side. This log must be built in such a way that it cannot be tampered with and since the log itself also contains personal information it must be encrypted in order to protect these data. Ideally, the only entity able to read

* Part of the research leading to these results has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement n° 216483. The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.

[1] We imply that "user" and "end user" throughout this paper are also data subjects in the system.

the log entry should be the one that the log entry concerns (i.e., the data subject). Currently there exists some examples of secure logs (e.g., the secure log pattern in [9] and the Schneier-Kelsey log [8]) The Schneier-Kelsey log has been further developed by Holt [2] addressing problems of public verification and by Ma and Tsudik [3] discussing solutions to attacks on integrity using forward secure sequential aggregation. There is also an example of a secure log built in a privacy setting based on a Schneier-Kelsey log [7]. The paper by Sackmann et al. is primarily on detecting policy breaches using a secure log and protecting user's log entries from each other by using the userid as part of the symmetric encryption key. However, these solutions all have their shortcomings from a privacy perspective and none of them addresses the question of unlinkability of, and secure anonymous access to log entries. Some work of unlinkability in connection with logs have been addressed by [10]. However, this work primarily addresses the unlinkability of logs between logging systems in a eGovernment setting rather than unlinkability of log entries within a log. Further, they do not address the problem of an inside attacker nor provide anonymous access to log entries. Because of this we decided to design and build a privacy-preserving secure log module in Java that can log events to an SQL database in such a way that the different events are only accessible[2] by the data subject that the entry refers to while minimizing the linkability of the log entries referring to a specific data subject. The result of our design work will be described in this paper. In the following, Section 2 will give a short overview of the PRIME core while Section 3 discusses requirements for the log and the assumed attacker model. In Section 4 an overview of the different components in the log system is given and Section 5 explains the different internal states and secrets needed in the solution. Section 6 gives an explanation of the structure of the log and discusses how the different fields are used to fulfill the different requirements and Section 7 presents an analysis of the solution. Finally, Section 8 concludes the paper.

2 The PRIME core

Within the scope of the PRIME [5] project a working prototype of a privacy-enhancing Identity Management System has been developed. This prototype is referred to as the PRIME middle-ware and is situated between an application and the different underlying data sources. The purpose of the middle-ware is to monitor and control access to any stored or released personal data and to track what data has been released and to whom. The middle-ware consist of a server-side component and a client-side component. However, both these components have the same functionality and capabilities and thus only play the role of client or server in a specific setting by configuration and not by design. The middle-ware component by itself consists of a number of components divided into the PRIME core and external components. In Figure 1 the external components are the PII LCM that handles obligation management, i.e, the upholdment of negotiated

[2] By accessible in this case we do not mean the log entry itself but rather the plain-text content of the log entry

rules for use, storage, deletion and release of personal data, the crypto module and the assurance manager that among other things handle the verification of code integrity in the PRIME middle-ware. The PRIME core consists of the system application interface and the primary access control (PAC) module. It is the PAC module and its different sub modules that control the access and release of stored and released data and keeps track of released information. For a more in-depth and thorough discussion of the PRIME middle-ware we refer to [1].

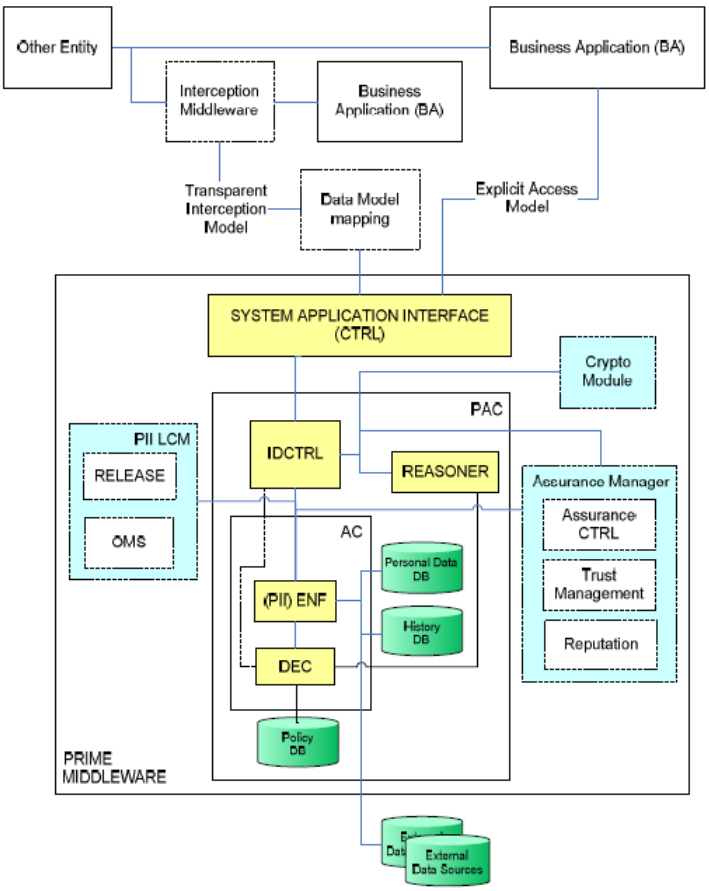

Fig. 1. The PRIME middle-ware [1]

3 Requirements and attacker model

As was mentioned briefly in the introduction we build this work on the secure log presented by Schneier-Kelsey [8] and further developed by Sackmann et al. [7]. However, we have modified and extended their ideas in order to further address the privacy problem and to try to overcome what we believe to be shortcomings in the previous solutions. The big differences are that we try to address the linkability problem in the log and that we use asymmetric encryption in some parts where the original solution uses symmetric encryption. The asymmetric encryption is used partly to solve the authentication problem and partly to guarantee irreversibility of committed log entries to any entity except the owner of the private key, i.e., the data subject that this entry refers to. The effect of this is that we can allow anonymous read access to the log entries. We have also tried to expand the integrity properties of the log using multiple hash chains.

The PRIME core itself is assumed to execute in a trusted environment having full control over its own data and execution. Thus the PRIME core is assumed to not behave maliciously if it is not compromised. Because of this we are concentrating on an attacker that either tries to compromise the PRIME core or that in one way or another manages to get access to the log entries. If the PRIME core is ever under the full control of an attacker there is little to be done in securing future log entries. However, it should not be possible for the attacker to alter the past without detection or to get knowledge about the content of previous events and log entries (sometimes referred to as perfect forward secrecy). Furthermore it should be hard for an attacker to link chains of log events to a specific data subject. All in all this gives us the following high level requirements on the log:

- It should not be possible for anybody except the data subject to decrypt log entries once they are committed to the log.
- It should not be possible to alter nor remove entries made prior to an attacker taking control of the data controller without detection.
- It should not be possible to link more than one log entry in the log referring to a specific data subject with that data subject except by the data subject herself[3].
- For efficiency reasons the solution should as far as possible not require that the whole log database is fully traversed by any entity or sent as a whole to the data subject.

4 Overview of the Log components

The general architecture of the log system is described in Figure 2 and consists of the components described below. Even though the full architecture is described, only the gray components have been developed in the first attempt.

[3] Ideally we would like to make it impossible to link any entry, however, our current solution makes one entry per data subject identifier linkable to that data subject identifier.

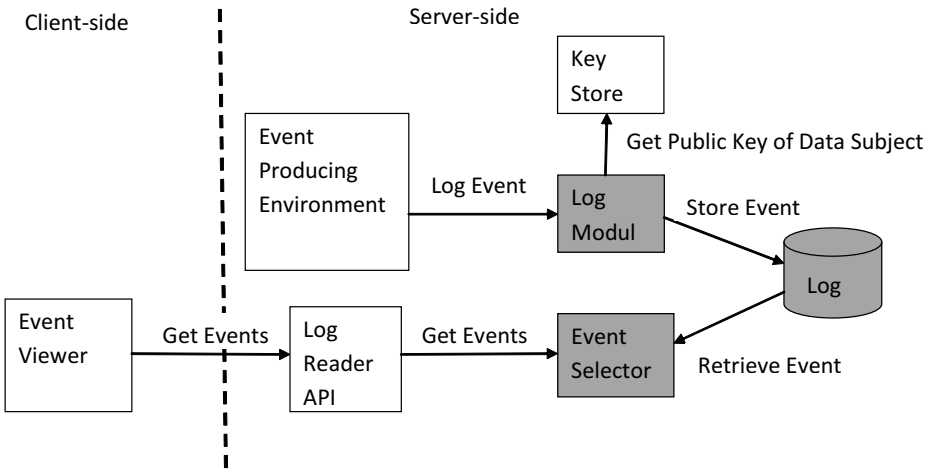

Fig. 2. Components in the log system

Each component in the figure is briefly described below:

The event producing environment: This is the component that is under audit and produces log event objects. These event objects consist of at least the subject (i.e., the identifier of the entity performing an action), the action (i.e., what was done), the purpose (i.e., why it was done), the object (i.e., the personal data that the action was performed on), and the data subject (i.e., the identifier of the "owner" of the personal data)[4].

The key store: This is a server side protected storage that contains public keys of data subjects and can hand them out to the log module if handed a data subject identifier. Where the key store is situated and what type of public key that is used is dependent on the application. However, in the PRIME case the key will be a self signed public key stored together with the PRIME data subject identifier (see foot note 3) in the personal data DB (see Figure 1) on the server side.

The log module: This is a module that receives log events and transforms them into secure privacy preserving log entries with the help of the public key of the data subject and stores them in the log.

The event selector: This is a module that given an entry identifier retrieves the requested log entry in the log to the requester.

The log reader API: This is a wrapper API that provides controlled or anonymous access to the Event selector depending on the requirements of the service.

[4] Please note that as soon as personal data is stored in any way in the PRIME system an identifier is generated. This is true even for anonymous access. However, the identifier might not be linked to a known user i.e., it might be a transaction pseudonym.

The event viewer: This is a module that presents and decrypts the events associated with a data subject in a user friendly fashion. It also contains functionality for searching, sorting, and comparing events as well as functionality for deciding if policy violations have occurred.

In the general case the event producing environment can be any trusted module capable of calling the log module. However, in our solution it is assumed to be the PRIME middle-ware configured as a server.

5 The log functionality

This section describes and discusses the different states needed in the client and the server in order for the solution to work. It also describes the procedures for storing and accessing log entries.

5.1 Secrets

The following secrets are needed in the scheme.

Secrets known and stored by the server:

1. SAS_0- A random number constituting the initial server secret used to authenticate all entries in the log for the server. This is also used as part of generating all the ServerIDs in the log.
2. $ServerID_0$- A random number constituting the initial ServerID seed.

Ideally these values are never directly stored on the server but securely stored somewhere else and only used when the integrity of the whole log needs to be verified (see Section 5.6). Instead the server initially stores SAS_1 and $ServerID_1$ calculated using formula 1 and 3 (see Section 5.2).

Secrets known and stored by each client for each data subject identifier used by a data subject using the client:

1. DSS_0- A random number constituting the initial data subject's secret used to authenticate all entries relating to the data subject identifier for the client. This is also used as part of generating all the data subject identifier's EntryIDs in the log.
2. $EntryID_0$- A random number constituting the data subject's initial EntryID seed for the data subject identifier.

The server gets DSS_1 and $EntryID_1$ in its first contact with the data subject and never needs to know DSS_0 and $EntryID_0$. The client calculates DSS_1 and $EntryID_1$ using DSS_0, $EntryID_0$ and formula 2 and 4 (see Section 5.2).

5.2 Log field and state value calculations

This section describes the calculations needed when adding an entry to the log. The states stored outside of the log is described in Section 5.3. Please note that values stored outside of the log are overwritten when new values are calculated, e.g., DSS_i is overwritten by DSS_{i+1}.

The following notation is used:

$hash(x)$ =any cryptographically secure one-way hash function.
$ENC_{PU_{DS}}(x)$= encryption of x under the public key of the data subject.
$SIGN_{PR_s}(x)$ =a digital signature on x using the private key of the server.
$HMAC_Y(x)$= The HMAC of x using the key Y.
i and j are indices where i is defined from $0..m$ and j from $0..n$; where m is the maximum number of log entries related to the data subject and n is the maximum number of log entries in the entire log.

The following calculations need to be performed for each log entry where DS always refer to the data subject identifier related to the log entry.

1. $SAS_{j+1} = hash(SAS_j)$ The SAS is used as an authentication key for the server fields used for integrity validation in the log. When new values are calculated the old SAS is permanently overwritten by the new one i.e., SAS_i is overwritten by SAS_{i+1}. This makes it computationally hard for anyone not knowing SAS_0 to recreate already used keys and thus computationally hard to alter stored entries.
2. $DSS_{i+1} = hash(DSS_i)$.The DSS is used as an authentication key for the data subject fields used for integrity validation in the log. When new values are calculated the old DSS is permanently overwritten by the new one, i.e., DSS_i is overwritten by DSS_{i+1}. This is used for the same reasons as SAS but for the data subject fields.
3. $ServerID_{j+1} = hash(ServerID_j, SAS_{j+1})$. This value is used by the server to identify the different log entries in the integrity verification process. It is hashed to make it computationally hard for anyone not knowing $ServerID_0$ to order the entries (see Section 6). The SAS parameter is used to make it computationally hard for anybody not knowing SAS_0 to forge $ServerIDs$.
4. $EntryID_{i+1} = hash(EntryID_i, DSS_{i+1})$. This value is used by the data subject to identify the different log entries in the integrity verification process. It is hashed to make it computationally hard for anyone not knowing $EntryID_0$ to order the entries or to link them to a specific data subject identifier(see Section6). The DSS parameter is used to make it computationally hard for anybody not knowing DSS_0 to forge $EntryIDs$.
5. $Data_{i+1} = ENC_{PU_{DS}}(SIGN_{PR_s}(logData), logData, nonce)$. The data field is encrypted with the data subject's public key and contains the $logData$ to be stored in the log, a signature of the $logData$ and a $nonce$. The signature, created using the server's private key, allows the data subject to prove that the data was committed to the log by the server. The $nonce$ is used to make it harder for an attacker, once having gained access to the log database, to

link entries to data subjects by generating common log entries and matching them to entries stored in the log.

6. $DataSubjectChain_{i+1} = HMAC_{DSS_{i+1}}(DataSubjectChain_i, EntryID_{i+1}, Data_{i+1})$. The $DataSubjectChain$ authenticates the log entry and all previous log entries for the data subject. The $DataSubjectChain$ is keyed with the current DSS value for the data subject.
7. $ServerChain_{j+1} = HMAC_{SAS_{j+1}}(ServerChain_j, DataSubjectChain_{i+1}, Data_{i+1}, EntryID_{i+1}, ServerID_{j+1})$. The $ServerChain$ authenticates the log entry and all previous log entries in the entire log for the server. The $ServerChain$ is keyed with the current SAS value.

5.3 Server state

The server is assumed to have some form of persistent data structures storing information needed for the algorithm. We will refer to these persistent data structures as state tables. After each new log entry created the "next authentication key" and "latest entry" part of the state is updated for the server state table completely overwriting the old values in the process. The same procedure is repeated for the "next authentication key" and "latest entry" part of the state table for the specific data subject associated with the log entry.

Data subject states stored at the server
One entry in the state table for each data subject identifier that includes:

1. Data Subject Identifier - the PRIME Core system identifier.
2. Next authentication key - if i is the index of the latest entry made for the data subject then DSS_{i+1} is stored here.
3. Latest entry - the pair (EntryID, DataSubjectChain) of the latest entry in the log for the data subject. The DataSubjectChain is used by the server when generating the next log entry (as part of the new entry's DataSubjectChain) and the EntryID is used to generate the next EntryID.

Server states stored
Only one entry in the state table that includes:

1. Next authentication key - SAS_{j+1}.
2. Latest entry - the pair (ServerID, ServerChain) of the latest entry in the log made by the server. The ServerChain is used by the server when generating the next log entry and the ServerID is used to generate the next ServerID.

5.4 Adding a log entry

The following section describes the steps needed in order to add a log entry. The data for the event is assumed to be present in the $logData$ variable.

1. Create an empty log record X.

2. Retrieve the stored SAS_j and $ServerID_{j-1}$ and calculate $ServerID_j$ according to formula 3 Section 5.2. Store $ServerID_j$ in X and overwrite the server state $ServerID_{j-1}$ with $ServerID_j$ (see Figure 3).
3. Retrieve the stored DSS_i and $EntryID_{i-1}$ and calculate $EntryID_i$ according to formula 4 Section 5.2. Store $EntryID_i$ in X and overwrite the data subject state $EntryID_{i-1}$ with $EntryID_i$(see Figure 3).
4. Retrieve the stored data subject's public key DS_{PU}, the server's signing key PR_S and $logData$. Generate a random nonce and calculate $Data_i$ according to formula 5 Section 5.2 and store it in X.
5. Retrieve the stored $DataSubjectChain_{i-1}$ and calculate $DataSubjectChain_i$ according to formula 6 Section 5.2. Store $DataSubjectChain_i$ in X and overwrite the data subject state $DataSubjectChain_{i-1}$ with $DataSubjectChain_i$ (see Figure 4).
6. Retrieve the stored $ServerChain_{j-1}$ and calculate $ServerChain_j$ according to formula 7 Section 5.2. Store $ServerChain_j$ in X and overwrite the server state $ServerChain_{j-1}$ with $ServerChain_j$(see Figure 4).
7. Calculate SAS_{j+1} and DSS_{i+1} overwriting the old values in the process and store X in the log.

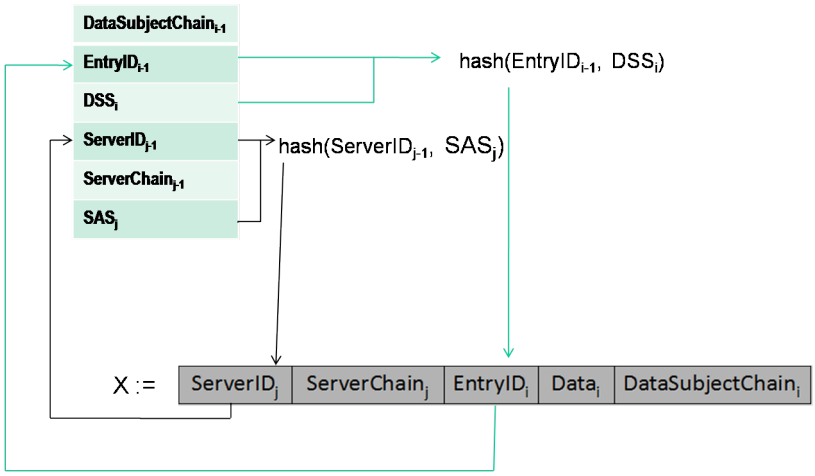

Fig. 3. Steps 2 and 3 in the algorithm

5.5 The Log API

1. GetLogEntry(EntryID) - returns the object(s) with the supplied EntryID. Since only a data subject knowing the right private key can decrypt the data field this method does not need the data subject to be identified and authenticated.

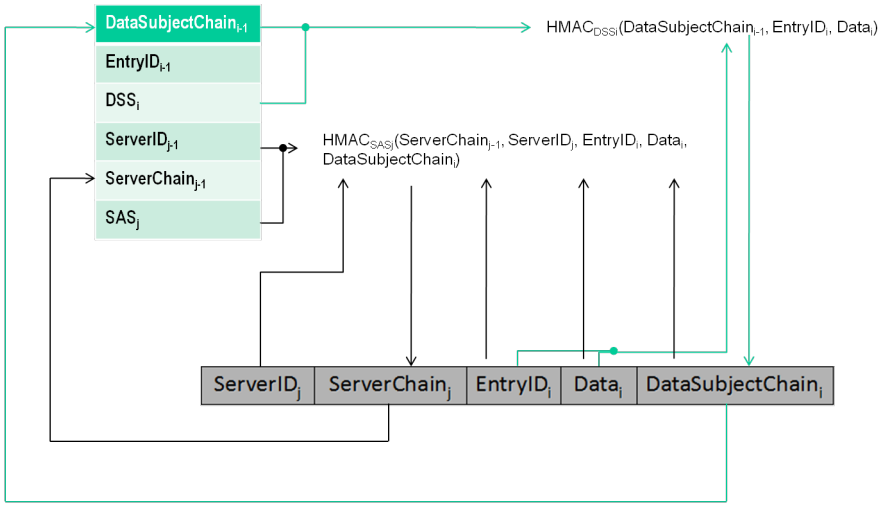

Fig. 4. Steps 5 and 6 in the algorithm

2. GetLatestEntryID(DataSubjectIdentifier) - returns a data structure containing the EntryID in the data subject state table for the data subject identifier and a nonce. The structure is encrypted with the public key stored for the data subject. This can be used by the data subject (but not fully relied on) when generating all the IDs of the log entries belonging to the data subject identifier. As above this method does not need the data subject to be identified and authenticated. The nonce is added to make the function generate different values each time it is called thus making it harder for an outside attacker to learn that new log entries have been added for the data subject identifier. Further this function should return seemingly valid responses for invalid DataSubjectIdentifiers making it harder for an outside attacker to deduce valid DataSubjectIdentifiers.

5.6 Operations

This section describes some key operations on the log i.e the client fetching entries in the log and the validation procedure on the server and the client side. All communication between the server and the client is assumed to be on an encrypted channel (the PRIME core uses SSL) and if anonymity is required the underlying network needs to be an anonymizing network, e.g., Tor as used in the PRIME Architecture [1]. We realize that the client behavior as described below can affect the linkability of entries to data subjects if not done properly. This issue is further discussed in Section 6.5.

Fetch all entries for a data subject identifier from the server The client is assumed to have knowledge of the initial data subject identifier's secrets DSS_0 and $EntryID_0$ for the data subject identifier.

1. Request GetLatestEntryID() from the server log API.
2. Generate and make a list of all IDs from $EntryID_1$ to the latest ID returned from the server using the formulas $EntryID_i = hash(EntryID_{i-1}, DSS_i)$, $DSS_i = hash(DSS_{i-1})$.
3. Request all the log entries based on EntryID in random order from the server (please see Section 6.5 where we discuss the need for the client to behave properly in order to keep unlinkability from an inside attacker).

Log Integrity Validation by the Client side

1. Fetch all the log entries from the server from $EntryID_1$ to GetLatestEntryID("your identifier") (see 5.6 above).
2. Generate the DataSubjectChain and compare it to the stored values in the entries. If it at any point doesn't match the validation fails.
3. Generate at least one more EntryID and request it from the server. If any entry is returned the validation fails.
4. Compare the recently downloaded entries in step 1 with the old entries (if any) stored in the client. If any entry differs or was not found on the server the validation fails.

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

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

6 Analysis of the solution

Like in the Schneier-Kelsey log[8] we are concerned about the security of the log entries committed to the log prior to an attacker compromising the server; once compromised little can be done to secure future commits to the log. When analyzing our solution we assume that all cryptographic primitives are ideal and that the anonymity set is of sufficient size, i.e., the event producing environment has produced events to log for a number of different data subjects for some time prior to the attacker taking control of the server.

6.1 The confidentiality of the log

The confidentiality of the data field in a log entry is dependent on how well the private key that decrypts the data field is kept secret by the corresponding data subject.

Claim. It is computationally hard for anybody except the data subject to decrypt log entries once they are committed to the log.

Justification. The properties of public key encryption and the use of the data subjects public key gives the property that it is computationally hard for anybody except the data subject to decrypt log entries once they are committed to the log.

6.2 The integrity of the log

The signature of the log data, as part of the data field once decrypted, allows the data subject to both validate the integrity of the log data and to prove that the log data was committed to the log by the server.

The DataSubjectChain and ServerChain, similar to the hash chain and MAC used by Schneier-Kelsey[8], serve to authenticate the log entry and all previous log entries for the data subject and server respectively. In fact, our chains are similar to what Holt discusses as a modification to his solution to enable cumulative verification ([2] Section 7). If we use a hash function instead of a MAC for the DataSubjectChain an attacker can simply use it to link all entries belonging to a data subject together. In the same way for the ServerChain, if we use a hash instead of a MAC, an attacker could order all entries in chronological order.

Claim. It is computationally hard to alter or remove entries made prior to an attacker taking control of the data controller without detection.

Justification: For an attacker to be able to modify an entry in the log in an undetectable manner she has to have knowledge of the right authentication keys for the DataSubjectChain and ServerChain fields for that entry. As stated in Section 5.3, all authentication keys are irretrievably overwritten as soon as a new key is generated. When an attacker takes over the server, all previously used authentication keys will therefore be inaccessible for the attacker. This prevents the attacker from falsifying either chain for any of the entries committed to the log prior to the attacker taking over the server. The process which detects any modifications made, for both a data subject and the server, is described in Section 5.6. The DataSubjectChain is used by the data subject to verify the integrity of the chain of log events relating to this data subject and the ServerChain makes it possible to verify the integrity of the whole log. Thus, the DataSubjectChain empowers the data subject to verify its log parts without having access to the whole log and the ServerChain can for example be used by auditors to verify that the log as a whole has not been tampered with.

6.3 Linking log entries to data subjects

Compared to the secure logs described by [8] and [2] the order of entries in our log is considered sensitive due to our requirement for a high degree of unlinkability between log entries and data subjects. The notion of order among our log entries is given by the EntryID and ServerID generation as described in Section 5. If an attacker was able to order all entries in the log in chronological order something as simple as an access log (like the Apache default access log) for a service using the PRIME Core together with some statistical analysis would probably aid an attacker greatly in linking entries to data subjects.

Claim. It is not be possible to link more than one log entry in the log referring to a specific data subject with that data subject except by the data subject itself.

Justification: An attacker, once having compromised the server and gained access to the log database, can by examining the server's state (see Section 5.3) link the latest entry in the log to a data subject for each data subject. The attacker can further deduce which entry was the latest entry made in the log and to which data subject it belongs to. Beyond what's already mentioned, barring any shortcomings as discussed in Section 6.5, an attacker needs to gain further knowledge about the secrets in the system to be able to link more entries to data subjects. To link all entries in the log belonging to a data subject an attacker needs to learn the initial data subject's secrets (DSS_0 and $EntryID_0$). This holds true as long as at least one other data subject's secrets are unknown to the attacker. If an attacker manages to learn the initial server secrets (SAS_0 and $ServerID_0$), all the entries in the log can be ordered in chronological order (based on the ServerID generation or by following the ServerChain as outlined in Section 5.2) which may severely affect linkability. Since we allow access to log entries based on EntryID anonymously, as stated in Section 4, an attacker will not be able to break unlinkability by simply waiting for data subjects to authenticate themselves towards the server as they download entries. However, an attacker looking at which entries are being accessed in the database and at what time might very well be able to link entries together; it all depends on how the client software behaves.

6.4 Two important scenarios explored

Imagine a very powerful attacker that at time t manages to compromise the server, accesses the log, learns the server's initial secrets and private key used for signing, learns the initial secrets and private keys for every data subject in the system with the only exception of the data subject identifier Bob's initial secrets and private key. The consequences are that the attacker can:

- Link all entries in the log to a data subject by generating all entry IDs belonging to each data subject except for Bobs. Bob's entries are the entries that remain unlinked to a data subject once all known data subject secrets have been used to generate IDs. Thus Bob's entries are also linkable.

- Read the contents of every entry in the log except for those belonging to Bob.
- Generate a valid ServerChain for every entry in the log and generate a valid DataSubjectChain for every entry in the log that doesn't belong to Bob.
- Sign any data with the server's private key.
- Update everything stored in the server's state (see Section 5.3), except for the entry concerning Bob, to a valid state.

This means that the attacker can replace any entry in the log, except for those belonging to Bob, with contents of her choosing with valid chains. She can also delete any entry except for those belonging to Bob and update the chains (and in some cases the server state) to a valid state. However, several factors prevent the attacker from in any way modifying any entry in the log without running a high risk of detection:

- Any of the compromised data subjects may already have downloaded an entry which the attacker has modified or deleted. This will be detected by the validation process as outlined in Section 5.6.
- Any modification to any entry belonging to Bob will result in an invalid DataSubjectChain due to a lack of authentication keys. In addition, the attacker will be unable to successfully update the server's state table for Bob with a valid DataSubjectChain or valid authentication key in the case of deletion. Last but not least, if any modifications are made to entries belonging to Bob all future entries committed to the log for the data subject Bob will have an invalid DataSubjectChain. All all of the above is detectable by Bob's validation process.
- Any modification to any entry in the log will result in the need to recreate the ServerChain for all the entries committed to the log after the modified entry. This change in the ServerChain will be detectable by every data subject's client upon validation if any entry committed to the log after the modified entry has been downloaded by any data subject prior to the modification.

Even with extensive knowledge of the secrets in the scheme and with access to the server an attacker is still severely limited when it comes to making undetected modifications to any entries in the log committed prior to compromising the server.

Another scenario that is of particular interest is if an attacker at time t has managed to compromise the server, gained access to the log and learned the initial secrets and private keys of all data subjects in the system leaving only the initial server secrets and the server's private key unknown to the attacker. For every entry in the log the attacker can link it to a data subject and read its contents. However, the ServerChain for any entry made prior to time t remains impossible for the attacker to modify without detection due to the lack of server authentication keys. In addition to having any change to any entry being detectable by the server validation process, there is also the chance that a data subject's client has downloaded an entry prior to the attacker modifying it.

6.5 Drawbacks and shortcomings

One drawback of our solution is that the authentication keys used for the DataSubjectChain and ServerChain fields are also used as part of the EntryID and ServerID generation respectively. This leads to a weakest link scenario where a flaw in either the MAC or hash algorithm used in an implementation will affect both the degree of unlinkability and the integrity of the log.

A problem is also that the behavior and functionality of the client software will play a role in the degree of unlinkability between data subjects and log entries in the scheme. For example, the validation process described in Section 5.6 can be used by an attacker who has compromised the server to link entries to data subjects simply by looking at when entries are requested (validation will cause a burst of requests that can be assumed to be from the same source). In addition the client also needs to have anonymous access to log entries based on entry ID; otherwise something as simple as tracking the IP-address of each request for an entry in the log would potentially allow an attacker to link log entries to data subjects (or at least to determine that some entries belong to the same data subject). It might be possible to address this issue using private information retrieval (see [4] and references there in). However, this is still subject for future work.

In [3] two security-related drawbacks are discussed for the Schneier-Kelsey log[8] and Holt [2] secure logs; a truncation attack and what the authors refer to as delayed detection. We claim that the truncation attack is only possible to a degree on our secure log when an attacker has extensive knowledge of the secrets in the scheme as discussed in the two examples in Section 7.3. Our secure log doesn't suffer from delayed detection from the server's point of view, since the server can validate the entire log independently.

7 Conclusions and Future Work

In this paper we have described the design of a privacy-friendly secure log for the purpose of making it possible for data subjects to get information on events relating to them on a server. The log builds on previous work and addresses primarily the questions of secure anonymous access to and unlinkability of log entries which previous work as far as we know have not addressed. We have implemented our design in Java as a standalone log thus showing that it is implementable. However, the lessons learned during the implementation have made us change the API slightly. These changes have not been implemented and thus the implemented version differs somewhat from the API presented in the paper. Further, we have not been able to do any extensive testing on the implementation regarding performance and penetration testing. When going from an idea to an actual implementation of the log system several questions are raised which could affect security and privacy;

1. Is it really possible to irreversibly overwrite the old authentication keys once stored in the server's state, i.e., memory?

2. Will the actual database used to store the log entries to some extent leak the order in which the entries were added to the database due to some internal structures or functionality?

Further research to find how these questions affect our solution is needed. We will also integrate the solution in to the PRIME core and implement a log view reader on the client side. By doing this we hope to find answers to the open issues on the optimal client behavior and the optimal logging strategy in order to balance performance and transparency.

Acknowledgment

The authors like to thank Professor Simone Fischer-Hübner for valuable input and suggestions during the work and Stefan Köpsell for fruitful discussions in connection with secure logs. We would also like to thank the participants and session chairs of the PrimeLife/IFIP Summer School for valuable inputs and suggestions.

References

1. M. Casassa-Mont, S. Crosta, T. Kriegelstein, and D. Sommer. Architecture v2. PRIME Deliverable D14.2.c, March 2007.
2. J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian workshops on Grid computing and e-research - Volume 54*, volume 167 of *ACM International Conference Proceeding Series*, pages 203–211. Australian Computer Society, 2006.
3. D. Ma and G. Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1), March 2009.
4. A. Pfitzmann, A. Juschka, A-K. Stange, S. Steinbrecher, and S. Köpsell. *Digital Privacy: Theory,Technologies and Practices*, chapter 2, pages 19–47. Auerbach Publications, 2008.
5. PRIME Project. https://www.prime-project.eu/.
6. PrimeLife Project. http://www.primelife.eu/.
7. S. Sackmann, J. Strüker, and R. Accorsi. Personalization in privacy-aware highly dynamic systems. *COMMUNICATIONS OF THE ACM*, 49(9), September 2006.
8. B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. *The Seventh USENIX Security Symposium Proceedings, USENIX Press*, pages 53–62, January 1998.
9. C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services and Identity Management.* Pearson Education Inc, 2006.
10. Karel Wouters, Koen Simoens, Danny Lathouwers, and Bart Preneel. Secure and privacy-friendly logging for egovernment services. In *3rd International Conference on Availability, Reliability and Security (ARES 2008)*, pages 1091–1096, Barcelona, Catalonia,ES, 2008. IEEE.