SLOPPI — a Framework for Secure Logging with Privacy Protection and Integrity

Felix von Eye, David Schmitz, and Wolfgang Hommel
Leibniz Supercomputing Centre, Munich Network Management Team, Garching near Munich, Germany
Email:{voneye,schmitz,hommel}@lrz.de

***Abstract*—Secure log file management on, for example, Linux servers typically uses cryptographic message authentication codes (MACs) to ensure the log file's integrity: If an attacker modifies or deletes a log entry, the MAC no longer matches the log file content. However, some privacy and data protection laws, for example in Germany, require the deletion or anonymization of log entries with personal data after a retention period of seven days. Such changes therefore do not constitute an attack. Previous work regarding secure logging does not support this use case adequately. A new log management approach with a focus on both the integrity and the compliance of the resulting log files with additional support for encryption-based confidentiality is presented and discussed.**

Keywords-log file management; secure logging; compliance

I. INTRODUCTION

System and application logs are of great value for administrators, e. g., for monitoring, fault management, and forensics. The predominant format for logs is plain text, which is the focus here; for example, the *syslog daemon* on UNIX and Linux systems, as well as applications like the Apache web server, store log entries in line-oriented plain text. Alternatively, proprietary binary or XML-based file formats as well as relational databases may be used; for example, this is used by the Microsoft Windows event log. Independent of the storage format, *secure* logs must fulfill the following basic criteria in practice:

- The log's *integrity* must be ensured: Neither a malicious administrator nor an attacker, who has compromised a system, may be able to delete or modify existing, or insert bogus log entries.
- The log must not violate *compliance* criteria [1]. For example, European data protection laws regulate the retention of personal data, which includes, among many others, user names and IP addresses. These restrictions also apply to log entries according to several German courts' verdicts that motivated our work.
- The *confidentiality* of log entries must be safeguarded; i. e., read access to log entries must be confined to an arbitrary set of users.
- The *availability* of log entries must be made sure of.

In a typical data center or network operations environment, the *integrity* criterion is usually fulfilled by using a trustworthy, central log server: Log entries are not (solely) stored locally on a device, but sent over the network to another machine; therefore, an attacker would have to compromise both this device and the log server before being able to manipulate the log without being detectable. The *compliance* criterion can be fulfilled by deleting old log files, e. g., after the common duration of seven days. As of today, *confidentiality* is typically handled in a per-file manner; for example, UNIX file system permissions are often used to make a log file readable for a specific user or group. Prospective modern logging facilities also enable confidentiality in a per-entry granularity, but are not yet in such wide use. Finally, the *availability* requirement for log files is the same as for other important data and typically ensured by data redundancy, i. e., copies and backups.

Our work is motivated by the large-scale distributed environment of the SASER-SIEGFRIED project (Safe and Secure European Routing) [2], in which more than 50 project partners design and implement network architectures and technologies for secure future networks. The project's goal is to remedy security vulnerabilities of todays IP layer networks in the 2020 timeframe. Thereby, security mechanisms for future networks will be designed based on an analysis of the currently predominant security problems in the IP layer, as well as upcoming issues such as vendor backdoors and traffic anomaly detection. The project focuses on inter-domain routing, and routing decisions are based on security metrics that are part of log entries sent by active network components to central network management systems; therefore, the integrity of this data must be protected, providing a use case that is similar to traditional intra-organizational logfile management applications.

In this paper, the focus is on *integrity* as well as *compliance* and the presentation of a novel, cryptographically enhanced log storage and processing approach that fulfills both criteria with or without a central log server. In the SASER-SIEGFRIED context, a central log server can use the presented framework to proof the logs' integrity so that the security metrics based on data from this log source can be considered reliable. On the other hand, the presented framework can also be used in other scenarios where a central log server may not be applicable or may not be a suitable solution, e. g.:

- Small organizations or small enterprises that do not have the resources, i. e., budget and/or skills, to operate a central log server.
- Machines and devices that do not have permanent network connectivity to a central log server, e. g., sensor networks, mobile devices, or servers that keep radio silence for a purpose, such as honeypots, which must appear to be easy prey for attackers.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-281-3

- Massively distributed systems with a high rate of log entries that do not need to be correlated on a regular basis, in which a central log server would become a performance bottleneck.

Especially in environments without trustworthy systems, such as a reliable central log server, cryptographic functions are used to make log files more secure. For example, message authentication codes (MACs) can be used to make log files evident of simple modifications: If an attacker manipulates or deletes a log entry, the log file's MAC no longer matches its content, making the attack obvious. However, previous work on secure logging did not account for desired changes to the log entries' content, such as the deletion of old log files. Doing so required the complex re-calculation of MACs and cryptographic hash values, impeding the practical use of these other approaches.

We present a secure logging approach that supports this use case of deleting old log files after an arbitrary retention period without re-keying. This approach uses cryptographic hash functions and asymmetric encryption to implement log entry integrity verification. Fresh cryptographic key material is generated after an arbitrary amount of time, e. g., daily, or when a certain number of log entries have been processed. However, unlike previous approaches, the solution fulfills the compliance criterion by allowing log files to be removed without violating the overall log data integrity and without the need to keep old cryptographic keys or re-calculate MACs for the whole log file. This approach also keeps the log files needed by applications in plain text, so they can, unlike a fully encrypted log file, be processed using any software tools that the administrator is already familiar with.

The presented solution can be extended to also ensure per-entry confidentiality and additional measures can be taken to improve the high availability. However, *integrity* and *compliance* are in the focus of this paper, which is structured as follows: In the next section, the terminology that is used throughout this paper is introduced. In Section III, previous approaches to secure logging and related work are summarized in order to outline the shortcomings that are addressed. The details of our approach are presented in Section IV. The paper closes with a conclusion in Section VI.

II. Terminology

The following terms and symbols are used in this paper with the specified meaning:

- The untrusted device $\mathcal{U}$, e. g., a web server. As a matter of its regular operations, $\mathcal{U}$ produces log data. It could be assumed that $\mathcal{U}$ may be compromised by an attacker and therefore the log data is not guaranteed to be *trustworthy*. However, this approach can be used to ensure the integrity and compliance of log data produced by $\mathcal{U}$, making it *reliable*.
- A trusted machine $\mathcal{T}$. In any related work and also in the presented work there is a need for a separate machine $\mathcal{T} \neq \mathcal{U}$. The working assumption in the related work is that $\mathcal{T}$ is secure, trustworthy, and not under the control of an attacker at any point in time. In the presented approach, $\mathcal{T}$ could be a connected printer, because a place is needed where only one initial key is saved, i. e., printed, without the possibility of intruder access.
- The verifier $\mathcal{V}$. The related work often differentiates $\mathcal{T}$ and $\mathcal{V}$; $\mathcal{V}$ then is only responsible for verifying the integrity and compliance of a log file or log stream. In this case, $\mathcal{T}$ is only used to store the needed keys and $\mathcal{V}$ has not to be as trustworthy as $\mathcal{T}$. Also in this case $\mathcal{T}$ is able to modify any log entry while $\mathcal{V}$ is not.

These symbols are used for cryptographic operations:

- A strong cryptographic hash function H, which has to be a one way function, i. e., a function which is easy to compute but hard to reverse, e. g., `sha-256(`m`)`.
- $\text{HMAC}_k(m)$. The message authentication code of the message m using the key k.

In our proposal we do not anticipate, which particular function should be used for cryptographic functions; instead, they should be chosen specifically based on each implementation's security requirements and constraints, such as available processing power and storage overhead.

Furthermore, without loss of generality, the terms *a)* log files, *b)* log entries, and *c)* log messages are discerned. A *log file* is an ordered set of *log entries*. For example, on a typical Linux system, `/var/log/messages` is a text-based log file and each line therein is a log entry; log entries are written in chronological order to this log file. *Log messages* are the payload of *log entries*; typically, log messages are human-readable strings that are created by applications or system/device processes. Besides a log message, a log entry includes metadata, such as a timestamp and information about the log message source.

As shown in Figure 1, the presented approach uses a couple of log files, which are related to each other in the following way. The *master log* L_m is only used twice a day to first generate and to then close a new integrity stream for the so-called *daily log* L_d. This log file L_d is used to minimize the storage needs for the master log L_m, which must never be deleted; otherwise, no complete verification could be performed. L_d is kept as long as necessary and contains a new integrity stream for the *application logs* L_a. These logs, e. g., the `access.log` or the `error.log` by an Apache web server or the `firewall.log` by a local firewall, are re-started from scratch once per day and yield all information generated by the related processes; they are extended by integrity check data. After a specified retention time – seven days in the scenario – these logs, which contain privacy-law protected data, have to be deleted completely because of legal constraints in Germany and various other countries. It is important to mention that a simple deletion would also remove any information about attempted intrusions and other attack sources. This would cover an intruder, who could be detected by analyzing the log files, so the log file should be analyzed periodically before this automated deletion. Other time periods than full days or a rotation that is based,

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-281-3

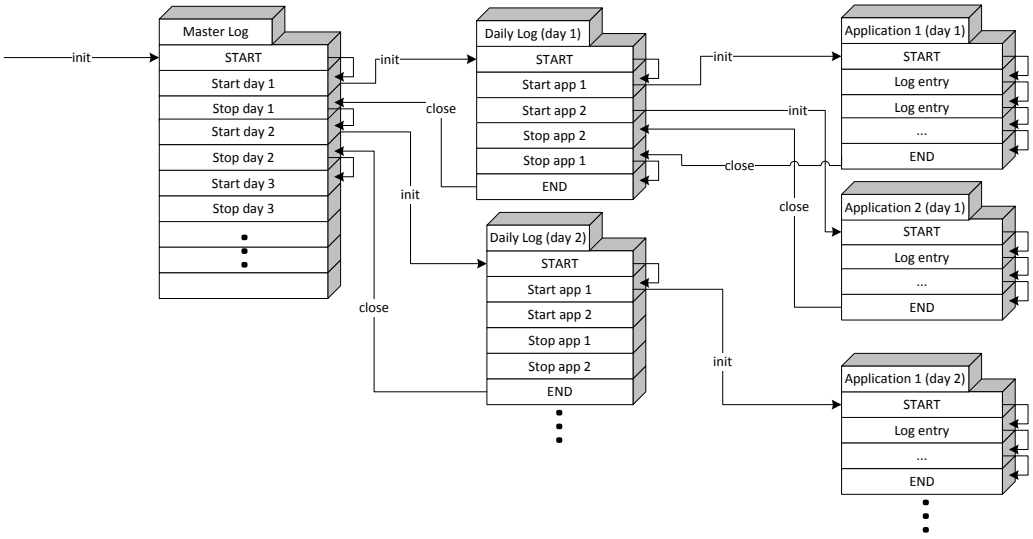

Fig. 1. Overview of all relevant log files.

e. g., on a maximum number of log entries per log file, as well as other deletion periods may be applied – but for the sake of simplicity *daily* logs and a seven day deletion period are used for the remainder of this paper.

III. Related work

With the exception of Section III-A, none of related work offers a possibility to fulfill compliance as it is not possible to delete logs afterwards. Complementary, the approach summarized in Section III-A does not address the integrity issues.

A. Privacy-enhancing log rotation

Metzger et al. [3] presented a system, which allows privacy-enhancing log rotation. In this work, log entries are deleted by log file rotation after a period of seven days, which is a common retention period in Germany based on several privacy-related verdicts. Based on surveys, Metzger et al. identified more than 200 different types of log entry sources that contain personal information in a typical higher education data center. Although deleting log entries after seven days seems to be a simple solution, the authors discuss the challenges of implementing and enforcing a strict data retention policy in large-scale distributed environments.

B. Forward Integrity

Bellare and Yee [4] introduced the term *Forward Integrity*. This approach is based on the combination of log entries with message authentication codes (MACs). Once a new log file is started, a secret s_0 is generated on $\mathcal{U}$, which has to be sent in a secure way to a trusted $\mathcal{T}$. This secret is necessary to verify the integrity of a log file.

Once the first log entry l_0 is written in the log file, the HMAC of l_0 based on the key s_0 is calculated and also written to the log file. To protect the secret s_0, there is another calculation of $s_1 = H(s_0)$, which is the new secret for the next log entry l_1. To prevent that an attacker can easily create

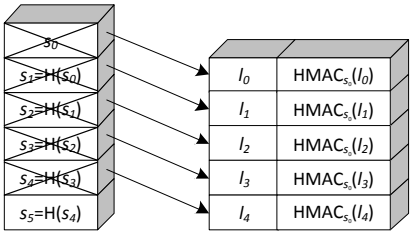

Fig. 2. The basic idea behind Forward Integrity as suggested in [4]

or modify log entries, the old and already used secret key for the MAC function is erased after the calculation securely. Because of the characteristic of one way functions it is not possible for an attacker to derive the previous key backwards in maintainable time. Figure 2 shows the underlying idea.

In their approach, in order to verify the integrity of the log file, $\mathcal{V}$ has to know the initial key to verify all entries in sequential order. If the log entry and the MAC do not correspond, the log file has been corrupted from this moment on, and any entry afterwards is no longer trustworthy.

However, the strict use of forward integrity also prohibits authorized changes to log entries; for example, if personal data shall be removed from log entries after seven days, the old MAC must be thrown away and a new MAC has to be calculated. While this is not a big issue from a computational complexity perspective, it means that the integrity of old log entries may be violated during this rollover if $\mathcal{U}$ has meanwhile been compromised.

C. Encrypted log files

Schneier and Kelsey [5] developed a cryptographic scheme to secure encrypted log files. They motivated the approach for encrypting each log entry with the need of confidential logging, e. g., in financial applications. Figure 3 shows the

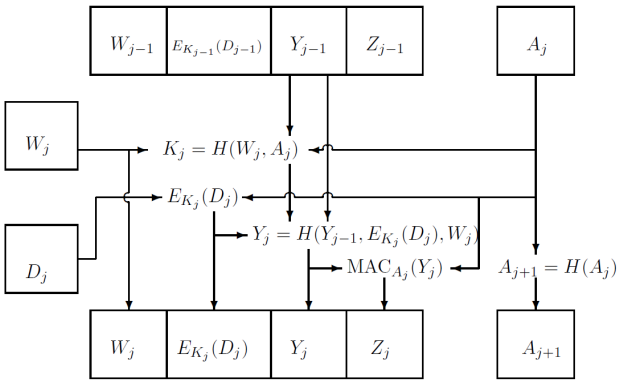

Fig. 3. The Schneier and Kelsey approach taken from [5]

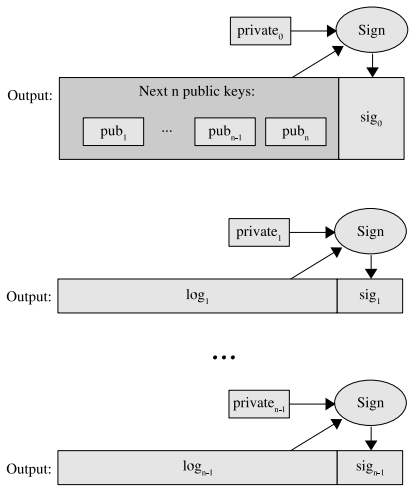

Fig. 4. The Holt approach taken from [6]

process to save a new log entry. Any log entry D_j on $\mathcal{U}$ is encrypted with the key K_j, which in turn is built from the secret A_j (in the paper s_j) and an entry type W_j. This entry type allows $\mathcal{V}$ to only verify predefined log entries. There is also some more information stored in a log entry, namely Y_j and Z_j, which are used to allow the verification of a log entry without the need of decryption of D_j. Therefore, only $\mathcal{T}$ is able to modify the log files.

However, this approach does not allow for the deletion method of log entries or parts thereof because then the verification would inevitably break.

D. Public key encryption

Holt [6] used a public/private-key-based verification process in his approach to allow a complete disjunction of $\mathcal{T}$ and $\mathcal{V}$. Therefore, a limited amount of public/private-key pairs are generated. The public keys are stored in a meta log entry, which is signed with the first public key, which should be erased afterwards securely. All other log entries are also signed with the precomputed private keys. If there are no more keys left, a new limited amount of public/private-key pairs are generated.

The main benefit of this approach is that the verifier $\mathcal{V}$ can not modify any log entry because he only knows the public keys, which can be used for verifying the signature but does not allow any inference on the used private key.

E. Aggregated Signatures

In scenarios where disk space is the limiting factor it is necessary that the signature, which protect each single log entry do not take much space. In all of the approaches sketched above, the disk usage by signatures is within $O(n)$, where n is the total amount of log entries. To deal with a more space-constrained scenario, Ma and Tsudik [7] presented a new signature scheme, which aggregates all signatures of the log entries. This approach uses archiving so that the necessary disk space amount is reduced to only $O(1)$.

The main drawback of this approach is that a manipulation of a single log entry would break the verification process but the verifier is not able to determine, which (presumably modified) entry causes the verification process to fail. As a consequence, it is also not possible to delete or to modify log entries, e.g., to remove personal data after reaching the maximum retention time.

IV. The SLOPPI framework

The survey of the related work shows that there is no solution yet that fulfills both necessary characteristics for log files: *integrity* and *compliance*. This approach combines key operations from those previous approaches in a new innovative way to achieve both characteristics. As introduced in Section II, a couple of types of log files, which are all handled a bit differently, are used for the framework.

First, the application log file L_a can be protected by any approach presented in Section III. After, for example, seven days these log files can also be deleted for compliance reasons. It is also possible to use log rotation techniques to fulfill local data protection policies. It is necessary to mention that any information about an attacker, which is not detected during this period, will be lost and cannot be recovered. But this is not a drawback of the presented framework because this is necessary to fulfill the data protection legislation especially in Europe or in Germany, which mandates to erase any privacy protected data after seven days. In scenarios where the log files can only be read after a longer offline period, e. g., low power sensor-networks devices, the period to delete log files should be set individually so an administrator is able to analyze any log data before they are deleted.

In the next step, the master log file L_m has to be secured. As it has only a few entries a day, it can be protected a public key scheme, e. g., RSA, to protect the log entries, which is described in detail in the upcoming Section IV-A. In the following work, the two keys of a public key scheme are called *signing key* (k_{sign}) and *authentication key* (k_{auth}).

Last but not least the daily log file L_d is considered in Section IV-B. Similar to the master log file, it only has very few entries per day, but they already must be considered too many entries for using public key schemes, so a symmetric key scheme is mostly the best choice.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-281-3

A. Master Log

To protect L_m, the following steps are followed:

- **Log initialization.** Whenever a new master log is initialized, $\mathcal{U}$ generates a authentication key (k^1_{auth}) and a signing key (k^1_{sign}), and sends k^1_{auth} to $\mathcal{T}$ over a secure connection, e. g., a TLS connection. k^1_{auth} and k^1_{sign} are the actual used secret. $\mathcal{U}$ can now initialize the log file by saving the first message `STARTING LOG FILE` in the log file as described next.
- **Saving new log entries.** Let m be the log message of the log entry to be stored in the log file. Between saving the last entry and the actual one, it can be assumed that there is enough time to generate a new authentication/signing key pair ($k^{n+1}_{\text{auth}}, k^{n+1}_{\text{sign}}$), while ($k^n_{\text{auth}}, k^n_{\text{sign}}$) is the actual secret. $\mathcal{U}$ now generates the log entry $m^* = (timestamp, m, k^{n+1}_{\text{auth}})$ and computes $\text{Enc}_{k^n_{\text{sign}}}(m^*)$. The last result is the new log entry, which is written to the log file. Immediately after calculating the encrypted result, the key pair (k^n_{sign},k^n_{auth}) is erased securely. Only the encrypted parts of the calculation are written to the log file.
- **Closing the log file.** If the master log file has to be closed, the last message `CLOSING LOG FILE` is saved into the log file. It is important that in this case it is not necessary to generate a new key pair and to save the next authentication key into the log file.

As L_m is used as meta log, which does not contain any application or system messages, we use message m. As mentioned before, the daily log is encrypted with a symmetric crypto scheme. Every day a new daily log is initialized by the system. The name and location of the created daily log is p_1. Furthermore p_2 is the first entry in L_d and finally p_3 appoints the necessary key for the log initialization step. m is then the concatenation of p_1, p_2, and p_3, e. g., `/var/log/2012-12-21.log;STARTING LOG FILE;VerySecretKey` together with $H(p_1, p_2, p_3)$.

Because of the need to detect manipulations of the L_m, it is necessary that m also contains a hash value of p_1, p_2, and p_3. With the knowledge of $H(p_1, p_2, p_3)$ it is possible to detect, where the decryption process failed.

To verify an existing master log, it is necessary to use the authentication key saved during the generation of the log file. With this key it is possible to decrypt the first entry, which leads to the next authentication key. This step can be performed until the actual last message or the `CLOSING LOG FILE` entry is reached. Because m consists of the necessary information about the daily log files, it is possible to verify any daily log that is still available. If a daily log has already been deleted, then this daily log and the connected application logs cannot be verified any more, but it is still possible to use the upcoming log entries of L_m.

B. Daily Log

Similar to the master log, the daily log is also processed in the three steps:

- **Log initialization.** Every day a new daily log has to be initialized. The above described systems $\mathcal{U}$ and $\mathcal{T}$ are in this case $\mathcal{U} = L_d$ and $\mathcal{T} = L_m$. $\mathcal{U}$ generates a symmetric key k_{sym} and sends k_{sym} to $\mathcal{T}$ together with the name and path of the actual log file and also the first message `STARTING LOG FILE`. The actual used secret is now k_{sym}. $\mathcal{U}$ can then initialize the log file by saving the first message `STARTING LOG FILE` in the log file as described next.
- **Saving new log entries.** As above, a symmetric key scheme is used for the daily log, e. g., `AES`. Let m be the message that has to be stored in the log entry and k_{sym} the actual used secret key. $\mathcal{U}$ now randomly choses a new secret key $k^{\text{new}}_{\text{sym}}$. Because of the use of symmetric key schemes, this step is not computationally expensive. The new log entry now is $\text{Enc}_{k_{\text{sym}}}(m^*)$ with the content $m^* = (timestamp, m, k^{\text{new}}_{\text{sym}}, H(timestamp, m, k^{\text{new}}_{\text{sym}}))$, which is written to the log file. The hash value is stored for verification purpose, so it is possible to detect the exact log entry, where a manipulation took place.
- **Closing the log file.** If the master log file has to be closed, the last message `CLOSING LOG FILE` is saved. This message, the file name and location, the MAC of the entire log file, and the last generated key are sent to the master log.

In the daily log, there are only three types of messages besides `STARTING LOG FILE` and `CLOSING LOG FILE`:

- `START APPLICATION LOG`. It contains a timestamp (in plain text), the file name and location of the log (plain text), the initialization key of the application log (encrypted), the first message of the application log (encrypted), and the file name and location of the log (encrypted). The encrypted parts of the message are condensed in one encryption step.
- `STOP APPLICATION LOG`. Similar to the start message, this message contains a timestamp (plain text), the file name and location of the log (plain text), the last key of the application log (encrypted), the last message of the application log (encrypted), and the file name and location of the log (encrypted). The encrypted parts are again condensed in one encryption step.
- `ROTATING APPLICATION LOG`. In case of a log rotation procedure this message contains a timestamp (plain text) and both file name before and after a rotation (plain text). As in the previous message type there are also the file names and locations in ciphertext.

It is important that all application logs, which have their starting message in a daily log, have to write their stopping message also to the same daily log. This is the reason why some contents in the log file are still in plain text. Otherwise the logging engine would have to remember, which application log is connected to which daily log. This also means that it is possible that a daily log is still open when the next daily log is initialized. The `ROTATING APPLICATION LOG` message could be in later daily logs because the specifics of the log

rotation algorithm are not known and it could be that a log rotation is performed only once a week.

The main reason to use the daily log is to reduce the space requirements of the main log. It is quite unusual that the main log is initialized for a second time if the system is running normally. There are round about two entries per day, which have to be stored over a long time. The daily log could be deleted after all application logs mentioned in this specific daily log are deleted. Depending on the amount of running applications on a server it is not unusual that there is more than one application log used on a system.

V. Verifying log entries and security analysis

To verify log entries, the initial master key is needed. As described above, each log entry in the master log is encrypted as $\mathrm{Enc}_{k_{\mathrm{sign}}^n}(m^*)$ with $m^* = (timestamp, m, k_{\mathrm{auth}}^{n+1})$. To decrypt the message only the authentication key is needed, which is stored at the log file initialization step. After the first log entry is decrypted, the authentication key to decrypt the second log entry is obtained and so on. The first time the next log entry could not be decrypted shows a manipulation of the log files, which causes by an attacker or a malicious administrator who has tried to blur his traces.

This verification step is to verify the master log and to obtain the verification keys for the daily log. As the entries in the daily log looks like $\mathrm{Enc}_{k_{\mathrm{sym}}}(m^*)$ with the content $m^* = (timestamp, m, k_{\mathrm{sym}}^{\mathrm{new}})$ the first entry could be decrypted by using the symmetric key stored in the master log. The symmetric key for any other entries are in the message payload of the previous log entry. As above in the master log, it is not possible for an attacker to modify any log entry in such a way that the encryption step works correctly.

To verify the application log it is necessary to take a look at Section III as it depends on the method used to protect the application logs. Generally it is necessary to use the secret stored in the daily log. In the following a short security analysis on the approach is shown. This analysis does not regard the application log as the security analysis in the original papers are sufficient.

On the one hand it is not possible for an attacker to gain information from the daily log or the master log as they are both encrypted with well known crypto schemes. On the other hand it is not possible to delete any entry of the log files because during the verification step, the decryption of the entry would fail and the manipulation would become evident. Furthermore, assumed that there is no implementation bug, any used key is deleted immediately so no attacker could restore it.

The only possibility of an attacker is to be fast enough to gain access to the system and to shut the logging mechanism out of service before any log entry is written to the disk. This could happen when the attacker is trying a DoS attack on the system before he is breaking in. Standard implementations of the logging service of the system are able to prevent the system from this method of attack.

VI. Conclusion

Data protection and privacy laws in Europe and several court verdicts in Germany demand the deletion of personal data from log files after a given retention time, which is a use case that is not supported by previous secure logfile management approaches. Motivated by these legal issues and log management requirements in the pan-European SASER-SIEGFRIED project, SLOPPI is a novel cryptographic logging framework that supports the deletion of old log entries without the need to re-calculate message authentication codes and to re-write the remaining log file entries. After outlining the requirements for a secure logging solution, establishing a terminology, and reviewing related work, the SLOPPI framework was presented in this paper along with its primary components, the master log, the daily logs, and the application of cryptographic functions to make the resulting logging solution tamper-evident. Finally, the verification scheme and relevant attack paths were discussed.

In the next step, SLOPPI will be implemented in the SASER project to gain first practical experiences in both setup variante, i. e., with and without a reliable central log server. SLOPPI will also be extended to facilitate the partial deletion of log entries, i. e., only those parts of log entries that contain personal data will be removed after reaching the retention period, whereas the rest of each log entry can be retained for an arbitrary longer time. Besides the removal of personal data, SLOPPI will also be extended to support anonymization and pseudonymization of personal data. The implementation will be made available as open source.

Acknowledgment

Parts of this work has been funded by the German Ministry of Education and Research (FKZ: 16BP12309).

References

[1] W. Hommel, S. Metzger, H. Reiser, and F. von Eye, “Log file management compliance and insider threat detection at higher education institutions,” in *Proceedings of the EUNIS’12 congress*, Oct. 2012, pp. 33–42.

[2] The SASER-SIEGFRIED Project Website. [retrieved: 03.04.13]. [Online]. Available: http://www.celtic-initiative.org/Projects/Celtic-Plus-Projects/2011/SASER/SASER-b-Siegfried/saser-b-default.asp

[3] S. Metzger, W. Hommel, and H. Reiser, “Migration gewachsener Umgebungen auf ein zentrales, datenschutzorientiertes Log-Management-System,” in *Informatik 2011*. Springer, 2011, pp. 1–6, [retrieved: 03.04.13]. [Online]. Available: http://www.user.tu-berlin.de/komm/CD/paper/030322.pdf

[4] M. Bellare and B. S. Yee, “Forward integrity for secure audit logs,” Department of Computer Science and Engineering, University of California at San Diego, Tech. Rep., Nov. 1997.

[5] B. Schneier and J. Kelsey, “Cryptographic support for secure logs on untrusted machines,” in *Proceedings of the 7th conference on USENIX Security Symposium*, vol. 7. Berkeley, CA, USA: USENIX Association, Jan. 1998, pp. 53–62.

[6] J. E. Holt, “Logcrypt: forward security and public verification for secure audit logs,” in *ACSW Frontiers*, ser. CRPIT, R. Buyya, T. Ma, R. Safavi-Naini, C. Steketee, and W. Susilo, Eds., vol. 54. Australian Computer Society, Jan. 2006.

[7] D. Ma and G. Tsudik, “A new approach to secure logging,” in *ACM Transactions on Storage*, vol. 5, no. 1. New York, NY, USA: ACM, Mar. 2009, pp. 2:1–2:21.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-281-3