A Secure Logging Framework with Focus on Compliance

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

***Abstract*—Handling log messages securely, for example, on servers or embedded devices, has often relied on cryptographic messages authentication codes (MACs) to ensure log file integrity: Any modification or deletion of a log entry will invalidate the MAC, making the tampering evident. However, organizational security requirements regarding log files have changed significantly over the decades. For example, European privacy and personal data protection laws mandate that certain information, such as IP (internet protocol) addresses, must only be stored for a certain retention period, typically seven days. Traditional log file security measures, however, do not support the delayed deletion of partial log message information for such compliance reasons. This article presents SLOPPI (secure logging with privacy protection and integrity), a three-tiered log management framework with focus on integrity management and compliance as well as optional support for encryption-based confidentiality of log messages.**

***Keywords*-log file management; secure logging; compliance; log message encryption; privacy by design.**

I. Introduction

For the secure logging, von Eye et al. presented SLOPPI [1] – a framework for secure logging with privacy protection and integrity, which is extended in this article. This framework helps to ensure that the log files, independent of the storage format, fulfill the well-known goals of information technology security:

- The log's *integrity* must be ensured: Neither a malicious administrator nor an attacker, who successfully has compromised a system, shall be able to delete or modify existing, or insert bogus log entries.
- The log shall not violate *compliance* criteria. 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 motivate the presented approach; details are part of previous work [2].
- The *confidentiality* of log entries shall be safeguarded; i. e., read access to log entries shall be confined to an arbitrary set of users.
- The *availability* of log entries shall be made sure of.

The security of log files is a very important aspect in the overall security concept of a service or device. Many attacks or resource abuse cases can be detected by analyzing log files as part of a forensics process, just like system or service breakdowns. With the knowledge embedded in log data, administrators and forensics experts are often able to reconstruct the way an attacker was intruding the system or the root cause of the system disaster.

Because of the concentrated information, the log data is often a primary target of attackers once they have compromised the system. On the one hand, an attacker could erase the whole log file to cover up the traces. This a very efficient way but it also provides a clear information that something went wrong on the system, which most likely will arouse the system administrator's suspicion or trigger an automated alert. When this happens, the administrator is able to detect the attack very fast, even if not every detail can be reconstructed. On the other hand, an attacker could change some of the log entries in a way that the manipulation is not obvious. The approach presented in this article focuses on the second scenario. In any way it would be possible for an attacker to fully delete the log files, which cannot be circumvented as long as the log file resides on a fully compromised machine. Even if there is the possibility to store the log files on an external system, such as a log server, the attacker can be able to disturb the connection between the system and the log server, e. g., by firewalling the connection, once the system has been hacked and the attacker managed to get administrator privileges. But beside this, there are also a lot of systems, which cannot be connected to a central log server, e. g., because of the mobility of the systems or when the organization is too small to operate a dedicated central log server.

This motivates the SLOPPI approach [1], which allows administrators to protect their log files against unwanted changes, while the deletion of log files, e. g., a regular log rotation, is still possible. The SLOPPI architecture allows administrators to have long time logs, while privacy related parts of the log file can be deleted after a well-defined period of time. In this article, the SLOPPI approach, which has substantially been improved since its introduction in [1], is presented. The primary limitation of the previous SLOPPI approach was, that the deletion of log entries also deletes any other information inside this log entry. So, it was not possible to keep parts of the information, e. g., for statistical or diagnostic reasons. In this paper we deal with this drawback and present an improved and more detailed approach, which is now able to keep some predefined parts of the information.

This work is motivated by the large-scale distributed environment of the SASER-SIEGFRIED project (Safe and Secure European Routing) [3], 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 today's IP layer networks in the 2020 timeframe. Thereby, security mechanisms for future networks are designed based on an analysis of the currently predominant security problems on the IP layer, as well as upcoming issues such as vendor-created loopholes and SDN-based (software defined network) 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 log file management applications.

The remainder of this article is structured as follows: Section II introduces the terminology and notation that is used throughout the article. In Section III, the related work and state of the art as well as its influence on the design of SLOPPI are discussed. SLOPPI's architecture and workflows are presented in-depth in Section IV. The process for verifying the integrity of SLOPPI log files is specified in Section V. Before the article's conclusion, Section VI analyzes the security properties of SLOPPI.

II. Terminology

In this article, a few terms and symbols are used to avoid ambiguity. These symbols and terms have the following meaning:

- In the special focus of this work is the untrusted device $\mathcal{U}$, which could be for example a web server or a Linux system. As a matter of its regular operations, $\mathcal{U}$ produces log data, which is saved in one or more log files. As $\mathcal{U}$ is a system, which is not necessarily hardened in any matter, it can be assumed that $\mathcal{U}$ may be compromised by an attacker and therefore, the log data is not guaranteed to be *trustworthy*, i. e., the security goals confidentiality, integrity, and availability cannot reliably be achieved. However, the SLOPPI approach can be used to ensure the integrity and compliance of log data produced by $\mathcal{U}$, making it *reliable* under this specific aspect.
- A trusted machine $\mathcal{T}$. In any related 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, $\mathcal{T}$ is not needed any more as a fully-fledged computer system. To ensure a uniform notation, $\mathcal{T}$ is also used in the following sections in the meaning of a trusted storage for a security key, e. g., a piece of paper that is written on with a pencil. As long as this written paper cannot be read by an outside attacker, it can be assumed as trusted enough. Certainly there are also other solutions, for example, saving the key on a USB memory device (universal serial bus), but in this article the focus is not on the hardening of $\mathcal{T}$ because offline and analogous solutions are sufficient.
- The verifier $\mathcal{V}$. 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 $\mathcal{V}$ does not have 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`)` (Secure Hash Algorithm) or Keccak.
- $\text{HMAC}_k(m)$ (hash-based message authentication code). The message authentication code of the message m using the key k.

SLOPPI does 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, system and data sensitivity, and induced storage overhead.

Furthermore, without loss of generality, the terms *log files*, *log entries*, and *log messages* are discerned as follows:

- A *log file* is an ordered set of *log entries*. The order is implied by the order, in which log messages are received by SLOPPI.
- In line-based logs, a *log entry* normally corresponds to one line of the log file. Otherwise, a log entry consists of all information, which is related to one event. For example, on a typical Linux system, the file `/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. For massively parallel operations, the resulting order is determined by the implementation of a syslog-style system service at run-time based on criteria outside the scope of this article. Log entries cannot be re-ordered once they have been written to a log file in the SLOPPI architecture, which is consistent with related work. Other log file formats, such as the proprietary binary Microsoft Windows event log format, can also be used with SLOPPI; for simplicity, however, all the examples given in this article refer to text-based log files with one log entry per line of text.
- *Log messages* are the payload of *log entries*; typically, log messages are human-readable character 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. Log messages typically have an application-specific structure of their own, which SLOPPI exploits for its slicing technique as detailed below.

As shown in Figure 4, the SLOPPI approach uses several log files, which are related to each other in the following way: The *master log* L_m is the root of the SLOPPI data structure and 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 basically used to minimize the storage needs for the master log L_m, which must never be deleted and therefore shall not contain any personal data or otherwise potentially compliance-offending content; otherwise, no complete verification of the integrity of all log messages 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` generated by an Apache web server or the `firewall.log` created by a local firewall, are restarted from scratch once per day and yield all information generated by the related processes; they are extended by integrity check data. Please note that other intervals than 24 hours are arbitrarily possible, but daily log file rotation are most commonly used and the term is used here for its clarity.

After an arbitrarily specified retention time – seven days in the SASER scenario – these logs, which contain privacy-law protected data, have to be cleaned up or purged completely because of legal constraints in Germany and various other countries. It is important to mention that a simple deletion of whole log files or log entries 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, 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 article. This setup is also used in the SASER project and currently recommended for IT services operated by European providers.

Extending [1], this article presents details about how the *application logs* can be handled with a fine-grained policy that allows to keep as much information as needed arbitrarily longer than the mentioned seven days, while still being compliant. We also discuss log message encryption to ensure confidentiality and how SLOPPI can be used to secure structured log messages, for example, in order to remove substrings from log messages for privacy reasons, to serve as a data source for business intelligence tools, and to facilitate the visualization of security-related log entries.

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 log entries or parts thereof a posteriori. Complementary, the approach summarized in Section III-A does not address the integrity issues.

A. Privacy-enhancing log rotation

Metzger et al. presented an organization-wide concept for privacy-enhancing log rotation in [4]. 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.

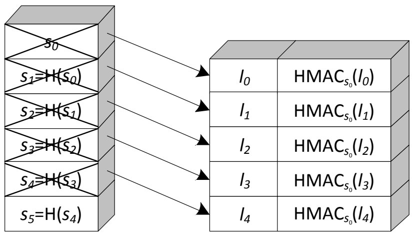

Fig. 1. The basic idea behind Forward Integrity as suggested in [5].

B. Forward integrity

Bellare and Yee introduced the term *Forward Integrity* in [5]. 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 to 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 or modify log entries, the old and already used secret key for the MAC function is erased after the calculation securely. Because of the characteristics of one-way functions, it is not possible for an attacker to derive the previous key backwards in maintainable time. Figure 1 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 subsequent entry 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 developed a cryptographic scheme to secure encrypted log files in [6]. They motivated the approach for encrypting each log entry with the need of confidential logging, e. g., in financial applications. Figure 2 shows the 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 this article 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

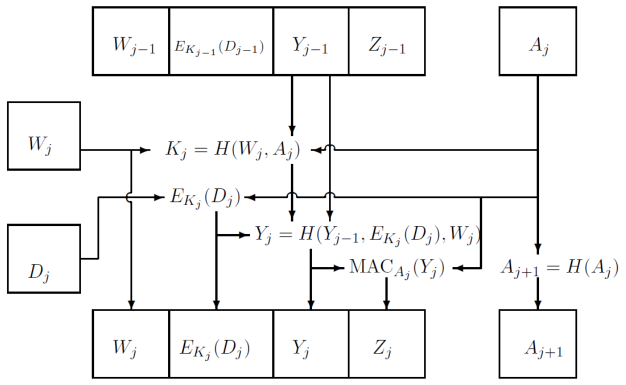

Fig. 2. The Schneier and Kelsey approach taken from [6].

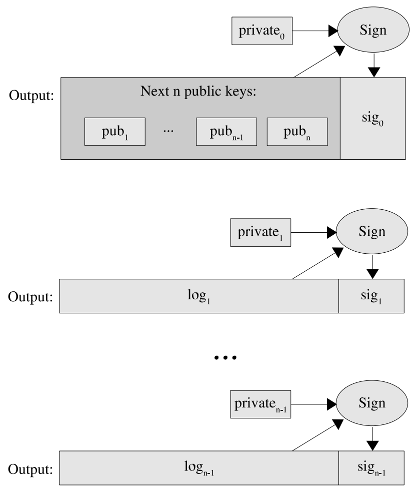

Fig. 3. The Holt approach taken from [7].

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 used a public/private-key-based verification process in his approach to allow a complete disjunction of $\mathcal{T}$ and $\mathcal{V}$ in [7]. 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 $\mathcal{V}$ cannot modify any log entry because it 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 protects each single log entry, does 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 presented a new signature scheme, which aggregates all signatures of the log entries in [8]. 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, yet 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.

F. BAF

Yavuz and Ning specified how log entries can be secured by using blind signatures in [9]. Their approach uses the log entry combined with the actual number of the log entry. For example, if the log entry D_n is the nth entry in the log file, D_n is combined with the number n to prevent an attacker from reordering the log entries. This result is hashed and modified with the secret key (a, b) by using a simple addition and multiplication modulo a large prime p. As in all other approaches, the secret key is updated and the previous version securely deleted.

The most interesting result of this approach is that a verification is possible for a verifier $\mathcal{V}$, while it is not possible for $\mathcal{V}$ to modify any entry in the log file. This property is normally only satisfied by public/private key schemes, which are typically very expensive to compute.

IV. The SLOPPI architecture and workflows

SLOPPI, as presented in [1], follows the classic client-server architecture of well-known POSIX (Portable Operating System Interface) logging mechanisms, such as syslog-ng and rsyslog. Any SLOPPI implementation therefore is a continuously running process, which offers interfaces, such as an application programming interface (API) and IPC (IP code), TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) sockets, to receive new log messages from various system services, applications, or remote servers. After internal processing, log entries are stored in plain-text files in the local file system, where they can be processed with whichever log file viewing mechanism the local users are familiar with; alternatively, log entries can be forwarded to remote SLOPPI servers where they are treated in the same manner. If the application log files make use of the optional encryption, SLOPPI tools can be used to decrypt the log entries using standard input and output channels, typically in combination with POSIX pipes. Similar tools can be used to strip any SLOPPI-specific information from log entries so any other log file processing tools can be used for parsing and processing the application log files even if they are not aware of the extensions brought by the SLOPPI data format.

As SLOPPI has been designed with compliance regulations as its primary motivation, ensuring integrity and allowing for log-file rotation without cryptographic re-keying are its core functionality. In the following sections, first described is how the master log, the daily log, and the application log are intertwined to achieve these properties. Afterwards follows a discussion of the various optional functionalities for the application logs, e. g., the encryption of application log messages.

A. The SLOPPI log file hierarchy

The analysis of the related work shows that there is no solution yet that fulfills both necessary minimum requirements for log files: *integrity* and *compliance*, where the latter requires making changes to integrity-checked log entries once they have reached a certain age. The SLOPPI approach combines key operations from 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. They form the following hierarchy as shown in Figure 4:

- The master log file is the root of the SLOPPI data format. It is created only once and must not be deleted. If it is deleted, e. g., by an attacker, integrity checking is no longer possible.
- The daily log files are, as implied by their name, created in a daily manner. Although other rollover periods could be used, such as hourly or weekly, we refer to them as daily logs for the sake of simplicity and because they are a de-facto industry standard. Daily log files can be deleted after a retention period, for which 7 days is the standard setting; it is, however, recommended to delete the affected application log files first.
- Application log files are the only log files, in which actual payload log messages are stored – both the master log and the daily logs only contain SLOPPI-specific meta-information. There can be an arbitrary number of application log files depending on how many files all the log information should be scattered across. SLOPPI supports the typical syslog-like distribution of log messages to log files based on the originating host, application, log level, and log message content in an administrator-configured manner. While storing a log message in exactly one log file is the usual mode of operation, the same log message can be logged to multiple log files if desired, or thrown away without being written to a file, and therefore without influencing the integrity mechanism. As an alternative to local log files, log entries can also be forwarded to remote SLOPPI services, which treat them similarly to locally logged messages; communication is secured using TLS (Transport Layer Security) connections.

At the very core of SLOPPI, the master log file L_m has to be secured. As it has only a few entries per day, it can be protected using a public key scheme, e. g., RSA, to protect the log entries, which is described in detail in the upcoming Section IV-B. In the subsequent sections, the two keys of a public key scheme are called *signing key* (k_{sign}) and *authentication key* (k_{auth}).

Then the daily log file L_d is considered in Section IV-C. 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 certainly the best choice. Finally, application log file details and options are presented in Section IV-D.

B. The SLOPPI Master Log File

As stated above, the master log file L_m contains important data of the SLOPPI approach to protect the integrity of the log files. To protect L_m, the following steps are necessary:

1) Log Initialization: Whenever a new master log is initialized, $\mathcal{U}$ generates an authentication key (k^1_{auth}) and a signing key (k^1_{sign}). These two keys are important to protect (using k^1_{sign}) now and to verify (using k^1_{auth}) the log file later. As the verification key should not be stored on $\mathcal{U}$, it is sent to $\mathcal{T}$ over a secure connection, e. g., a TLS connection. As mentioned above, it is not necessary that $\mathcal{T}$ is a computer system as k^1_{auth} could also written on a piece of paper by the administrator. But mostly it could be assumed that $\mathcal{T}$ is a specially secured and encrypted database. After sending the authentication key, $\mathcal{U}$ deletes (k^1_{auth}) securely.

$\mathcal{U}$ can now initialize the master log file by saving the first message `STARTING LOG FILE` in the log file as described next. For this step, k^1_{sign} is the actually used secret. Important is that the master log is normally generated only once per SLOPPI instance.

2) Saving New Log Entries: Let m be the log message of the log entry to be stored in the log file. As the master log file has only one entry per day, it can be assumed that there is enough time between saving the last entry and the actual one to generate a new authentication/signing key pair ($k^{n+1}_{\text{auth}}, k^{n+1}_{\text{sign}}$), while k^n_{sign} is the actual secret.

$\mathcal{U}$ now generates

$$m^* = (\textit{timestamp,}m, k^{n+1}_{\text{auth}})$$

and computes

$$e = \text{Enc}_{k^n_{\text{sign}}}(m^*).$$

The result e is the new log entry, which is written to the log file. Immediately after calculating the encrypted result, the keys k^n_{sign}, k^{n+1}_{auth}, which are not needed anymore, are erased securely from the system. Now the master log file only contains fully encrypted data and k^{n+1}_{sign} is the secret for the next log entry. The motivation for the data format used for m^* is the following:

- The *timestamp* is used to verify the time, at which a new event is logged in the master log. An abnormal high or low rate of entries in a specific time interval can indicate a system failure or an attack. To prevent any changes of the timestamp, this is also part of the encrypted data. item

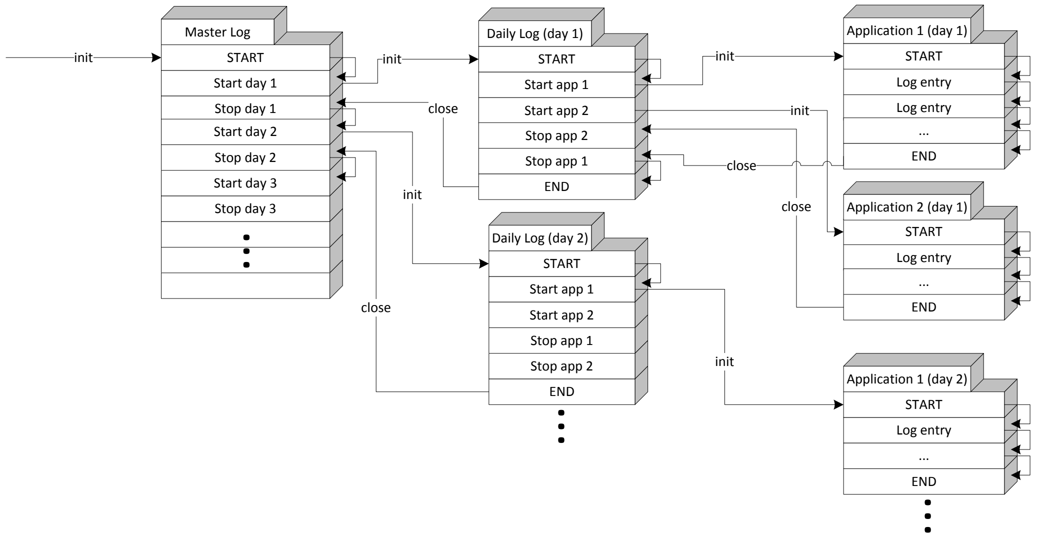

Fig. 4. Overview of all relevant log files.

The *log message* m has to be encrypted, to protect the main content of the log entry.

- k^{n+1}_{auth} is the *verification key* for the next log entry. This is the application of the forward integrity approach, because the necessary information to decrypt and to verify the next step are all available if the previous verification and decryption step is completed. As this is data very worthy of protection, it is naturally part of the encrypted data.

For example, let the first log message be

$$m = \texttt{STARTING LOG FILE}$$

and the actual secret k^1_{sign}. After generating $(k^2_{\text{auth}}, k^2_{\text{sign}})$, $\mathcal{U}$ now composes

$$m^* = (1408096800, \texttt{STARTING LOG FILE}, k^2_{\text{auth}})$$

and computes

$$e = \text{Enc}_{k^1_{\text{sign}}}(m^*)$$

$$= \text{Enc}_{k^1_{\text{sign}}}(1408096800, \texttt{STARTING LOG FILE}, k^2_{\text{auth}}),$$

which is written to the log file. Now, k^1_{sign} and k^2_{auth} are deleted securely.

3) Closing the Log File: During regular SLOPPI operation, there should not be the need to close the master log file. But in case of a system restart, a serious failure, or in the case where the storage requirement of the master log is too much increasing, there can be the desire to restart the master log.

If the master log file has to be closed gracefully, 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 therefore, the next authentication key is also irrelevant. To fulfill the data format defined above, it is needed that the log entry consists the next authentication key, which is set to an empty string.

4) Content of Log Messages: As L_m is used as a meta log, which does not contain any application or system messages, the content of the log messages m are now specified. 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 the variable p_1. Furthermore, the variable p_2 is the first entry in L_d and finally the variable 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/2014-08-15.log;STARTING LOG FILE;VerySecretKey` together with $H(p_1, p_2, p_3)$.

Because of the need to detect manipulations of 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 exactly.

C. The SLOPPI Daily Log File

The main reason to use the daily log is to reduce the storage 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 much more than one application log used on a system.

1) Log Initialization: Every day a new daily log has to be initialized if a daily log rotation is configured on the system. Otherwise, another initialization interval is used for starting a new daily log. At the beginning, $\mathcal{U}$ generates a symmetric key k_{sym} which is necessary for both, encryption and verification. This key is the initial secret and has to stored in a trusted space, e. g., on $\mathcal{T}$. As the SLOPPI approach does not need a separate $\mathcal{T}$, the already existing L_m is used as a trusted third party. As described above, it is unlikely that an attacker can get information out of L_m because this log file is fully encrypted. If the attacker has already compromised the system, no new log message can be trusted any longer and the attacker is able to modify any computation step. Therefore, it is secure to store the key inside of L_m. The key k_{sym} is now written together with the name and the path of the actual log file. This information is combined with the first message `STARTING LOG FILE`.

The actually used secret is now k_{sym}. $\mathcal{U}$ can then initialize the daily log file by saving the first message `STARTING LOG FILE` in the log file as described next.

2) Saving New Log Entries: To encrypt the important information of the daily log, a symmetric key scheme is used, e. g., `AES` (Advanced Encryption Standard). In difference to the master log, not all of the information stored in L_d can be encrypted, because parts are needed in plain text during later steps as detailed below.

Let (m, m') be the message that has to be stored in the log entry and $k_{\mathrm{sym}}^{\mathrm{old}}$ the actually used secret key. The precise meaning of (m, m') is defined below along with the log message content.

$\mathcal{U}$ now randomly chooses a new secret key $k_{\mathrm{sym}}^{\mathrm{new}}$. Because of the use of symmetric key schemes, this step is not computationally expensive. Analogous to the master log processing, an expended message is now generated by $\mathcal{U}$, which has the structure

$$m^* = (timestamp, m, k_{\mathrm{sym}}^{\mathrm{new}}, H(m', (timestamp, m, k_{\mathrm{sym}}^{\mathrm{new}})))$$

. With this, the log entry

$$e = (m', \mathrm{Enc}_{k_{\mathrm{sym}}^{\mathrm{old}}}(m^*))$$

can be computed, which is written to the log file. The hash value is stored for verification purposes, so it is possible to detect the exact log entry where a manipulation took place.

Immediately after calculating the encrypted result, the key $k_{\mathrm{sym}}^{\mathrm{old}}$, which is not needed anymore, is erased securely from the system. Now $k_{\mathrm{sym}}^{\mathrm{new}}$ is the secret for the next log entry. The motivation for the data format used for m^* is the following:

- The *timestamp* is used to verify the time when a new event is logged in the daily log. An abnormal high or low rate of entries in a specific time interval can indicate a system failure or an attack. To prevent any changes of the timestamp, this is also part of the encrypted data.
- The *log message* m has to be trivially encrypted, to protect the main content of the log entry. Besides m, there is also a part of information, which has to remain in plain text to use them in the typical usage of the SLOPPI tools.
- $k_{\mathrm{sym}}^{\mathrm{new}}$ is the *verification key* for the next log entry. This is the application of the forward integrity approach, because the necessary information to decrypt and to verify the next step are all available if the previous verification and decryption step is completed. As this is a data very worthy of protection, it again is naturally part of the encrypted data.

3) Closing the Log File: If the daily log file has to be closed, the last message `CLOSING LOG FILE` is saved. It is important that in this case it is not necessary to generate a new key pair and therefore, the next authentication key is also irrelevant. To fulfill the data format defined above, it is required that the log entry consists the next authentication key, which is set to an empty string. This message, the file name and location, the MAC of the entire log file, and the last generated key are committed to be stored in the master log.

4) Content of Log Messages: In the daily log, there are five types of messages. In difference to the master log, which contains only meta information, which is only interesting for verification purpose, the daily log also contains information, which is used in the daily use of the system. Therefore, these parts of the contained information of a message have to remain unencrypted. In the following, '_' means an empty string of zero bytes size.

- `STARTING LOG FILE`. This message only consists of the string, which should be encrypted.

$$(m, m') = (\texttt{STARTING LOG FILE}, '\_').$$

- `CLOSING LOG FILE`. This message only consists of the string, which should be encrypted.

$$(m, m') = (\texttt{CLOSING LOG FILE}, '\_').$$

- `START APPLICATION LOG`. This message contains the timestamp when the application log was initialized (this is in most cases the same timestamp, which is used above for saving the log file), the file name, and location of the application log. This information is saved in plain text inside the daily log because it is needed to identify the application logs, which are connected to the specific daily log.
  Furthermore, the message consists of the initialization key, the first message, and the file name and location of the application log in encrypted form. The encryption step happens when the log entry is being saved to the daily log. The initialization key of the application log is saved in the daily log because the daily log is the trusted third party $\mathcal{T}$ for the application log.
  This leads to $m =$ (`START APPLICATION LOG`, initialization key, first message, file name, location) and $m' =$ (`START APPLICATION LOG`, timestamp, file name, location).
- `STOP APPLICATION LOG`. Similar to the start message, this message contains in plain text a timestamp, the file name, and the location of the application log. The encrypted parts, which are also encrypted when saving the log entry, are the last key, the last message, and the file name as well as the location of the application log.
  This leads to $m =$ (`STOP APPLICATION LOG`, last key, last message, file name, location) and $m' =$ (`STOP APPLICATION LOG`, timestamp, file name, location).

- `ROTATING APPLICATION LOG`. In case of a log rotation procedure, it is necessary that the SLOPPI tool is able to know, which new log file was previously protected by the SLOPPI approach and which daily log this new log file is assigned to. For this reason, the message contains a timestamp and both file names, i. e., before and after a rotation, in plain text. As in the previous message type, there are – also for security reasons – the file names and locations as parts of the cipher text.
  This leads to $m =$ (`ROTATING APPLICATION LOG`, file name before, file name after) and $m' =$ (`ROTATING APPLICATION LOG`, timestamp, file name before, file name after).

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.

D. The SLOPPI Application Log Files

In general, the application log files L_a can be protected by any approach presented in Section III and are not a mandatory part of the SLOPPI architecture. 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.

SLOPPI's core components ensure that the essential requirements of secure logging are fulfilled: integrity and compliance. However, in high-security environments, additional characteristics are required, such as the confidentiality of application log entries with selective deletion of personal data. In this section, we present modular extensions to the SLOPPI architecture and workflows to provide such additional functionality. Due to the openness of SLOPPI's architecture, arbitrary other modules can also be implemented.

1) Application Log Message Encryption: In many cases application log files should be in plain text to allow other applications or human administrators to take a look into the log files. However, in other cases it is appropriate to protect the content of the log files from prying eyes.

In the basic SLOPPI approach, it is not required to encrypt the log entries in the application log files explicitly. But sometimes it becomes necessary that the application logs get encrypted. For this, SLOPPI provides an extension, which supports different encryption methods for the most common scenarios as discussed below.

In general, any existing encryption algorithm may be used for securing particular application logs. So, here the encryption extension for SLOPPI is specified in an abstract way assuming an encryption method using particular, respective secrets k_{sign} (encryption) and k_{auth} (decryption) to secure a particular application log L_a. For sure, $k_{\text{sign}} = k_{\text{auth}}$ if symmetric encryption scheme is used, which is generally recommended due to the amount of data that is typically logged to application logs.

Based on this assumption, there are still several different options how to actually encrypt the respective application log messages. These options must be bases upon the underlying scenario; the most common ones are discussed in the following:

- Sometimes different persons or groups are responsible for the administration of a system or service. As this sharing of responsibility mostly leads to more than one application logs, in which the different groups are divided, it is important to define, whether all application log files using the same key pair for encryption or different ones for each log file are chosen.
- Log messages in application logs can be very critical. For example, in a bank it is possible that each money transfer is also logged into the application log to prove that the transfer succeeded or failed. Assume that in this case it is insufficient if the log message is readable even for a short period of time. So here is the requirement that each log message is encrypted with a key, which is different from the previous message as well as from the following message.
- As SLOPPI should be runnable on nearly every system, it is sometimes difficult to generate a new randomly chosen key every time (especially in the case "one message, one key"). For this scenario, SLOPPI also supports an auto-derivation of the encryption key by using an appropriate key derivation function. This function can, for example, be based on Keccak or other algorithms that also support the generation of arbitrary-length key material.
- In general, log file analysis should be done nearly in real time to detect unusual events. For this it is not necessary to take a look into the log file after a predefined period of time, e. g., after the daily log rotation. If this is the case, it is sufficient to analyze the log files, while the verification process is running and so the log file encryption can start, for example, with a randomly chosen key pair. Otherwise, the verification and the log file analysis, i. e., the log file decryption, are different processes. In that case, the decryption key has to be deduced by applying an appropriate key derivation function. It is obvious that in this case the initial secret has to be stored on $\mathcal{T}$ otherwise,

decryption would also possible for any attacker.

The encryption extension for SLOPPI supports any combination of the described options above. Which options are chosen in a specific scenario depends on the characteristics of the application log file and environment used, e. g., the number of log lines per day, the speed of adding of log lines in maximum and in average, the computing power and memory resources of the system versus the confidentiality in the face of potential attackers for a day-period.

Even if the single key-pair option, i. e., for one or more log files one key per day, is chosen, a potential attacker will not be able to read any application log files of prior days, but he may be able to read the log file of the current day, as the single key pair is still accessible.

In any case, all necessary information about the choices made has to be stored initially in the respective start application log message in the daily log file. Therefore, the content of the startup application log message has to be extended in comparison to basic SLOPPI:

- $m =$ (`START APPLICATION LOG`, initialization key, first message, file name, location) has to be changed to $m =$ (`START APPLICATION LOG`, initialization key, encryption method, initial key or derivation info, iteration used?, chaining used?, first message, file name, location), where *encryption method* gives some information about the used encryption algorithm and also, which of the extensions above is used. *initial key or derivation info* is the place, where the used (initial) decryption key is stored. If this key is computable from previous information, the key information is replaced by information about the used key derivation function. The last two optional parts *iteration used* and *chaining used* define information about parts of the extensions described above.
- $m' =$ (`START APPLICATION LOG`, timestamp, file name, location) does not require any changes.

All application log messages of the respective application log file have to be encrypted according to the choices made. Additionally, if the *chaining* mode was chosen, each newly selected decryption key for the next application log entry has to be concatenated with the proper message of the current application log entry before this concatenation is encrypted using the current encryption key. This leads to the following:

- $applogentry_i = E_i(decryptkey_{i+1}; msg_i)$ if chaining is used
- $applogentry_i = E_i(msg_i)$ otherwise

2) Application Log Message Enrichment: Log messages sent to SLOPPI can be parsed and enriched before being stored in an application log file. This brings the following benefits:

- The application-specific structure of log messages can be made explicit, for example, by marking data fields containing personal data or specific error codes. This simplifies processing SLOPPI application log files in log management tools or business intelligence software, such as Splunk.
- Recommendations on how the content of the log message should be visualized can be added.

Given log message m of length n, $c_1, \ldots, c_n$ denotes the individual characters that m is composed of. A *slice* is defined as tuple $\langle c_i, c_j \rangle$ with $i \leq j$ and denotes the boundaries of an arbitrary non-empty sub-string of m. Unique names are assigned to slices and can be used to describe their semantics. For example, if a network service's log message contains the IP address of a client, the resulting slice could be named *ipaddress*. The exact boundaries for each slice in any log message are determined by parsing rules. Typically, regular expressions will be used to parse a log message in order to determine the slice boundaries. If a parsing rule matches the given log message, the slice name and boundaries are appended to the log entry. For example, if the IPv4 address 10.31.33.7 is detected in a log message starting at c_i with $i = 27$, then the string `@ipaddress:27-36` is appended to the log entry.

In general, slices can be overlapping. For example, a whole log message may be slice-tagged as *service x's error message*, while a sub-string may be tagged *ipaddress*. However, for the personal data anonymization procedure discussed below, care must be taken that the two types of used slices do not overlap to ensure that one slice's hash value does not change when another slice is being modified. Otherwise, the a-posteriori anonymization of log files would break the verification procedure.

Slices are useful for two slightly different use cases: On the one hand, the SLOPPI tools can be instructed to verify and decrypt only selected slices; this allows for a fine-grained access model where system administrators can be restricted to which parts of single log file entries they are allowed to read – this allows for a more detailed access management than the traditional per-file or per-log-entry model found in most of today's implementations. On the other hand, slice names and ranges can be fed to other log entry processing tools, such as business intelligence software and log file visualization tools. This not only saves the overhead of parsing the same log message multiple times in different processing tools, but its true power lies in that the slicing can be integrity-checked. For example, it becomes obvious whether an IPv4 address has been recognized as such by looking at the slice names; compliance violations, such as not checking which parts of log messages contain personal data that must be removed after the maximum retention time become easy to spot for an auditor. Also, administrators do not need to wait until the maximum retention time has been reached to verify whether an anonymization batch run will modify the correct parts of the log messages.

E. Generalized Privacy Protection for the Application Logs

As introduced in the previous section, SLOPPI also provides a semantic tagging of log entries. This can be used to identify privacy-relevant data. With this knowledge it is possible to anonymize these relevant parts of the log file, while the remainder of the information can be stored untouched.

In general a log message is only a string without any semantics for the normal logging process. Because of that the normal logging process is not able to differentiate privacy relevant data. With SLOPPI and the application log message enrichment extension, it is possible to give the logging process the needed information to separate compliance sensitive data from other text. As the goal is to enable the a-posteriori anonymization of log files, the content of each log message is split into two categories:

- The part of the log, which has to be anonymized later on.
- Everything else.

To perform the seperation between anonymized and non-anonymized data, there has to defined a priori the exact log format. For example, in the SSH (Secure Shell) log files, the log entries are always in a uniformed format, e. g., on a standard Debian web server there is always at the beginning a timestamp, followed by the host name, the process name and number and finally the log message, which contains the authentication method, the user name and finally the source IP address and the port. In this example, only the username and the IP address is necessary to anonymize for fulfilling the legislation boundaries. With the help of regular expressions, the positions of this information can be found. In other examples, this might be more complicated, but in general each logging source has its own specified output format. If there are to many mixed entries in a log file, there is always the possibility to split this log file into more than one others.

As written above, for the protection of the application log any approach of Section III can be used. In the following the forward integrity approach is described; any other approach can be handled analogous.

For the integrity protection, a MAC of each log entries is used. This MAC would change if an anonymization is performed afterwards and a verification would be impossible. But with the knowledge of the relevant parts, it is possible to calculate two MACs. The first is the original MAC with all information, while the other MAC is calculated with already anonymized parts of the log message.

These two MACs are appended accordingly to the log entry. The verification process can now check both MAC values. If no anonymization has been performed yet, the original MAC can be used. Otherwise, the verification step uses the new MAC. For this it is important that the SLOPPI anonymization uses the same anonymization string every time. To ensure human readability, i. e., administrators should know, which parts of a log message have already been anonymized, a placeholder replacement string, such as XXX or $***$ should be used. For simple implementation, it needs to be of constant length; this also avoids issues regarding the de-anonymization success probability when using variable-length strings.

In general it is important to know, which log entry has been anonymized at what time. Because an attacker could otherwise, anonymize any log entry himself to cover up the traces. To prevent this, SLOPPI generates a new log entry in the daily log, which contains the message LOG ANONYMIZATION and also the information, about which log file and which log entries are being anonymized, e. g., all log entries between the timestamps 1408010400 and 1408096800. To avoid race conditions, this log entry is created after an anonymization batch run has successfully finished.

F. Generalized Privacy Protection for the Application Logs with Partial Encryption

Both extensions of sections IV-D1 and IV-E can be combined to allow for on the one hand the partial deletion of log entries, e. g., to support privacy for relevant data, and on the other hand for partial encryption of log entries, simultaneously. Assumed is an partial application log file L_a and its application log messages being partitioned into different slices (i. e., tagged parts):

- Different slices of application log messages are – either iteratively or in chained mode – to be hashed separately as described in Section IV-E; this is done using multiple initial hash secrets, which are being stored in the respective start application log message for the application log file in the daily log file.
- Particular slices of application log messages can in addition be iteratively encrypted similar to IV-D1, but only for the particular slice. Each slice is treated separately with different encryption methods and secrets. Therefore, the method identifier needs to be stored along with the initial description key for each slice in the respective application log message for the application log file in the daily log file.

In sum, this requires the slicing of log messages along with partial encryption of each slice and therefore adds some overhead to the application and daily log files.

V. SLOPPI Log Entry Verification Procedure

To verify log entries, the initial master key is needed. Each log entry in the master log is encrypted as $\mathrm{Enc}_{k^n_{\mathrm{sign}}}(m^*)$ with $m^* = (timestamp, m, k^{n+1}_{\mathrm{auth}})$. To decrypt the message only the authentication key is needed, which is stored during the log file initialization step. After the first log entry is decrypted, the authentication key to decrypt the second log entry is obtained implicitly, and so on. The first occurrance of a log entry, which cannot be decrypted gives proof that a manipulation of the log files, which has been caused 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 look like $\mathrm{Enc}_{k_{\mathrm{sym}}}(m^*)$ with the content $m^* = (timestamp, m, k^{\mathrm{new}}_{\mathrm{sym}})$, the first entry could be decrypted by using the symmetric key stored in the master log. The symmetric key for any other entries is 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.

A. Master Log: Verification of Log Messages

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 literal `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 subsequent log entries of L_m. Regularly the master log file is not closed because it is intended that the master log runs endlessly. Therefore, the last timestamp has to be near the current timestamp, but the derivation depends on the daily log scenario.

In the case of a failure, e. g., in the storage device, it is possible that writing in the master log file is interrupted suddenly. In this case, the last message of the master log is not the `CLOSING LOG FILE` message and probably has a somewhat out-dated timestamp, depending on the daily log scenario. In this case, verification is only possible up until the last timestamp just as if the master log had been closed regularly and no new master log had been started.

B. Daily Log Verification of Log Messages

Because of the need to detect any manipulation of L_m, it is necessary that e also contains a hash value of the message. With the knowledge of this hash value it is possible to detect, where the decryption process failed. To verify an existing daily 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 in turn 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 application log files, it is possible to verify any application log that is still available. If an application log has already been deleted, then this application log cannot be verified any more, but it is still possible to use the subsequent log entries of L_d.

In the case of a failure, e. g., in the storage device, it is possible that writing in the daily log file is interrupted suddenly. In this case, the last message of the daily log is not the expected `CLOSING LOG FILE` message; it may also have an out-dated timestamp. In this case, verification is only possible until the last timestamp just as the daily log has been closed regularly and no new daily log has been started.

Furthermore, the transfer of the data between the daily log and the master log can fail. In this case the master log doesn't get the information that a daily log was initialized or was closed. This leads to a point where the verification process fails because the last secret keys are already deleted and so it is impossible to recover them to verify the last log entry.

C. Application Log: Verification of Log Messages

The procedure for verifying application log entries heavily depends on the options chosen for encryption. Basically, the key material that is required for beginning with the verification is part of the daily log. If the application log is not encrypted, the verification is concerned only with checking the application log's integrity by calculating the hash values given the current application log file line by line, and comparing the result calculated during the verification process with the values stored in the log file. Any mismatch is an indicator that the system has been compromised after the timestamp of the previous entry, i. e., the one before the offending line.

If the application log is additionally encrypted, there are two verification methods. On the one hand, full verification requires decrypting the application log file line by line as the decryption key material required for the next line is either based on the current line's material (iterative mode) or stored encrypted in the current log entry (chaining mode). An application log file's full verification is therefore $O(n)$ with n being the total number of lines in an application log file. Full verification is successful when all hash values match and the last line decrypts to yield the end application log line. On the other hand, the partial, or basic, verification omits any decryption steps. In other words, it just ensures that the integrity-protecting hash values are correct. The advantage of this approach is that basic verification can be performed without supplying the decryption key. It therefore can be used with automatisms not trusted enough to be provided with the decryption password.

VI. SLOPPI Security Analysis

In the SLOPPI approach, the method authenticate-then-encrypt is used to secure for example the daily log, as the the message m is in a first step authenticated by a hash function H and then both (m and $H(m)$) are encrypted. As Krawczyk discovered, there are some problems by using authenticate-then-encrypt, which makes them vulnerable against chosen plaintext attacks [10]. But on the other side, by choosing the right encryption methods (e. g., CBC (Cipher Block Chaining) mode or stream ciphers) also the authenticate-then-encrypt method is secure. This security consideration has to be taken into account, if the SLOPPI framework is implemented.

Furthermore, it is possible to relinquish most of the calculated MACs by using authenticated encryption with associated data cipher modes [11]. By using these cipher modes, only the encryption step is necessary to perform to gain encryption and also authentication of the messages. On the other side it has to be evaluated, if the performance and memory needs of these ciphers are comparable to traditional cryptographic schemes.

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, assuming proper implementation, any used key is removed securely from memory 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 down before any log entry is written to the disk. This may eventually happen, e. g., when the attacker performs a DoS attack on the system before he is breaking in. However, implementations of the logging service of the system are able to prevent the system from this method of attack by aggregating multiple identical events before writing them to application log files; the same approach can be used for future SLOPPI implementations.

The confidentiality of log messages is related to the feasibility of unauthorized decryption. For SLOPPI, the following aspects need to be considered:

- The master log file is strongly encrypted in any of the operational modes described above, i. e., for the iterative mode as well as the chaining mode, which is applied to each individual application log file. The master log file uses public/private key encryption and therefore secure as long as either of both keys is kept out of an attacker's reach.
- The daily log file is also strongly encrypted in any of the operational modes. However, it uses a symmetric encryption scheme, which means its security relies on generating high-quality random key material, which an attacker must never see. Generating key material, e. g., by using pseudo-random number generators (PRNGs), typically works well on systems with good entropy pools, such as networked servers or interactively used machines. However, especially embedded systems without entropy-generating sensors may be bad at generating new key material. In this case, manually planting an unique random number seed that provides enough input for further key derivation during the system's estimated run-time before system deployment is mandatory.
- In the basic SLOPPI operating mode, only iteratively securely hashed key material is used. The initial hash secret is saved in the respective opening daily log message in the daily log. Therefore, the confidentiality of the daily log file, which has been discussed above, is sufficient to ensure the secrecy of this keying material.
- When SLOPPI is used to also encrypt log messages in the application log files, these log files make use of the same iterative or chained encryption:
  - The initial decryption key $k_{m,d,1}$ is being stored along with its initial hash secret in the respective daily log message about opening the application log; the confidentiality of the daily log protects this key material.
  - With each log entry $i, i \geq 1$, in the application log, the entry meta-data also contains the next iteration of the decryption key $k_{m,d,i+1}$.

When using SLOPPI with application log file encryption, using either iterative or chained key material generation deserves further discussion. The basis for both options is the generation of new key material in the beginning; it depends on the quality of the PRNG, similar to the daily log, which has been discussed above. Furthermore, key material that is no longer needed needs to be correctly and irrevocably be erased from memory, i. e., after the first log message iteration or after the application log file has been closed. The confidentiality also depends on the strength and quality of the used key derivation algorithm and its mode of operation, e. g., counter mode, feedback mode, or double-pipeline iteration mode. Similar to encryption, many options are available and SLOPPI does not prescribe, which one to use.

If SLOPPI is used without any iteration re-keying option, then an attacker may be able to read plain text messages that are already written to the application log file before the attacker gained control of the system, but limited to the particular application log file of the current day. For any other, already closed application log files, i. e., those from earlier days, the used key material should already have been erased irrevocably from memory.

If the auto-iteration SLOPPI mode is used, which is based on key-derivation functions to generate new key material during re-keying, and the proper removal of no longer needed key material from memory is ensured, then the attacker is unable to access the plain text of the written application log messages up until the current re-keying at the time of system compromise.

Finally, if the chaining SLOPPI mode is used, i. e., new key material is generated for each application log entry, the quality of the achieved protection again is based on the quality of the used PRNG and proper reliable removal of outdated key material from memory. Attackers cannot gain access to previously written log entries' plain text either. Depending on the cipher that is used, ideally both options only differ in the amount of new key material that needs to be generated.

As the application log files can be deleted independent of their encryption status – which is required to fulfill compliance policies regarding retention time – without violating their integrity status because the actual content of the application log message is irrelevant for an integrity check, the mode of encryption does not interfere with regular SLOPPI operations.

It must not be neglected that SLOPPI does not address the availability issue of log files. Attackers can remove traces of attacks by simply deleting log files. Although this leads to an alarming situation, i. e., administrators can be informed about missing, truncated, or integrity-violating files, SLOPPI does not guarantee full traceability of all logged messages. Other techniques, such as using write-only memory, would be needed as this problem is all but impossible to fix in software realistically, given today's operating systems and their potential vulnerabilities, i. e., even a write-once file system could be accessed in a reading fashion, e. g., after system reboot and re-parametrization by an attacker who gained administrative privileges.

VII. Conclusion

SLOPPI is a log file management framework that supports integrity-checking and confidentiality through encryption like other approaches, but has a special focus on compliance. Compliance with privacy and data protection acts can, at least in Europe, only be achieved by limiting the retention time of personal data, which may also be a part of log messages written to log files. SLOPPI therefore supports the fine-grained partial removal of log entries, while still ensuring the properly working integrity-monitoring of the other logged data. For example, German Internet service providers and any other providers of Internet-based services are obliged to remove personal data from log files after 7 days based on court verdicts that have become effective. Using the SLOPPI approach, log files older than the maximum personal data retention time can be modified to have all personal data removed, while still ensuring that the log file has not been tampered with otherwise.

In this article, we have presented the inner workings and security analysis of SLOPPI in more detail than our previous work [1] and specified two extensions to the original SLOPPI architecture. First, we introduced the concept of slicing log messages and use them for semantic tagging, which can also be used in conjunction with external tools, such as business intelligence applications, for a-posteriori removal of personal data from log entries, and applying encryption for fine-grained access management protecting read access to log file information. Second, we presented the two options of iterations and chaining to address re-keying for encryption application log files, which were not part of the original SLOPPI specification.

To this extent, SLOPPI has several important functional advantages over the previously state-of-the-art logging mechanisms. However, they come with the inherent overhead of additional cryptographic operations and, to a large extent, depend on the quality of random numbers generated for constructing key material, which can be a problem for low-interaction systems without sufficient entropy-generating pools. As future work on SLOPPI it is therefore planned to support adaptive protection mechanisms that vary the strength of the used key material depending on the sensitivity of the data it is applied to. For example, stronger cryptography could be applied to log message slices containing personal data, while trivial log entries are only protected using weaker mechanisms.

Acknowledgment

Parts of this work has been funded by the German Ministry of Education and Research (FKZ: 16BP12309). The authors wish to thank the members of the Munich Network Management (MNM) Team for helpful comments on previous versions of this article. The MNM-Team, directed by Prof. Dr. Dieter Kranzlmüller and Prof. Dr. Heinz-Gerd Hegering, is a group of researchers at Ludwig-Maximilians-Universität München, Technische Universität München, the Universität der Bundeswehr München, and the Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities.

References

[1] F. von Eye, D. Schmitz, and W. Hommel, "SLOPPI – a framework for secure logging with privacy protection and integrity," in *ICIMP 2013, The Eighth International Conference on Internet Monitoring and Protection*, W. Dougherty and P. Dini, Eds. Roma, Italia: IARIA, Jun. 2013, pp. 14–19. [Online]. Available: http://www.thinkmind.org/download.php?articleid=icimp_2013_1_30_30021 [accessed: 2014-12-01]

[2] 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.

[3] M. Hoffmann, "The SASER-SIEGFRIED project website." [Online]. Available: http://www.celtic-initiative.org/Projects/Celtic-Plus-Projects/2011/SASER/SASER-b-Siegfried/saser-b-default.asp [accessed: 2014-12-01]

[4] 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. [Online]. Available: http://www.user.tu-berlin.de/komm/CD/paper/030322.pdf [accessed: 2014-12-01]

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. A. Yavuz and P. Ning, "BAF: an efficient publicly verifiable secure audit logging scheme for distributed systems," in *ACSAC*, 2009, pp. 219–228.

[10] H. Krawczyk, "The order of encryption and authentication for protecting communications (or: How secure is ssl?)," in *Advances in Cryptology CRYPTO 2001*, ser. Lecture Notes in Computer Science, J. Kilian, Ed. Springer Berlin Heidelberg, 2001, vol. 2139, pp. 310–331.

[11] C. S. Jutla, "Encryption modes with almost free message integrity," in *Advances in Cryptology EUROCRYPT 2001*, ser. Lecture Notes in Computer Science, B. Pfitzmann, Ed. Springer Berlin Heidelberg, 2001, vol. 2045, pp. 529–544.