Secure Logging in between Theory and Practice: Security Analysis of the Implementation of Forward Secure Log Sealing in Journald

Felix Dörre*
Astrid Ottenhues*
felix.doerre@kit.edu
astrid.ottenhues@kit.edu

ABSTRACT

This paper presents a security analysis of forward secure log sealing in the journald logging system, which is part of systemd and used in modern Linux distributions. Forward secure log sealing is a cryptographic technique used to ensure the integrity of past log entries even in the event of a full system compromise. We analyze the implementation of this technique in journald, identifying multiple security vulnerabilities resulting from a gap between the model of the cryptographic primitives and their usage in a larger context. In particular one vulnerability allows to forge arbitrary logs for past entries without the validation tool noticing any problem. We demonstrate the found attacks on the journald implementation by providing a concrete security definition for the larger system, an implementation close to the security experiment and a corresponding attacker defeating it when used with a vulnerable version of journald. For the more serious vulnerabilities, we provide patch recommendations, which prevent the implemented attack. Our findings break the security guarantee from log sealing completely, without the error resulting from an inconsistency in the theoretical model nor being a simple implementation mistake. This provides a practical example of the problems that can occur when applying cryptographic primitives to a complex system in reality and that fall in between theory and practice.

KEYWORDS

Secure Logging, Systemd, Journald, Forward Security, Key-Evolving, Implementation Issues, Symmetric Cryptography

1 INTRODUCTION

System logs are an important tool for determining the impact and cause of a security breach. This is also known to attackers who routinely cleanse logs to hide traces of their attacks. Therefore it is important to protect system logs even on a fully compromised system. The two main types of protection are using additional hardware (such as specialized storage or an external logging server) and log sealing to make tampering with logs evident without requiring additional hardware of which the trustworthiness needs to be assumed.

The two main publicly available and practically usable implementations of log sealing are those included in *syslog-ng* and the mechanism implemented in *journald*. As journald is installed by default on many Linux distributions, enabling sealing here, only requires setup of the sealing keys. For many cryptographic systems, designing the system and implementing it are two completely different challenges. While for a theoretic specification there are typically very precise and formally sound security definitions, those often lack when such a system is implemented in a real environment. For journald the whole security definition in the documentation is, that logs sealing "may be used to protect journal files from unnoticed alteration"[1].

To give an idea of the security guarantee a user might expect and as an overview of the more formal definition we will give later, we now show the scenario where log sealing aims to make a difference:

(1) Sealing is enabled on a host system H and the admin saves the verification key confidentially on a verification system V.
(2) The system H logs application defined messages and seals every fixed time interval.
(3) An attacker breaks into system H and has full access to the system including reading the current sealing key and can modify all journal files.
(4) When an administrator (auditor) suspects an attack, or also in regular intervals, the log is copied from H to V and verified there. Verification outputs if the log is consistent and the time span for which verification succeeded.
(5) On system V the administrator displays the log with various `journalctl` options to draw conclusions about the attack.

The intuitive security expectation would be, that an attacker can not modify (or delete/truncate) the journal on system H for all messages that were in a finished epoch before the break-in, without the verification raising an alarm.

1.1 Contribution

This work narrows the gap between the theoretical specification of log sealing and the concrete implementation by journald. We provide a concrete security definition for the sealing system implemented by journald, we deem plausible for what a person relying on such a logging system might expect.

As a main contribution, we uncover three practical security vulnerabilities, which all allow an attacker to break the security definition. One vulnerability allows an attacker to produce arbitrary logs for an arbitrary point of time in the past, breaking all security guarantees that log sealing is trying to make. The other two vulnerabilities concern unnoticed truncation of the log and filtering the log while displaying and both affect the completeness of the

*Both authors contributed equally to this research.

[1] See `man journald.conf(5)`

displayed log. We provide easy and compatible patch suggestions for the most relevant vulnerabilities.

To round this up, we provide a toolkit for inspecting and modifying journal files to have an environment where we can verify the described attacks to demonstrate their practical impact.

1.2 Related Work

Log sealing is a long standing subject of cryptography research as one instance of forward security [1], yet it is still not prevalent in general purpose computer systems. Recent research focuses on the danger of storing, sending and sealing log messages asynchronously, as is common for most logging systems to not impact the applications performance. Paccagnella et al.[9] (with improvements in [6]) make synchronous sealing feasible by splitting sealing from further processing, and optimizing the required synchronous computation. In contrast, preventing log tampering with events just before the compromise is definitely out of journald's scope, as journald does not seal messages one at a time, but using time-based epochs. Journald trades this benefit for allowing to verify log excerpts with epoch granularity. As [9] does not bind the MAC keys to realtime, the whole log from the beginning needs to be verified, otherwise an attacker could forge seemingly old entries and seal them with a new key.

Another research of verifying excerpts in logging scheme was presented in [4], where the authors also formally model log schemes and provide security notions. One security aspect of this work—in addition to forward security—is completeness and truncation resistance of the log. These properties are of high importance to asynchronous logging schemes to get the insurance that an intruder did not modify the log prior to the time span until the break-in. A solution for truncation security is also presented in [5]. Both works are in the line of constructing forward secure logging schemes via forward secure digital signature schemes [1]. The other line of research which constructs forward secure logging schemes via symmetric cryptography started with [2], and our work builds upon this line.

2 PRELIMINARIES

The cryptographic foundation of log sealing in `journald` is the work on seekable sequential key generators by Marson and Poettering [8], which also details their application and implementation into `journald` for log sealing. The construction describes a way to generate a forward secure sequence of keys, that additionally allows efficient seeking without having to calculate all intermediate states. Those keys are then used for a message authentication scheme based on HMAC [7] to seal sections of a log file.

Therefore, we start by introducing the notations and (security) definitions for a seekable sequential key generator SSKG and a forward secure message authentication scheme FSMAS.

2.1 Seekable Sequential Key Generator SSKG

Seekable sequential key generators, introduced by Marson and Poettering [8], are based on the primitive of sequential key generators, introduced in the same work, which is a similar primitive as stateful generators, introduced by Bellare and Yee [2]. We focus on the definitions by [8] as they constructed a SSKG version from *factorization-based shortcut permutations* which yields in combination with a cryptographic MAC the implementation of `journald`.

Definition 2.1 (Seekable Sequential Key Generator SSKG *[8]).* The interface of a SSKG is given by a set of five PPT algorithms :

$$\begin{aligned}
\mathtt{KeyGen}: &\quad (1^\lambda) \mapsto (pubpar, sek)\\
\mathtt{StateGen}: &\quad (pubpar) \mapsto st_0\\
\mathtt{Update}: &\quad (st_i) \mapsto st_{i+1}\\
\mathtt{Seek}: &\quad (sek, st_0, i) \mapsto st_i\\
\mathtt{GetKey}: &\quad (st_i) \mapsto k_i.
\end{aligned}$$

We assume that Update securely erases the *old* state st_i at the end of its call. We expect a SSKG to fulfill its notion of correctness, i.e. that whenever $(pubpar, sek) \leftarrow \mathtt{KeyGen}(1^\lambda)$, and $st_0 \leftarrow \mathtt{StateGen}(pubpar)$, then $\forall i \in \mathbb{N} : \mathtt{Seek}(sek, st_0, i) = \mathtt{Update}^i(st_0)$.

$\mathsf{Exp}^{\mathtt{RoR\text{-}IND\text{-}FS},b}_{\mathtt{SSKG},\mathcal{A}}$

(1) $KList \leftarrow \emptyset$
(2) $(pubpar, sek) \leftarrow \mathtt{KeyGen}(1^\lambda)$
(3) $st_0 \leftarrow \mathtt{StateGen}(pubpar)$
(4) $(state, n, m) \leftarrow \mathcal{A}^{O_{\mathtt{KEY}}}(pubpar)$
$\mathcal{A}$: $k_i \leftarrow O_{\mathtt{KEY}}(i)$:
- $KList \leftarrow KList \cup \{i\}$
- $k_i \leftarrow \mathtt{GetKey}(\mathtt{Update}^i(st_0))$

(5) $k_n^0 \xleftarrow{\$} \{0,1\}^*$
(6) $k_n^1 \leftarrow \mathtt{GetKey}(\mathtt{Update}^n(st_0))$
(7) $st_m \leftarrow \mathtt{Update}^m(st_0)$
(8) $b' \leftarrow \mathcal{A}^{O_{\mathtt{KEY}}}(state, st_m, k_n^b)$
(9) Return 0 if $n \in KList$ or $m \leq n$, else b'

Figure 1: The RoR-IND-FS **security experiment for** SSKG**.**

*Definition 2.2 (*RoR-IND-FS *of* SSKG*, adapted from [8]).* A seekable sequential key generator SSKG is real-or-random indistinguishable with forward security against adaptive adversaries (RoR-IND-FS) if for all PPT adversaries $\mathcal{A}$ that interact in experiment $\mathsf{Exp}^{\mathtt{RoR\text{-}IND\text{-}FS},b}_{\mathtt{SSKG}}$ from Fig. 1, with $b = 0$ being the random and $b = 1$ the real game, the following advantage function is negligible:

$$\mathsf{Adv}^{\mathtt{RoR\text{-}IND\text{-}FS}}_{\mathtt{SSKG},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathsf{Exp}^{\mathtt{RoR\text{-}IND\text{-}FS},1}_{\mathtt{SSKG},\mathcal{A}} = 1\right] - \Pr\left[\mathsf{Exp}^{\mathtt{RoR\text{-}IND\text{-}FS},0}_{\mathtt{SSKG},\mathcal{A}} = 1\right]\right|.$$

2.2 Forward Secure Message Authentication Scheme FSMAS

Message authentication schemes with symmetric cryptography are used to provide integrity protection and authenticity to messages. These schemes use a private key, which is kept secret, to generate a message authentication code (MAC) *tag* that can be used to verify the integrity of the message and authenticate the sender.

Definitions for message authentication schemes MAS are provided in Appendix A.

Forward security in message authentication schemes refers to the property that a compromised key in the future cannot be used to retroactively forge MACs for messages that were logged in the past. In other words, even if an attacker gains access to the secret key used to generate MACs in the future, they cannot use that knowledge to create new MACs for modified past messages.

Definition 2.3 (Forward Secure Message Authentication Scheme FSMAS *based on [2]).* The interface of a FSMAS is given by a set of four PPT algorithms :

$$\begin{aligned} \texttt{KeyGen}: &\quad (1^\lambda) \mapsto (k_0) \\ \texttt{Update}: &\quad (k_i) \mapsto (k_{i+1}) \\ \texttt{Sign}: &\quad (k_i, msg) \mapsto (tag_j) \\ \texttt{Vfy}: &\quad (k_i, msg, tag_j) \mapsto \{0,1\}. \end{aligned}$$

We expect a FSMAS to fulfill the notion of correctness, i.e. that whenever $(k_0) \leftarrow \texttt{KeyGen}(1^\lambda)$ and $\forall i > 0 : (k_{i+1}) \leftarrow \texttt{Update}(k_i)$, then $\forall i, j : (tag_j) \leftarrow \texttt{Sign}(k_i, msg)$ holds $\texttt{Vfy}(k_i, msg, tag_j) = 1$.

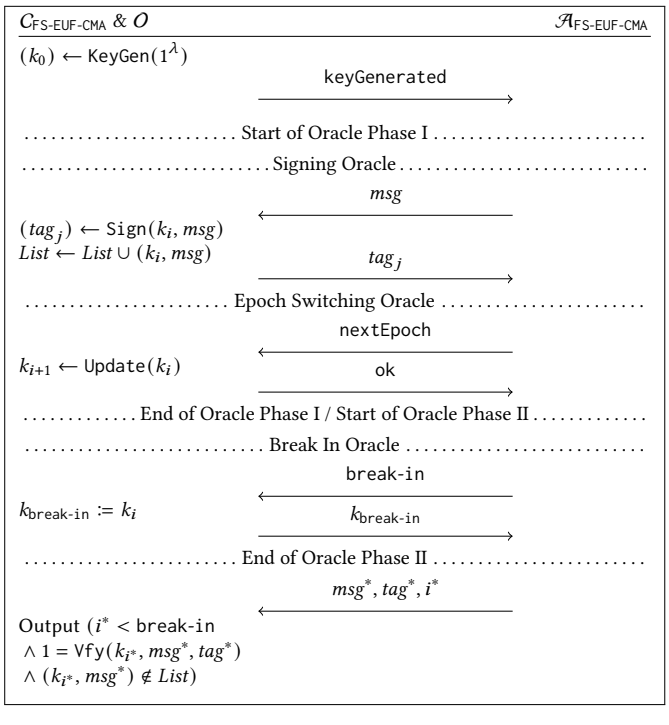

Figure 2: The FS-EUF-CMA **Game for** FSMAS

*Definition 2.4 (*FS-EUF-CMA *of* FSMAS*).* A FSMAS is forward secure existentially unforgeable under chosen message attacks (FS-EUF-CMA) if for any PPT adversaries $\mathcal{A}_{\text{FS-EUF-CMA}}$ the advantage to win the FS-EUF-CMA game shown in Fig. 2 is negligible in λ.

The number of epochs, stated by counter i, increases with every key update. The counter j states the number of tags, i.e. the signatures of the MAS. Both counters are independent variables, e.g., the amount of tags per epoch can be greater than one. An FSMAS can be constructed from a message authentication scheme (Definition A.1) by using keys generated from an SSKG. In journald the SSKG is the one presented in [8] based on factoring and the MAS is HMAC (Hash-based Message Authentication Code) [7].

3 SECURITY ANALYSIS OF A FORWARD SECURE SYMMETRIC LOG SCHEME

To begin with, we define in Section 3.1 a forward secure seekable message authentication scheme FSSMAS combining the security properties of a forward secure message authentication scheme FSMAS given in Section 2.2 with the seeking feature of a seekable sequential key generator SSKG given in Section 2.1. We construct the FSSMAS from a message authentication scheme MAS given in Appendix A, i.e. HMAC, with keys provided by a SSKG of Section 2.1.

With this FSSMAS, we build in Section 3.2 a forward secure log scheme LS, which is very close to the implementation of the *Forward Secure Log Sealing* in journald. The LS is a formal model of the implementation to provide a direct security analysis and thereby reducing the gap between theory and practice.

3.1 Forward Secure Seekable Message Authentication

In the following definition of a forward secure seekable message authentication scheme, we highlight in orange the adaptions to include the function of *seeking* in the definition of forward secure message authentication of Definition 2.3. *Seeking* allows to calculate in a fast-forward manner an actual state/key solely with the knowledge of the first state/key and a seeking information/key *sek*. It is easy to see, that *seeking* is a function to improve the efficiency of a scheme and does not effect its security. We include this feature to be closer to the real-world, i.e. the implementation of journald.

Definition 3.1 (Forward Secure Seekable Message Authentication Scheme FSSMAS*).* The interface of a FSSMAS is given by a set of five PPT algorithms:

$$\begin{aligned} \texttt{KeyGen}: &\quad (1^\lambda) \mapsto (sek, st_0) \\ \texttt{Update}: &\quad (st_i) \mapsto (st_{i+1}) \\ \texttt{Sign}: &\quad (st_i, msg) \mapsto (tag_j) \\ \texttt{Seek}: &\quad (sek, st_0, i) \mapsto (st_i) \\ \texttt{Vfy}: &\quad (st_i, msg, tag_j) \mapsto \{0,1\}. \end{aligned}$$

We expect a FSSMAS to fulfill the notion of correctness, i.e. that whenever $(sek, st_0) \leftarrow \texttt{KeyGen}(1^\lambda)$ and $\forall i > 0 : (st_{i+1}) \leftarrow \texttt{Update}(st_i)$, then $\forall i : st_i = \texttt{Seek}(st_0, sek, i)$ and $\forall i, j : (tag_j) \leftarrow \texttt{Sign}(st_i, msg)$ holds $\texttt{Vfy}(st_i, msg, tag_j) = 1$.

Construction 1 (Forward Secure Seekable Message Authentication Scheme FSSMAS). *Given a seekable sequential key generator* SSKG *as defined in Definition 2.1 and a message authentication scheme* MAS *as defined in Definition A.1, we construct a forward secure seekable message authentication scheme* FSSMAS *in Fig. 3.*

Construction 1 gives a concrete construction of a FSSMAS. In order to derive the MAC keys using SSKG we use the generated

$\mathsf{KeyGen}(1^\lambda)$:
- $(pubpar, sek) \leftarrow \mathsf{SSKG.KeyGen}(1^\lambda)$
- $(st_0) \leftarrow \mathsf{SSKG.StateGen}(pubpar)$

$\hookrightarrow$ Output (sek, st_0).

$\mathsf{Update}(st_i)$:
- $(st_{i+1}) \leftarrow \mathsf{SSKG.Update}(st_i)$

$\hookrightarrow$ Output (st_{i+1}).

$\mathsf{Sign}(st_i, msg)$:
- $k_i \leftarrow \mathsf{MAS.KeyGen}(1^\lambda; \mathsf{SSKG.GetKey}(st_i))$
- $tag_j \leftarrow \mathsf{MAS.Sign}(k_i, msg)$

$\hookrightarrow$ Output (tag_j)

$\mathsf{Seek}(sek, st_0, i)$:
- $st_i \leftarrow \mathsf{SSKG.Seek}(sek, st_0, i)$

$\hookrightarrow$ Output (st_i)

$\mathsf{Vfy}(st_i, msg, tag_i)$:
- $k_i \leftarrow \mathsf{MAS.KeyGen}(1^\lambda; \mathsf{SSKG.GetKey}(st_i))$

$\hookrightarrow$ Output $\mathsf{MAS.Vfy}(k_i, msg, tag_i)$.

Figure 3: The Construction of Forward Secure Seekable Message Authentication Scheme FSSMAS

keys as randomness for MAS.KeyGen. The security notion FS-EUF-CMA FSMAS (Definition 2.4) can directly be adapted as notion for FSSMAS as the $\mathcal{A}_{\text{FS-EUF-CMA}}$ does not get the output of SSKG.GetKey.

The security of Construction 1 is similar to the security of the same scheme without Seek algorithm, as the additional key from KeyGen is not used by the challenger and the existence of another algorithm as such does not give the adversary any advantage.

A security argument can be given via a hybrid game, where the intermediate keys are not generated by the SSKG, but are drawn by the challenger at random. The attacker cannot distinguish this game from the actual security game, because of the RoR-IND-FS security of SSKG. The remaining security game is exactly the EUF-CMA security of MAS.

3.2 Construction of a Forward Secure Seekable Log Scheme

A log scheme is a protocol that runs between a log host, auditor and an application which sends messages to be logged. Bases for our model are taken from the secure logging models of [4, 5]. However these definition do not fit the logging scheme created by journald as they are using asymmetric signing keys and do not have a mechanism to incorporate seeking, which is the primary feature of journald's implementation.

We construct a logging scheme LS which is based on a forward secure seekable message authentication scheme as presented in Section 3.1. One main goal of the description of LS is to provide a connection between the real-world implementation of the *Forward Secure Sealing* in *journald* and its idealized theoretical model. Therefore, we highlighted some parts of the construction, i. e. additions and ~~removals~~, which describe the improvements upon the implementation in systemd version 252, which has security issues as pointed out in Section 5.

Definition 3.2 (Logging scheme LS*).* The interface of a log scheme LS is given by a set of four PPT algorithms (KeyGen, Sign, AppendAndSeal, Vfy):

$$\begin{aligned}
\mathsf{KeyGen}: &\quad (1^\lambda) \mapsto (sek, state)\\
\mathsf{Sign}: &\quad (state) \mapsto (state, journal)\\
\mathsf{AppendAndSeal}: &\quad (state, msg, time) \mapsto (state, journal)\\
\mathsf{Vfy}: &\quad (sek, journal, target) \mapsto \{0,1\}\\
\mathsf{Query}: &\quad (journal, target) \mapsto (msg)_i
\end{aligned}$$

We expect a LS scheme to fulfill the notion of correctness: After an arbitrary sequence of Sign and AppendAndSeal invocations, with the last message having $time^*$. For all *target* with $\mathsf{EpochStart}(target + 1) \leq time^*$: $\mathsf{Vfy}(sek, journal, target)$ is true, and $\mathsf{Query}(journal)$ returns all messages with $time < \mathsf{EpochStart}(target)$, logged with AppendAndSeal.

Semantically, the difference between *signing* and *sealing* is that we require a sealing algorithm to evolve the key and to securely erase the old key. The assumption of *secure erasure* is an important fact and a research topic on its own as it is not easy to fulfill [3]. A *journal* as output of the Sign and AppendAndSeal algorithm is the representation of the whole log including all log messages, tags, and counters. The Sign algorithm serves no functional purpose but is used to model the behavior of the real journald application.

Construction 2 (Forward Secure Log Scheme). *Given a forward secure seekable message authentication scheme* FSSMAS *as defined in Definition 3.1, we construct a forward secure log scheme* LS *in Fig. 4. We describe the behaviour of a executing log host* **host**, *an auditor* **auditor**, *and an application* **app** *which provides messages. The algorithm* $\mathsf{EpochStart}(i)$ *returns the actual fixed time when the epoch i started. In practice the mapping between realtime and epoch is chosen when the key is generated. For example, the epoch starts with value zero at the time of key generation and increments every 15 minutes. The parameter isfinal is set when an epoch is sealed, e.g. when* AppendAndSeal *is called.*

3.3 Security Notion for a Log Scheme

We show that the previously constructed log scheme LS of Section 3.2 provides the security notion of FS-EUF-CLMA for symmetric log schemes by reducing this security to the notion of the FS-EUF-CMA security of the underlying FSSMAS scheme.

In the security experiment we will let a challenger play the role of the log host and auditor, while the adversary can send messages with the goal to forge log messages in the end. When an application sends `init` to the host—triggered in practice e.g. by the suspension of a system—then the actual log will only be signed but not sealed. The sealing takes place, when a log message is processed: The message will be appended to the actual log and the log is sealed.

*Definition 3.3 (*FS-EUF-CLMA *of* LS*).* A log scheme LS is forward secure existentially unforgeable under chosen log message attacks

$state = (log, journal, i, j, st_i)$

$\mathsf{KeyGen}(1^\lambda)$:

- $(sek', st_0) \leftarrow \mathsf{FSSMAS.KeyGen}(1^\lambda)$
- Initialize $log, journal := \emptyset$.
- Initialize epoch counter i and tag counter j: $i, j := 0$.

↪ Send $(sek', state)$ to **auditor** and erase sek.

$\mathsf{Sign}'(state, isfinal)$:

- $(tag_j) \leftarrow \mathsf{FSSMAS.Sign}(st_i, (log, i, j, isfinal))$
- $log_j := log$
- $journal \leftarrow journal \| (log_j, tag_j, i, j, isfinal)$
- Empty $log := \emptyset$
- Increase tag counter j

↪ Output $(state, journal)$.

$\mathsf{Sign}(state)$:

↪ Output $\mathsf{Sign}'(state, 0)$.

$\mathsf{AppendAndSeal}(state, msg, time)$:

- While ~~If~~ $(time \geq \mathsf{EpochStart}(i+1))$:
  - $\mathsf{Sign}'(st_i, 1)$
  - ~~While $(time \geq \mathsf{EpochStart}(i+1))$~~
    - $st_{i+1} \leftarrow \mathsf{FSSMAS.Update}(st_i)$
    - Erase securely old state st_i
    - Increase epoch counter i
- $log \leftarrow log \| (msg, time)$

↪ Output $(state, journal)$.

$\mathsf{Vfy}(sek, journal, target)$:

- Parse $sek : (sek', st_0)$
- Parse $journal : (log_j, tag_j, i, j, isfinal_j)$
- $st_i \leftarrow \mathsf{FSSMAS.Seek}(sek', st_0, i)$
- If not $\forall (log_j, tag_j, i, j, \cdot) \in journal$: $\mathsf{FSSMAS.Vfy}(st_i, (log_j, i, j, isfinal), tag_j)$, output 0
- If the sequence of all tag counters j in $journal$ is not continuous and strictly monotonic, output 0
- If the sequence of all epoch counters i in $journal$ is not monotonic, output 0
- If not $\forall i \in [0, target] : (\cdot, \cdot, i, \cdot, 1) \in journal$, output 0
- ~~If not $\forall (log_j, \_, i, j, \_) \in journal \; \forall time \in log_{k>j}$: $time \geq \mathsf{EpochStart}(i)$, output 0~~
- ~~If not $\forall (log_j, \_, i, j, \_) \in journal$: $time_{|log_{j-1}|} \in log_{j-1} < \mathsf{EpochStart}(i+1)$, output 0~~
- If not $\forall (log_j, \cdot, i, j, \cdot) \in journal \; \forall time \in log_j$: $\mathsf{EpochStart}(i) \leq time < \mathsf{EpochStart}(i+1)$, output 0

↪ Output 1.

$\mathsf{Query}(journal, target)$:

- Parse $journal : (log_j, tag_j, i, j, isfinal_j)$

↪ Output $\{msg_k | i \in [0, target], (log_j, \cdot, i, j, \cdot) \in journal, (msg_k, time_k) \in log_j\}$

Figure 4: The Construction of Forward Secure Seekable Log Scheme LS.

(FS-EUF-CLMA) if for any PPT adversaries $\mathcal{A}_{\mathsf{FS\text{-}EUF\text{-}CLMA}}$ the advantage to win the FS-EUF-CLMA game shown in Fig. 5 is negligible in the security parameter λ.

It is trivial for an adversary to forge a log message after and retrospectively during the epoch of the break-in as it gets the SSKG-state of that epoch and can derive the signing key. Also the truncation during the last epoch is trivial for an attacker. Therefore, the security notion provides forward secure existential unforgeability until the last epoch before the break-in.

Lemma 3.4. *The log scheme* LS *fulfills the security of* FS-EUF-CLMA*, e.g. it is forward secure existentially unforgeable under chosen log message attacks, if the underlying forward secure message authentication scheme* FSSMAS *is* FS-EUF-CMA*-secure.*

Proofsketch. *Assuming that* (FSSMAS.KeyGen, FSSMAS.Update, FSSMAS.Sign, FSSMAS.Seek, FSSMAS.Vfy) *is a correct forward secure message authentication scheme* FSSMAS *fulfilling* FS-EUF-CMA *security and assuming we have an adversary* $\mathcal{A}_{\mathsf{FS\text{-}EUF\text{-}CLMA\text{-}LS}}$ *who has non-negligible success probability in winning the* FS-EUF-CLMA *game given in Fig. 5 with respect to* LS = (KeyGen, Sign, AppendAndSeal, Vfy)*, we can directly derive an adversary* $\mathcal{A}_{\mathsf{FS\text{-}EUF\text{-}CMA\text{-}FSSMAS}}$ *with non-negligible success probability in winning the* FS-EUF-CMA *game given in Fig. 2 with respect to* FSSMAS *by reformulating the query messages to log scheme queries and vice-versa. If* $\mathcal{A}_{\mathsf{FS\text{-}EUF\text{-}CLMA\text{-}LS}}$ *wins the* FS-EUF-CLMA *game, then it modified journal* in one of the following ways:*

- *If journal* and journal have the same amount of sealed messages,* $\mathcal{A}_{\mathsf{FS\text{-}EUF\text{-}CLMA\text{-}LS}}$ *provides a journal* including* (msg^*, tag^*, i^*) *which is not in journal, e.g.* $\mathcal{A}_{\mathsf{FS\text{-}EUF\text{-}CLMA\text{-}LS}}$ *changed a msg.*
- *If journal* is shorter than journal,* $\mathcal{A}_{\mathsf{FS\text{-}EUF\text{-}CLMA\text{-}LS}}$ *provides a last message in last epoch of journal* that can be used as forgery message* msg^**, and vice-versa (if journal is shorter). The corresponding message that is included in journal is different as either "isfinal" was not set or the message did not exist. That means that* msg^* *was never requested from* O_{Sign}.

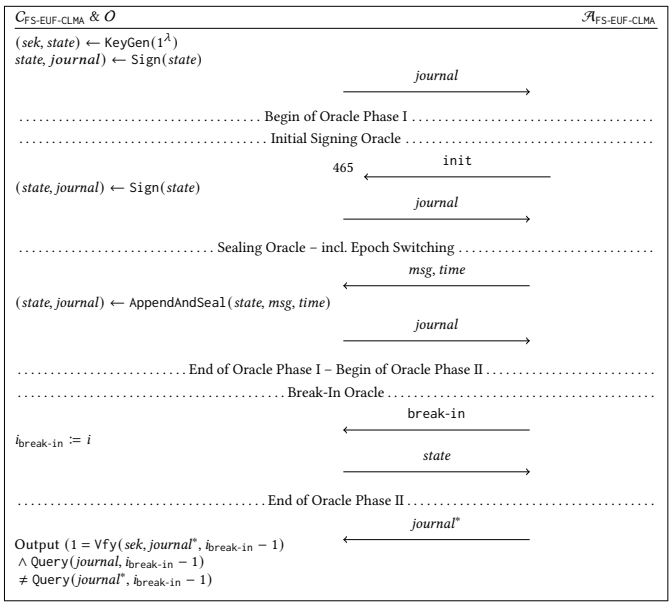

Figure 5: The FS-EUF-CLMA **security game for** LS

The found message msg^* can be used by $\mathcal{A}_{\text{FS-EUF-CMA-FSSMAS}}$ to win the FS-EUF-CMA *game. It is a successful forgery pair as* FSSMAS.Vfy *outputs 1 and the O_{Sign} did not sign the message msg^* in epoch i.*

We can conclude the following theoretical results: Let SSKG be a RoR-IND-FS secure seekable sequential key generator. Let MAS be a EUF-CMA secure message authentication scheme. Then, we get a seekable FS-EUF-CMA secure forward secure seekable message authentication scheme FSSMAS. This yields a FS-EUF-CLMA secure forward secure seekable log scheme LS.
Additionally and simultaneously, the protocol for the log scheme LS is a model close to the implementation of *Forward Secure Sealing* in journald.

4 THE JOURNAL FILE FORMAT

While the formal model describes the journal file as a sequence of tuples, the reality looks much more complex and involved. Log messages are not plain strings but key-value pairs, that store meta information in addition to the regular log messages. To allow quick access to sub-sequences of the journal (also based on the meta fields) and to not store multiple occurrences of the same value multiple times, journals are stored as a structure of linked objects indexed by hash tables. In order to understand more concretely how the attacks can be carried out in practice we start with a brief overview of the relevant details of the on-disk format before describing the actual attacks.

4.1 Structure

For journald each log entry is, apart from time and sequence information, a set of key-value pairs encoding the log message, and other structured information like log-level, executable name, systemd unit name of the logging program. Journald stores logs in a binary file format on-disk. The individual structures are defined in `journal-def.h` [2], with a high-level description of the structure given in an accompanying documentation. [3]

On a high level a journal file consists, after a header, of a sequence of objects each identified with type and length. These objects contain data and references to each other as addresses (offsets from the start of the journal file), visualized in Figure 6. Each log entry is stored as an `EntryObject` e referencing multiple `DataObjects` $D(e)$, one for each key-value pair, reusing `DataObjects` that contain the same data. `FieldObjects` are used to link all `DataObjects` with the same key together. See Appendix B for a simplified example of a textual representation of a journal file.

For writing the journal file the main design goal was to allow appending new log entries with only changing little of the existing journal file. Also highly repetitive messages (as well as repetitive values for other fields) should only be stored once. For reading, the file format allows to quickly seek to specific time ranges, display log messages with a specific value for a field, and listing all fields used and possible values for all those fields. These different access modes are supported by a collection of quick-access structures that are maintained in the file:

- A hash table allowing to access `FieldObject` by value (H_f).

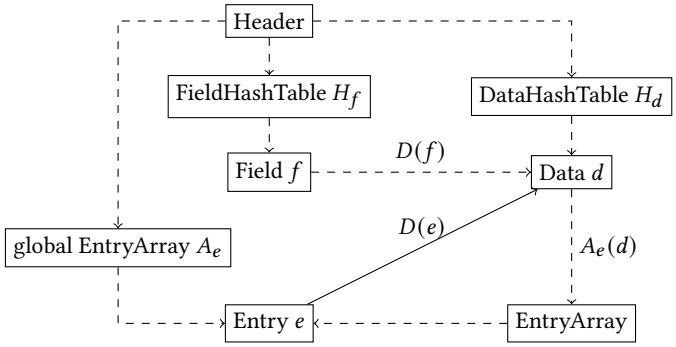

Figure 6: Journal File Format, the `Object` suffix of each of the entity names is omitted. Only the solid link between Entry and Data is authenticated by MACs

- A hash table allowing to access `DataObject` by value (H_d).
- The data objects chained in a linked list $D(f)$, allowing to list all individual values for field f.
- A global `EntryArray` A_e to list all entries in the log file.
- A chain of `EntryArrays` $A_e(d)$ per data object d, listing all usages of the data object.

Accessing the log file can happen in various ways. When a user queries for all log entries in a given range using the `journalctl` tool, that program will use the the global `EntryArray` to iterate through the log entries and print the messages. When a user queries log entries for a given unit, `journalctl -u` will locate the corresponding `DataObject` and use the corresponding `EntryArrays` to print only the matching entries. When specifying the unit with a glob expression, instead of a concrete string, `journalctl` will use the unique values of that field to evaluate the glob expression on and then query log entries for all matching unique values. Special options of `journalctl` can be used to list all fields and all unique values of a field.

Time. For each `EntryObject` there are multiple timestamps stored. As time keeping is a complex topic, journald obtains and stores *real time* and *monotonic time* for each entry the moment the entry is processed by journald. Optionally entries may also carry a timestamp attached by the logging process. They are stored additionally in the `_SOURCE_REALTIME_TIMESTAMP` and `_SOURCE_MONOTONIC_TIMESTAMP` fields.

When displaying a log, per default the `_SOURCE_REALTIME_TIMESTAMP` is displayed, falling back to the internal real time timestamp if the source timestamp is not available. When choosing an output format displaying monotonic time, the same logic is applied, displaying `_SOURCE_MONOTONIC_TIMESTAMP` and falling back to the internal monotonic time.

The semantics of real and monotonic time are inherited from their semantics in the linux kernel, described in `man clock_gettime(2)`:

> **real time** A settable system-wide clock that measures real (i.e., wall-clock) time. [...] This clock is affected by discontinuous jumps in the system time [...], and by the incremental adjustments performed by `adjtime(3)` and NTP.

[2] https://github.com/systemd/systemd/blob/v252/src/libsystemd/sd-journal/journal-def.h
[3] https://systemd.io/JOURNAL_FILE_FORMAT/

> **monotonic time** A nonsettable system-wide clock that represents monotonic time[...]. [...] The CLOCK_MONOTONIC clock is not affected by discontinuous jumps in the system time [...], but is affected by the incremental adjustments performed by `adjtime(3)` and NTP.

4.2 Sealing with the Journal File Format

In order to seal the journal, `TagObjects` are regularly appended to the log file. They are numbered with an `epoch` and `seqnum`, corresponding to i and j from the theoretical model, respectively. They contain a MAC across parts of all objects until that point, starting from the last `TagObject` or the beginning of the file, for the first tag. The MAC covers the type and length for each object and all parts that are considered immutable (i. e. will not be modified when appending), in particular the actual data. This ensures that for all immutable parts the MAC secures the object's location in the journal file. To put it another way, without having a MAC key, the only messages an attacker can get accepted by the MAC-check are prefixes of the original log file where "mac-covered" regions are unchanged, and all other regions can contain arbitrary data. In particular, `FieldObject` and `DataObject` have their payload secured and `EntryObjects` have all pointers to `DataObject` secured. All structures and fields that are modified after writing the object initially are not covered by the MAC and their integrity relies on additional checks while verifying the journal.

Time for Sealing. The keys for sealing are bound to fixed-length real time epochs. Whenever a real time epoch expires (and definitely before any entry of the next epoch is written down), `journald` appends a tag and evolves the sealing key. Additionally, when stopping (for example in case of a clean system shutdown), `journald` appends a tag without evolving the key. As the epoch does not have ended yet, the key cannot be evolved as it might be needed again to seal further messages in that epoch, that might occur if `journald` is started again before the epoch ends.

Verification of a journal file imposes three restrictions on the timestamps:

- An entry following a tag must not have an older realtime than the beginning of the epoch for that tag
- The last entry before a tag must not have a newer realtime than the end of the epoch of that tag
- If two entries share the same `bootid`, the monotonic time between them must not decrease.

Note, that both `_SOURCE_*` timestamps are not verified in any way. This is especially problematic as they are overriding the internal timestamps when being displayed in all output formats except `verbose`, `export` and `json`, including the default.

The epochs for the sealing keys are interpreted as real time, which means that monotonic time is not bound to the epochs of the sealing keys, and just checked for internal consistency.

Also note that the constraints on an entry's real time are very lax. As regular operation of `journald` may produce two tags for the same epoch (which we suggest to omit, see Problem 2), the first constraint cannot be simply tightened. But as-is those constraints do not prevent an attacker from sealing back-dated entries with a tag from a newer epoch (see Problem 1).

We propose to extend (or replace) those checks with a different check: For each sealed section we propose to obtain the minimal and maximal real time timestamp over all entries and verify that this interval is completely contained in the epoch interval of the following seal. This directly subsumes the second check, and in combination with a check that crypto epochs only increase, it also contains the first check. These adjustments are already included in Construction 2.

5 SPECIFIC PROBLEMS

We now describe the three individual issues that each allow different kind of forgeries and thereby break the security definition.

5.1 Sealing older entries with newer keys (CVE-2023-31439)

While the verification checks that the tags for a given crypto-epoch are not used to seal newer entries, the check in the other direction is missing allowing a key to seal arbitrary journal entries. This is a problem, as the current sealing key needs to be stored on the system, allowing an attacker to use it to seal (or re-seal) existing journals. In particular an attacker having broken into a system, can use the current (newer) sealing key to generate new MACs for any previous `TagObjects` after tampering with the log entries. Verification would report that the sealing happened correctly and the modifications go unnoticed. If messages prior to the currently last tag should be added, that tag needs to be removed first.

Remediation Recommendation. We recommend to extend the check for log entries against crypto epochs to also report too old entries sealed with a given crypto key and not only too new entries. The journal writer as-is will not seal too old entries with a newer key, so this change should not report any false-positive violations.

We deem the patch for this issue rather simple and provide a suggestion in `CVE-2023-31439.patch` in the supplementary material. This patch is compatible in both directions. As it only modifies the verification, the produced journals are identical with the patch compared to without it. That means that applying this patch does not create any incompatibility between systemd versions.

5.2 Attack on Completeness (CVE-2023-31438)

While log sealing can confirm the existence of log entries, it can also confirm the completeness of a given log section. In particular, if verification of a journal file succeeds for a given time frame the attacker should not be able to add, modify or remove any log entry in that time frame.

Journald's handling of epochs without log entries does not allow to verify the absence of log entries. Specifically an attacker breaking into a system can truncate previous sealed epochs of the system log without being revealed when checking the log seals, as the journal is not updated when an epoch passes without new log entries; only the sealing key is evolved forward.

Luckily for most practical applications epochs without log entries are very rare—except the system is suspended. So an epoch without sealed log entries is suspicious by itself.

Additionally due to the mechanism of sealing multiple times, an attacker might delete tagged blocks of the last epoch. As there are

multiple tags from the same key in such a case, an attacker can just delete blocks from the end, except the first one, and adjust the tag-counter to resume logging without being detected.

Additionally, this motivates further investigation whether the current verification implementation notices gaps in a journal spanning multiple files.

Remediation Recommendation. We recommend to create new `TagObjects` regardless of whether there are new log entries in the corresponding epoch, when moving to the next epoch. The `TagObjects` are relatively small in comparison to regular log entries and there are rarely epochs without log entries in practice. In the special case, when a system has been suspended for more than one epoch, Journald would have to create `TagObjects` for every epoch that was missed. This can be made more efficient, by adding another field to the `TagObject`, indicating how many epochs are about to be skipped (and then be sealed with the old key). That way the journal only grows by a constant amount, and the missing empty epochs are still sealed. The verification process needs to be adjusted to recognize this new field (ensuring that each epoch is accounted for).

Additionally each `TagObject` needs to be marked if it is the last tag for a given crypto epoch to prevent truncation in the last finished epoch. We are not aware of a simple implementation that can keep this compatible with the current implementation. Alternatively signing parts before an epoch ended could be skipped. This operation does not seal entries, as it does not evolve the key and thereby does not prevent an attacker from re-creating the tag. As multiple tags with the same key do not increase (in this case even decrease) security, un-sealed entries could be left as-is when journald is stopped. When logging resumes, the entries can be sealed if the epoch has ended, or get new entries appended, when it has not. This change would be compatible.

We provide an incomplete suggestion for the simple remediation variant, in `CVE-2023-31438-incomplete.patch` in the supplementary material. Together with removing the intermediate tagging the patch resolves this vulnerability. This patch is compatible in the way that new journals can still be verified (incompletely) with the old version, but journals produced without this patch might fail verification with the new version if an epoch is missing. The more efficient variant would probably require more considerations if extra care should be taken that the new journals still pass verification with older version, if that is desired.

5.3 Missing verifications of fast access structures (CVE-2023-31437)

Currently the verification process builds three lists of locations while scanning through the journal initially. Let L_e be the list of all `EntryObjects`. Let L_a be the list of all `EntryArrayObjects`. Let L_d be the list of all `DataObjects`. These are used later to verify some of the additional access structures. Note that `FieldObjects` are suspiciously absent here. For the `FieldHashTable` and `DataHashTable` additional lists are not required as these objects are guaranteed to appear at most once in a journal file which can be checked without creating a list.

We list the additional checks along with references into the source code of `journal-verify.c` [4] of version 252:

- $|A_e| = |L_e|$ [line 1235, line 711]
- The addresses in A_e are strictly increasing [line 745]
- $A_e \subseteq L_e$ [line 751]
- All used `EntryArrays` are in L_a [line 725]
- $\forall e \in A_e : D(e) \subseteq L_d$ [line 656]
- $\forall e \in A_e : D(e) \subseteq H_d$ [line 665]
- $\forall e \in A_e : \forall d \in D(e) : e \in A_e(d)$ [line 685]
- $H_d \subseteq L_d$ [line 564]
- H_d contains only unique elements [line 574, line 579]
- $\forall d \in H_d : A_e(d) \subseteq L_e$ [line 456, line 503]
- All `EntryArrays` used in $A_e(d)$ are in L_a [line 478]
- $\forall d \in H_d : A_e(d) \subseteq A_e$ [line 561, line 508] (not needed)

Missing checks:

(1) Missing check for fields: Build a complete list of authenticated fields L_f (like for Data/Entry/EntryArray) and verify the `FieldHashTable` against it (similar as done for `DataHashTable`, with additionally checking the count): $L_f = H_f$.
(2) Missing check for the linked `DataObjects`: For each `FieldObject` check that the payload does not contain "=". Go through the list of linked `DataObjects`, check that their payload starts with the field name followed by "=". Count the number of visited `DataObjects` and check that it matches the number of valid data objects.
This (Missing checks 1, 2) allows an attacker to change the set of "visible" fields (`journalctl -N`) and the available values for each field (`journalctl -F <field>`). As a consequence, when the journal is queried with a glob expression for a field like a unit (e.g. `journalctl -u myservice*`), the glob expression is evaluated on the adjusted set of values.
Luckily those data objects are not used directly but looked up in the `DataHashTable` instead for retrieving matching entries, so an attacker can only hide log entries.
(3) Missing check if `DataObjects` only reference back to correct `EntryObjects`: In verify_data check that all referenced entries link back to the corresponding `DataObject`, similar to the check in verify_entry: $\forall d \in H_d : \forall e \in A_e(d) : d \in D(e)$. The missing check allows an attacker to add additional entries to the `EntryArrayObject` of a `DataObject`, leading to additional log entries being displayed when filtering for a specific `DataObject`.

Remediation Recommendation. We recommend to add the missing checks. Additionally the verification process could also check that there is exactly one `DataHashTable` and one `FieldHashTable` to avoid an internal error later. Currently, the code only checks if there is "at least one".

6 IMPLEMENTATION

For this work we created a small toolkit allowing the close inspection and modification of existing journal files to demonstrate the

[4] https://github.com/systemd/systemd/blob/v252/src/libsystemd/sd-journal/journal-verify.c

described security flaws. In contrast, the implementation from systemd focuses on generating valid journal files and is optimized for appending to an existing journal file. Rewriting a journal file and adjusting pointers is not that easy with the library provided by systemd. However, the journal file that `journalctl --verify` consumes comes from a potentially compromised system, making it adversary-controlled input. In particular this means, that `journalctl --verify` needs to cope with any possible journal file contents. Our implementation allows the creation of unexpected journal files to simulate attacker behavior.

We provide an implementation of the `SSKG` construction by [8] in `FSPRNGKey` to be able to derive future sealing keys from the `fss`-file on disk. Additionally, we implement logic to parse the different journal objects in `JournalObject` to allow inspection and modification. That foundation is then used in `Attacker` to provide a concrete attacker against the security definition. To show that this attacker wins against the security definition we implement a specialization of the security experiment (Figure 7) to show that the behavior of this attacker wins in the security experiment. We omit to allow the attacker to specify the log displaying query as filtering the query for allowed `journalctl` options is complex and does not help this demonstration.

In addition we implement a test harness to allow the isolated invocation of a specific `journald` binary. The harness facilitates interaction between the security experiment and the journal process: One interaction is feeding specific log messages from the security experiment or simulated adversary, while regular log messages of the hosting system are not affected. Another interaction is to advance the epoch to force sealing of the log without having to wait for the epoch to expire normally. To advance epochs the harness uses libfaketime[5] and adjusts the perceived system time for the `journald` process accordingly. Lastly, the test harness invokes `journald` using Bubblewrap (`bwrap`[6]) to bind mount test directories. This is required, because `journald` searches for configuration, journal and the key files in predefined hard coded paths. Bubblewrap allows to set up a mount namespace for the process under test, and adjust the directories visible under those paths. This is even allowed for non-privileged users, which is comfortable for running test cases and attacker simulations without requiring root privileges.

Figure 7 shows our implementation of the security experiment (from `Challenger`). In line 2 the attacker is given the opportunity to interact with the log system through the interface shown in Figure 8. When that interaction is done, the oracle is deactivated (line 3), and the experiment advances to the break-in phase. The attacker is given a copy of the journal file to modify in line 10. After modification `journalctl --verify` is invoked to check that the modified journal still passes verification and note is taken of the reported "sealing"-times in line 12. The same is done for the journal before modification (line 13). If the attacker-modified journal is reported as being sealed less (line 14), the attack has failed. In the security definition this is covered inside `Vfy`. The timestamp from `journalctl --verify` is rounded towards the past, so we need to increment it by one second for the `-U` option of journalctl to ensure

[5]https://github.com/wolfcw/libfaketime
[6]https://github.com/containers/bubblewrap

```
1  AtomicBoolean inPhase1 = new AtomicBoolean(true);
2  a.interactWithLog(getLogAdapter(inPhase1));
3  inPhase1.set(false);
4
5  // Break-in Phase
6  Files.copy(journal, journalPatched, StandardCopyOption.
       REPLACE_EXISTING);
7  JournalFileBuffer fileToModify = JournalFileBuffer.open(
       journalPatched, true);
8  FSPRNGKey key = new FSPRNGKey(JournalFileBuffer.open(
       keyFile, false));
9
10 a.tamperWithLogfile(fileToModify, key);
11
12 Date untilAttacker = verifyJournal(journalPatched);
13 Date untilOriginal = verifyJournal(journal);
14 if(untilAttacker.getTime() < untilOriginal.getTime()) {
15     throw new AttackFailedException("Attacker␣patched␣
           journal␣is␣sealed␣less");
16 }
17 String untilOffByOne = dateFormat
18         .format(new Date(untilOriginal.getTime() + 1000))
               ;
19
20 byte[] reply = runProcess(JOURNALCTL, "--file=" + journal
       , "-U", untilOffByOne);
21 byte[] replyPatched = runProcess(JOURNALCTL, "--file=" +
       journalPatched, "-U", untilOffByOne);
22 if (Arrays.equals(reply, replyPatched)) {
23     throw new AttackFailedException("Output␣does␣not␣
           differ");
24 } else {
25     System.out.println("Attack␣Succeeded");
26 }
```

Figure 7: The specialization of the security experimented for demonstration

```
1  public interface LogAdapter {
2      public void log(String message);
3
4      public void advanceEpoch();
5
6      public JournalFileBuffer obtainJournal();
7
8  }
```

Figure 8: The interface for the adversary before the break-in phase

that the latest sealed message is included as well (line 17). Finally, we display the journal (line 20 and 21). This attack requires no special query parameters, so we left out asking the adversary for query parameters, but just display the journal contents regularly. The experiment ends successfully if the displayed contents differ (line 22).

7 DISCLOSURE

On January 18, 2023 we reached out to the security contact for systemd with a preliminary version of this article detailing the findings together with a high-level overview. On February 7 we got an acknowledgement, that the findings are being investigated.

After checking in again on how that investigation was going on March 17, we received a reply denying that any of the finding was a security vulnerability, including the first one allowing us to forge arbitrary log outputs. After some additional rounds of discussion, they stopped responding on March 20, without anything being acknowledged as vulnerability and no further action taken. At the same time as submitting this article for review we provide the same version including artifacts to the systemd security mailing list.

8 CONCLUSION

We think the security model presented by us in this paper is what a user relying on log sealing will reasonably expect. We still defend the standpoint that the presented security vulnerabilities are valid and should be addressed. Truncation resistance might be an often overlooked aspect of integrity but we believe it should be part of the security guarantee and it is included in our definition. Log sealing should detect truncation, also because it is not too difficult to achieve compared to the security benefit it brings. From a practical point of view log completeness should be further investigated for logs spanning multiple journal files.

In summary this work shows how large the gap between a security model and its implementation is sometimes, resulting in serious vulnerabilities. We improve this, by bringing more real-world complexity into the theoretical model. Further improvement could be to incorporate even more real-world artifacts in the model, or adjust the implementation to have less quirks that deviate from the model.

Acknowledgements. We thank the CCS 2023 anonymous reviewers for their constructive feedback. The work presented in this paper has been funded by the German Federal Ministry of Education and Research (BMBF) under the project "Sec4IoMT" (ID 16KIS1692) and the project "ASCOT", and by KASTEL Security Research Labs.

REFERENCES

[1] Mihir Bellare and Bennet Yee. 1997. *Forward integrity for secure audit logs.* Technical Report. Citeseer.
[2] Mihir Bellare and Bennet Yee. 2003. Forward-Security in Private-Key Cryptography. In *Topics in Cryptology — CT-RSA 2003*, Marc Joye (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–18.
[3] Peter Gutmann. 1996. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, Vol. 14. 77–89.
[4] Gunnar Hartung. 2016. Secure Audit Logs with Verifiable Excerpts. In *Topics in Cryptology - CT-RSA 2016*, Kazue Sako (Ed.). Springer International Publishing, Cham, 183–199.
[5] Gunnar Hartung, Björn Kaidel, Alexander Koch, Jessica Koch, and Dominik Hartmann. 2017. Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures. In *Provable Security*, Tatsuaki Okamoto, Yong Yu, Man Ho Au, and Yannan Li (Eds.). Springer International Publishing, Cham, 87–106.
[6] Viet Tung Hoang, Cong Wu, and Xin Yuan. 2022. Faster Yet Safer: Logging System Via {Fixed-Key} Blockcipher. In *31st USENIX Security Symposium (USENIX Security 22)*. 2389–2406.
[7] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. 1997. HMAC: Keyed-Hashing for Message Authentication. RFC 2104. https://doi.org/10.17487/RFC2104
[8] Giorgia Azzurra Marson and Bertram Poettering. 2013. Practical secure logging: Seekable sequential key generators. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*. Springer, 111–128.
[9] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. 2020. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1551–1574.

A MESSAGE AUTHENTICATION SCHEME

Definition A.1 (Message Authentication Scheme MAS *[2]).* The interface of a MAS is given by a set of three PPT algorithms (KeyGen, Sign, Vfy):

$$\begin{aligned} \texttt{KeyGen}: &\quad (1^\lambda) \mapsto (k) \\ \texttt{Sign}: &\quad (k, msg) \mapsto (tag) \\ \texttt{Vfy}: &\quad (k, msg, tag) \mapsto \{0,1\}. \end{aligned}$$

We expect a MAS to fulfill the notion of correctness, i.e. that whenever $(k) \leftarrow \texttt{KeyGen}(1^\lambda)$, then $\texttt{Vfy}(k, msg, \texttt{Sign}(k, msg)) = 1$.

$\texttt{Exp}^{\texttt{EUF-CMA}}_{\texttt{MAS},\mathcal{A}}$

(1) $(k) \leftarrow \texttt{KeyGen}(1^\lambda)$
(2) $(msg^*, tag^*) \leftarrow \mathcal{A}^{O_{\texttt{Sign}}(k,\cdot)}(1^\lambda)$
  - $List \leftarrow List \cup msg$
  - $\mathcal{A}: (tag) \leftarrow O_{\texttt{Sign}}(k, msg)$:
(3) Return 1 iff $msg^* \notin List$ and $\texttt{Vfy}(k, msg^*, tag^*) = 1$, else 0.

Figure 9: The EUF-CMA **security experiment for** MAS**.**

*Definition A.2 (*EUF-CMA *of* MAS*).* A MAS is existentially unforgeable under chosen message attacks (EUF-CMA) if for any PPT adversaries $\mathcal{A}_{\texttt{EUF-CMA}}$ the advantage to win the EUF-CMA game shown in Fig. 9 is negligible in λ.

B AN EXEMPLARY JOURNAL FILE

```
Header:
All Entries Array: EntryArray 1

FieldHashTable:
MESSAGE: Field 2
_SOURCE_REALTIME_TIMESTAMP: Field 1
DataHashTable:
MESSAGE=Journal started: Data 2
PRIORITY=6: Data 3
....

Tag 1(seqnum: 1, epoch: 0)

Data 1(_SOURCE_REALTIME_TIMESTAMP=1686130981246175, First
    Entry: Entry 1, NextField: Data 4)
Field 1(_SOURCE_REALTIME_TIMESTAMP, HeadData: Data 1,
    NextHash=Field 3)
Data 2(MESSAGE=Journal started, First Entry: Entry 1,
    NextField: Data 5)
Field 2(MESSAGE, HeadData: Data 2)
Data 3(PRIORITY=6, First Entry: Entry 1, More Entries:
    EntryArray 2, NextHash=Data 5)
Field 3(PRIORITY, HeadData: Data 3)

Entry 1(Data 1, Data 2, Data 3)
EntryArray 1(Entry 1, Entry 2, Entry 3)
EntryArray 2(Entry 2, Entry 3)

Data 4(_SOURCE_REALTIME_TIMESTAMP=1686130981268374,
    NextField: Data 6)
```

```
Data 5(MESSAGE=Hello World, NextField: Data 7)

Entry 2(Data 4, Data 5, Data 3)
Tag 2(seqnum: 2, epoch: 0)

Data 6(_SOURCE_REALTIME_TIMESTAMP=1686130981302838)
Data 7(MESSAGE=Hello World after Epoch)
Entry 3(Data 6, Data 7, Data 3)
```

This textual representation of a simplified journal file shows the different data structures contained. First note the 3 `EntryObjects`. They encode 3 log messages with different `MESSAGE`, `_SOURCE_REALTIME_TIMESTAMP` and the same `PRIORITY`. Note that all 3 entries therefore link to the same `DataObject` (Data 3), and this priority is only stored once. There is a `FieldObject` for each of the different names. The `FieldObjects` link to the start of a linked list (using the `NextField` property) of all their `DataObjects` with their `HeadData` property. We have two `EntryArrays`. The first one is a list of all `EntryObjects` contained in the journal. The second one is a list of all referencing Data 3. Note that the first entry is stored inside the `DataObject` and `DataObjects` with only one entry do not have an `EntryArray`.

The two HashTables at the beginning provide quick access by data to all existing `FieldObjects` and `DataObjects`. Hash collisions between `FieldObjects` (like between Field 1 and Field 3) are resolved as linked lists with the `NextHash` property. Similarly hash collisions between `DataObjects` (like Data 3 and Data 5) are resolved with a `NextHash` field there.

The first `TabObject` is inserted directly under the header. Another one is inserted after the corresponding `Entry`. They share the same epoch and thereby the same key, as they have been created in the same sealing interval.