Securing Operating System Audit Logs

Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller
Department of Computer Science, College of Engineering, University of Illinois at Urbana-Champaign

INTRODUCTION

Audit logs provide definitive ground truth of a system's activities, making them crucial to detect and explain system intrusions.

```
type=SYSCALL msg=audit(1364481363.243:24287): arch=c000003e syscall=2
success=no exit=-13 a0=7fffd19c5592 a1=0 a2=7fffd19c4b50 a3=a items=1 ppid=2686
pid=3538 auid=500 uid=500 gid=500 euid=500 suid=500 fsuid=500 egid=500 sgid=500
fsgid=500 tty=pts0 ses=1 comm="cat" exe="/bin/cat"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key="sshd_config"
type=CWD msg=audit(1364481363.243:24287):  cwd="/home/shadowman"
type=PATH msg=audit(1364481363.243:24287): item=0 name="/etc/ssh/sshd_config"
inode=409248 dev=fd:00 mode=0100600 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:etc_t:s0
```

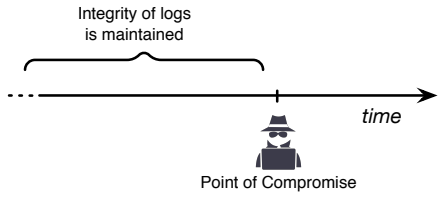
Examples of Linux Audit logs

However, malware can engage in *anti-forensic* activity after an intrusion, including modification and deletion of the logs to cover their tracks.

We need methods to guarantee the integrity of system audit logs in commodity operating systems.

GOALS

Our solution must guarantee tamper-evidence, meaning that that logs generated before system compromise cannot be falsified by the attacker without that being detectable.

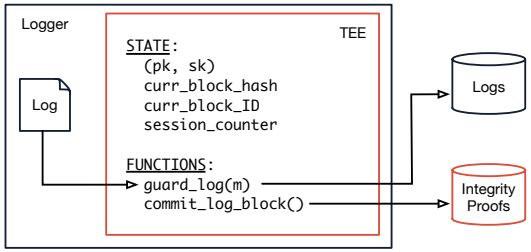


Further, to be viable within commodity operating systems, our solution must be **transparent, efficient, and minimally invasive to existing audit frameworks**.

Finally, log verifiability should not depend on a fully trusted verifier or remote server (single point of failure).

SYSTEM DESIGN

Our system leverages features of increasingly available trusted execution environments (TEE) to enable transparent tamper-evident logging.



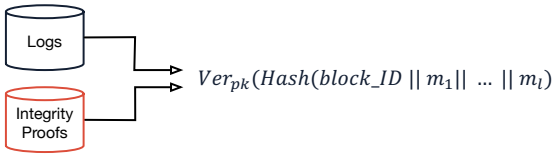
It employs a parameterizable batch signature scheme, signing log events in blocks.

Before a message is logged, the logger passes it to TEE, where the current block's hash value is extended with the message.

When a block is ready to be committed, the TEE uses its private key to sign over the block's incremental hash and its ID, and then releases the resulting digital signature σ to the untrusted environment.

$$\sigma = Sig_{sk}\,(Hash(curr\_block\_ID \,||\, m_1 ||\, \ldots \,||\, m_l)$$

These signatures can be verified to attest log integrity:



To provide continuity across power cycles, the TEE seals its state at shutdown and unseals it at startup. At each startup the session monotonic counter is incremented.

In a centralized logging framework, blocks can be committed upon each log transmission to the central server.

SECURITY ANALYSIS

Removing a subset of events for any block will invalidate its integrity proof, which cannot be forged.

Similarly, an attacker cannot insert or modify events into a committed log block without invalidating its integrity proof.

An attacker also cannot remove or reorder an entire log block because that will invalidate the chronologically-ordered sequence of block IDs.

Killing the logger process to prevent a block's commitment will be detected from the missing sealed state at the next startup (session not terminated correctly).

Similarly, rollback attacks will fail during the startup.

PRELIMINARY EVALUATION

We implement a prototype version of our solution for the Linux Audit system that uses Intel SGX as a TEE.

We measure the average overhead of `guard_log` when processing 40,000 identical log events:

Default Logger	Modified Logger	Overhead
5.67μs	6.61μs	0.9μs (16.6%)

FUTURE WORK

- Measure the system-wide impact of our solution
- Develop strategies to prevent log tampering

ILLINOIS

Poster: Securing Operating System Audit Logs

Riccardo Paccagnella*, Pubali Datta*, Wajih Ul Hassan*, Adam Bates*, Christopher Fletcher*, Andrew Miller*

*University of Illinois at Urbana-Champaign
{rp8,pdatta2,whassan3,batesa,cwfletch,soc1024}@illinois.edu

***Abstract*—System auditing mechanisms are an important concern when investigating and responding to security incidents. Unfortunately, attackers regularly engage in *anti-forensic* activities after an intrusion, erasing or falsifying system logs to cover their tracks. While a variety of tamper-evident logging solutions have appeared in the industry and the literature, these techniques have seen limited usage because of impractical limitations such as requiring expensive cryptographic computations, fully trusted servers or specialized no-rewrite storage. In this poster, we introduce a novel framework for scalable tamper-evident system logging. Our system leverages features of increasingly available trusted execution environments (TEE) to enable the verification of log integrity while being minimally invasive to the underlying logging framework. We further present a security analysis of our system and evaluate its performance overhead on the standard Linux's audit framework.**

I. Introduction

Logging is an essential component of building and maintaining secure systems. When suspicious events occur, audit logs are frequently turned to as a definitive ground truth of the system's activities. Unfortunately, system intruders also understand the value of a system's audit logs—because audit logs describe an attacker's method of entry, mission objectives, and further propagation within a system, attackers regularly engage in *anti-forensic* countermeasures to erase or conceal this vital forensic evidence. Penetration testing tools such as Metasploit [3] go so far as to automate this process, allowing an intruder to erase audit records from a variety of sources with a single command. Perhaps worse than log removal, attackers may also edit existing events or insert new ones so as to inject confusion into subsequent investigations.

In light of this reality, it is surprising that commodity operating systems offer no special protections for their logging frameworks. To solve this problem, prior solutions relied on specialized Write-Once-Read-Many (WORM) storage, remote trusted servers, or expensive cryptography. However, a lack of widespread adoption indicates that these approaches are inapplicable for the demands of operating system logging.

In this poster, we revisit the goal of tamper-evident logging within the context of standard operating system abstractions. To this end, we introduce a scalable and practical tamper-evident framework. Our framework leverages features of increasingly-available Trusted Execution Environment technologies to record logs in such a way that they cannot later be falsified without detection. We analyze the security of our system by showing how it can always detect log integrity violations. Finally, we implement a prototype of our system and evaluate its performance to show how log integrity is provided with low computational overhead for the logger.

II. System Design

The goal of our work is to store logs in a way that they cannot be falsified without detection. To this end, our system generates integrity proofs over the logs as they are processed by the underlying audit framework. These proofs are digital signatures stored to disk altogether with the log. In particular, our protocol consists of five routines: (1) Initialization; (2) Startup; (3) Logging; (4) Commitment; and (5) Shutdown. Each of these routines corresponds to a call to a function running inside the TEE, which is trusted to confidentially store cryptographic keys. To overcome the performance limitations of digital signatures, our protocol employs a parameterizable batch signature scheme, signing log events in blocks.

a) Initialization phase: The initialization phase is used when our system is first deployed on the host. This phase starts with creating an asymmetric key pair for the TEE. Next, the TEE is used to initialize a new monotonic counter, and a block ID variable e is initialized to zero. The TEE is then used to seal these values so that they can be securely stored on disk; they will be unsealed in the startup phase.

b) Startup phase: The startup phase is invoked once per host startup. This phase starts with unsealing the previously sealed key, monotonic counter, and block ID. In case of any missing or corrupted data, an error is raised and the enclave is forced to run a new initialization routine. The system then increments the value of the monotonic counter to mark the beginning of a new session. Finally, the block ID e is incremented and an incremental hash for its corresponding block is initialized. The TEE maintains the key, monotonic counter, block ID, and incremental hash in its protected memory throughout the system execution until the shutdown phase is invoked. Once this phase is complete, the system is ready to receive log events from the underlying audit framework.

c) Logging phase: The logging phase is invoked whenever the audit framework produces a new log event. This phase consists of extending the current block's hash value with the new log event. Performance is paramount in this routine since it is invoked with high frequency.

d) Commitment phase: The commitment phase is invoked to generate an integrity proof over the current block of logs e. In this phase, the TEE uses its private key to sign over e's incremental hash, and then releases the resulting digital signature to the untrusted environment. Finally, a new block with ID $e + 1$ is started.

e) Shutdown phase: Upon receiving a shutdown notification, the system must complete the current block and seal the current block ID e together with the current value and ID

of the monotonic counter. This phase ensures that (1) all log entries up to the moment of shutdown are successfully signed and (2) when the system is started up again it can continue with block ID $e + 1$.

A. Verifying log integrity

Once a log block M with id e and spanning a set of consecutive log entries $\langle m_1, ..., m_l \rangle$, is committed and its corresponding signature σ is released by the logger, a verifier can attest its integrity as follows. Verifying the integrity of logs in M consists of recomputing the hash value $hash = H(\langle e \| m_1 \| ... \| m_l \rangle)$ and then using the host's public key pk to verify that σ is a valid signature for *hash*.

III. Security Analysis

We now analyze the security of our tamper-evident logging mechanism. Let t be the time of system compromise. We discuss our system protects against various tampering techniques an adversary may attempt to use, on the recorded logs or on our system itself, to covertly hide traces of their compromise from the logs recorded at any time $\leq t$.

- *Log Deletion*. An attacker cannot delete arbitrary events from a node's log records. Removing a subset of events for a block will invalidate its integrity proof, which cannot be forged. An attacker also cannot remove an entire log block because that will invalidate the chronologically-ordered sequence of block IDs. A truncation attack on the log will similarly be detected by validating the sequence of block IDs against a new block committed by the logger.
- *Log Insertion / Modification*. An attacker cannot insert or modify events into a committed log block without invalidating its integrity proof. An attacker also cannot re-order blocks because that will still invalidate the chronologically-ordered sequence of block IDs.
- *Protocol Termination*. An attacker with root privilege is able to terminate the logger process at any time. An attacker may use this ability to try to prevent a block's commitment by terminating the logger, but will then need to restart the logger so it can respond to future challenges. Because the current block is committed during the logger's shutdown phase, the attacker will have to force kill the process. However, if the attacker does so then there will not be sealed data on disk that will unlock to the TEE's current state. Because the attacker cannot forge such data, the logger will detect such tampering and raise an error. By the same logic, the attacker is also unable to launch rollback attacks while the logger is shutdown because this will cause parameter unsealing to fail during the Startup Phase.

IV. Evaluation

a) Setup: We implement our system on top of the Linux Audit project using Intel SGX as a TEE, and configure Linux Audit to log all system calls that are of potential interest for forensic analysis. We measure the performance of our implementation on a bare metal server-class machine with an Intel Core i7-7700K processor at 4.20GHz (4 hyper-threaded cores) and 64GB RAM.

TABLE I. Microbenchmarks on Logger operations. We run each phase 100 times and report the median execution time.

Phase	Time
Initialization	94.55 ms
Startup	109.10 ms
Logging (ecalls)	4.71 μs
Logging (Hotcalls)	0.92 μs
Commitment	128.87 μs
Shutdown	188.98 μs

b) Results: We manually invoke each logging routine 100 times and include in the measurement the time required to context switch into and out of the enclave. Table I shows the results. The phases that involve interaction with a hardware counter, Initialization and Startup, are the most costly. This is because SGX monotonic counter operations are notoriously slow [2], but these operations typically occur only once per session. The next most costly phases, Commitment and Shutdown, involve cryptographic signatures. However, these operations will occur orders of magnitude less frequently than the Logging operation. Fortunately, Logging is the most efficient at $4.71\mu s$ per log event generated. Furthermore, using Hotcalls [4] results in a significant speed-up (0.92 μs).

To evaluate how our modifications to Linux Audit affect the time required to record a single log event, we also instrument `auditd` to measure the average times in nanoseconds that both Vanilla and modified Linux Audit take to process a single event. After observing the processing of 40,000 identical log events, we discover that the modified `auditd` takes an average of 6.61μs/event whereas Vanilla `auditd` takes an average of 5.67μs/event. Our modifications thus impose an average 16.6% overhead on `auditd`. This 0.9μs overhead compares favorably to prior work on SGX-based logging, which reports a median overhead of 215μs per event [1]. We conclude that our tamper-evident logging protocol imposes reasonable overheads on Linux Audit's log processing time.

V. Conclusion

In spite of the central importance of system logs in responding to modern security incidents, today's commodity operating systems fail to assure for the integrity of logs beyond the use of typical user space access controls. In this poster, we introduced a viable approach to tamper-evident logging that supports standard operating system abstractions. Our system can be integrated with enterprise logging framework to detect the anti-forensic activities of system intruders.

References

[1] V. Karande, E. Bauman, Z. Lin, and L. Khan, "SGX-Log: Securing system logs with SGX," in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2017.

[2] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *Proc. of the USENIX Security Symposium (USENIX)*, 2017.

[3] Rapid7, "Metasploit, the world's most used penetration testing framework," https://www.metasploit.com/, last accessed 11-06-2018.

[4] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with HotCalls: A fast interface for sgx secure enclaves," in *Proc. of the Annual International Symposium on Computer Architecture (ISCA)*, 2017.