Logging to the Danger Zone: Race Condition Attacks and Defenses on System Audit Frameworks

Riccardo Paccagnella
University of Illinois at Urbana-Champaign
rp8@illinois.edu

Kevin Liao
University of Illinois at Urbana-Champaign
kliao6@illinois.edu

Dave Tian
Purdue University
daveti@purdue.edu

Adam Bates
University of Illinois at Urbana-Champaign
batesa@illinois.edu

ABSTRACT

For system logs to aid in security investigations, they must be beyond the reach of the adversary. Unfortunately, attackers that have escalated privilege on a host are typically able to delete and modify log events at will. In response to this threat, a variety of secure logging systems have appeared over the years that attempt to provide tamper-resistance (e.g., write once read many drives, remote storage servers) or tamper-evidence (e.g., cryptographic proofs) for system logs. These solutions expose an interface through which events are *committed* to a secure log, at which point they enjoy protection from future tampering. However, all proposals to date have relied on the assumption that an event's *occurrence* is concomitant with its *commitment* to the secured log.

In this work, we challenge this assumption by presenting and validating a race condition attack on the integrity of audit frameworks. Our attack exploits the intrinsically asynchronous nature of I/O and IPC activity, demonstrating that an attacker can snatch events about their intrusion out of message buffers *after* they have occurred but *before* they are committed to the log, thus bypassing existing protections. We present a first step towards defending against our attack by introducing KennyLoggings, the first kernel-based tamper-evident logging system that satisfies the *synchronous integrity* property, meaning that it guarantees tamper-evidence of events upon their occurrence. We implement KennyLoggings on top of the Linux kernel and show that it imposes between 8% and 11% overhead on log-intensive application workloads.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Malware and its mitigation**.

KEYWORDS

system auditing; race conditions; operating systems; Linux kernel; digital forensics; tamper-evident logs; forward security

ACM Reference Format:
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 (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372297.3417862

1 INTRODUCTION

System auditing is an essential tool when responding to security incidents. As cyber-attacks become increasingly sophisticated and hard to defend against, auditing technology is experiencing a renaissance, with defenders seeking out new means (e.g., [42, 45, 59, 64, 68]) of investigating and recovering from potential threats [13, 58, 74, 98, 125]. Indispensable to all of these advancements are system logs, which record the history of system execution. Recent reports reveal that 75% of incident response specialists consider logs to be the most valuable artifact during an investigation [115] and that the global log management software market is a multi-billion dollar one, growing at a steady rate [48, 106].

Lost in this optimism is the reality that attackers have long known the value of system logs, which contain highly-incriminating evidence of their methods of intrusion, privilege escalation, and ultimate objectives within the system. Unsurprisingly, attackers regularly engage in *anti-forensic* activities to cover their tracks, including erasure and manipulation of system logs [112]. Such capabilities do not exclusively belong to nation-state adversaries [14]; in fact, log tampering is within reach for any would-be intruder that can read an instructional blog post [9, 86], launch penetration testing tools like Metasploit [96], or download a simple script [21, 36, 38, 71]. Case in point, log tampering is reported as the top evasion tactic by an increasing 87% of incident response specialists [115].

To solve this problem, a variety of secure logging solutions have appeared throughout the industry and the literature. Commercial solutions typically rely on trusted storage devices, such as Write Once Read Many drives [4, 47, 70, 83, 107] and centralized log servers [19, 46, 55, 105, 110, 114]. State-of-the-art systems based on cryptography generate *tamper-evident* proofs for the logs, protecting the signing key through forward security [25, 73], or trusted hardware [54, 89]. All of these solutions expose an interface through which individual log events are at some point *committed* to a secure log, at which point they enjoy protection from future tampering. Such approaches raise the degree of difficulty of successfully launching a covert attack by guaranteeing that all events committed prior to compromise cannot be undetectably tampered with by an adversary. However, when considering a sophisticated adversary in possession of valuable zero-day kernel exploits, are such countermeasures truly sufficient to deter an anti-forensic attack?

In this paper, we answer this question in the negative. First, we observe that there exists a window of time before commitment during which log events are still vulnerable to tampering. The fundamental reason for this vulnerability is that existing proposals operate asynchronously, making the implicit assumption that an event's *occurrence* is concomitant with its *commitment* to the secured log. That is, these designs do not consider the window of time in which an event (e.g., a system call) has been permitted to proceed but whose record has not yet been committed and is therefore still vulnerable to tampering. In fact, due to the pervasive presence of asynchronous message passing in software, such windows are ubiquitous in commodity systems. We empirically validate the presence of this "Danger Zone" on Linux Audit and discover that the time between an event's occurrence and its commitment is often non-negligible and grows with the system load.

Second, we discover that this vulnerability enables a race condition attack that can be used by an adversary to undetectably intercept and suppress "in-flight" log events, effectively concealing all evidence of their intrusion before it is committed to the log. To this end, we develop a *log-interceptor* tool that seizes control of the kernel log buffer to remove compromise-related events *after* they have occurred (granting root access to the adversary), but *before* they are processed and committed by the logging framework. We concretely validate the feasibility of this attack in both a local and a remote adversarial setting. While the efficacy of our attack relies on a small backlog of existing events in the kernel buffer, we find that it is trivial for attackers to induce such load innocuously.

In contrast, defending against the race condition attack is nontrivial. Retrofitting commercial solutions (i.e., writing events to trusted storage devices) by ensuring that each event is stored before any other events can occur would prevent our attacks. However, synchronous storage of events to the log would lead to prohibitively large latencies, especially on modern hosts which can generate millions of system calls per second [24]. As an alternative, we explore the possibility of synchronous event commitment through cryptographic schemes. That is, we set out to determine if it is practical to cryptographically commit events as they occur in the kernel, rather than when they are processed and stored by the user space audit daemon [25, 54, 73, 89]. This comes with new challenges. First, as multiple system calls can execute concurrently, ordering and synchronization issues arise that existing buffered designs do not have to deal with. Second, the performance requirements of such a design are stricter, as adding operations to the critical path of system call execution affects all applications running on the host.

We successfully address these challenges by presenting KennyLoggings, the first tamper-evident logging system that satisfies the *synchronous integrity* property, meaning that it guarantees tamper-evidence of log events upon their *occurrence*. KennyLoggings solves the ordering issues by running its cryptographic commitment operations within the existing critical section used by the operating system to prevent concurrent access to the kernel log buffer; it solves the performance issue by using an efficient forward secure message authentication scheme, concretely instantiated with

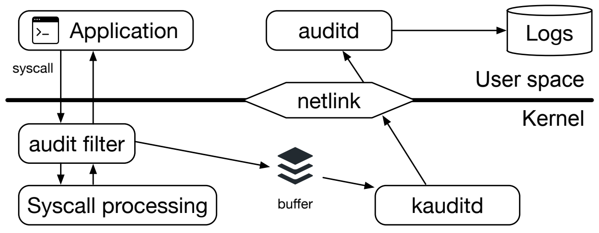


Figure 1: System-level architecture of Linux Audit. Audit filters hook syscalls and enqueue records of their execution to a buffer. These records are then processed one at a time by `kauditd`, which sends them from the kernel to user space, where they are recorded to disk.

fast cryptographic primitives (SipHash [5] and BLAKE2 [6]). Our design also includes an optimization that reduces logging latency by precomputing signing keys in batches. As a result, our approach is able to scale to meet the demands of commodity operating systems.

In summary, the contributions of this paper are as follows:

- We identify a design vulnerability overlooked by existing secure logging systems that allows an attacker to bypass their safeguards. We empirically validate the presence of this "Danger Zone" on Linux Audit, finding that there exists a non-negligible window of time between an event's occurrence and its commitment during which the event is vulnerable to tampering.
- We present a race condition attack on audit frameworks that exploits the above vulnerability to snatch all evidence of an intrusion out of kernel memory before it is committed. Unlike existing race condition attacks on audit systems [119], our attack can tamper with events that have already occurred and independently of the presence of concurrency bugs. We implement our attack into a *log-interceptor* tool and empirically demonstrate its practicality both in a local and in a remote adversarial setting.
- We introduce KennyLoggings, the first kernel-based tamper-evident logging system that provides the property of synchronous integrity. Our system cryptographically commits events upon their occurrence, guaranteeing tamper-evidence for all the events that lead up to full system compromise. As such, KennyLoggings can give investigators insight into the window of time during which existing secure logging systems are vulnerable.
- We implement KennyLoggings on the Linux kernel[1] and provide an evaluation of its performance. Our results show that KennyLoggings imposes 8% to 11% overhead on realistic application workloads that generate large volumes of log events. We also discuss how KennyLoggings can be extended to work in conjunction with asynchronous secure logging systems and identify directions for future performance improvements.

2 BACKGROUND

System Logging. This paper discusses anti-forensic strategies to compromise a host without leaving traces in the *system logs.* System logs, sometimes referred to as *audit logs* [81], provide a chronological record of all the activities that have affected an operating system (OS). Unlike application logs, which are generated

[1]Our prototype is publicly available at https://bitbucket.org/sts-lab/kennyloggings.

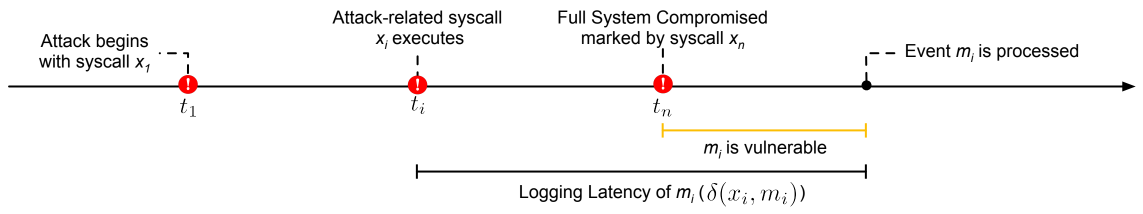

Attack begins with syscall x_1
Attack-related syscall x_i executes
Full System Compromised marked by syscall x_n
Event m_i is processed
t_1
t_i
t_n
m_i is vulnerable
Logging Latency of m_i ($\delta(x_i, m_i)$)

Figure 2: Audit records describing an attack are vulnerable due to the latency introduced by asynchronous logging. In this timeline, the log record m_i is vulnerable because the time between x_i's execution and m_i's processing ($\delta(x_i, m_i)$) extends past the moment that the attacker fully compromises the system.

by user space code, system logs are generated by the OS based on customizable rules defined by a system administrator. They include records at the granularity of system calls (which we will also refer to as *syscalls*) and are thus capable of watching file access, recording user commands, and monitoring security events and network access. Several security-related certifications require storing system logs for compliance [26, 49, 94]. For the Linux project, auditing subsystems were originally introduced as part of broader efforts to achieve certification under common criteria such as the Controlled Access Protection Profile (CAPP) [82], leading to the introduction of the Linux Audit Subsystem (LAuS) in Linux 2.6 [108].

Linux Audit. Linux Audit is Linux's standard system log collection framework [94]. Its architecture (shown in Figure 1) consists of two main parts: a kernel component (`kauditd`), and a user space component (`auditd`). When Linux Audit is enabled, every syscall passes through a kernel audit filter that decides if it needs to be logged based on the Audit rule configuration. Then, if it needed to be logged, a log event is created and enqueued to a buffer. The kernel component `kauditd` dequeues events from such buffer and sends them to the user space component `auditd`, which creates entries in the log file. For performance reasons, `kauditd` processes events asynchronously: in the kernel control path[2] log messages are only enqueued to the kernel buffer and then later processed, *one by one and in a first-in-first-out (FIFO) manner*, by `kauditd`. When this backlog of events (which is of configurable capacity) is full, the kernel can be configured to drop new events until the buffer has space (keeping a counter of the number of events lost) or to handle the error by, for example, causing a kernel panic. Netlink is used as the transmission channel between `kauditd` and `auditd`. While this paper will be evaluated on Linux Audit, asynchronous logging mechanisms are ubiquitous and apply to system logging frameworks on both Linux [8, 69, 109] and Windows [66, 75, 76].

3 THE DANGER ZONE: ASYNCHRONOUS LOGGING

In this section, we present and characterize a race condition vulnerability on operating system logging frameworks. Let us assume that an administrator has configured the logging framework to record a variety of syscalls such as process `exec` and file `open`. We consider an adversary that is capable of entering the machine, escalating privilege, and ultimately engaging in anti-forensic measures to conceal evidence of the attack after fully compromising the system. Wary of this threat, the administrator has taken the precaution of using a tamper-evident or tamper-resistant logging framework [25, 54, 73, 89]. Thus, the administrator believes that critical records about the attack, from the initial intrusion up to the privilege escalation, can be audited. To reflect that a secure logging safeguard is in place, we will assume that a log event is "safe" the moment it is ingested for processing by the user space audit daemon. This assumption is conservative in that it does not consider the overheads and latencies required to commit the event to the log in a tamper-evident or tamper-resistant fashion. Thus, in practice, the window for the vulnerability we describe is larger.

3.1 Race Condition Vulnerability

For performance reasons, asynchronous message passing paradigms are pervasively used in operating systems, including their logging frameworks. When a process invokes a syscall, the kernel control path traverses an audit filter that creates a record of the syscall and enqueues it to a buffer of log events. A separate kernel thread is responsible for transmitting log events from the buffer to user space for processing and storage. An example of this workflow is given for Linux Audit in Figure 1. It is almost essential that this workflow be asynchronous, as synchronous transmission of events at a syscall-by-syscall granularity would be prohibitively expensive.

We observe that asynchronous logging creates a race condition vulnerability—actions associated with an ongoing attack are permitted to proceed before their events are securely processed by the logging framework and recorded to the log. We formally define this vulnerability with respect to the timeline illustrated in Figure 2. Let us consider an individual attack-related action, the system call x_i, which occurs at time t_i. Syscall x_i is just one action in a sequence of attack-related actions between the beginning of the attack at t_1 and the full compromise of the system at t_n. The *logging latency* for this system call, denoted $\delta(x_i, m_i)$, is the time between action x_i and its log event m_i being ingested by the user space audit daemon. If $t_i + \delta(x_i, m_i)$ exceeds t_n (i.e., the time at which m_i is logged comes after the time of full compromise), then m_i is vulnerable to tampering. In fact, if m_i is vulnerable to tampering, then any events generated from actions between time t_i and t_n are also vulnerable.

3.2 Vulnerability Characterization

A variety of factors might affect whether or not an *attack trace* (i.e., the sequence of log events recorded from t_1 to t_n) is vulnerable. It may be that, in practice, the logging latency is so negligible that all events in the attack trace are secure. This may be likely in the case

[2]We refer to *kernel control path* as the code executed in kernel mode to handle a syscall.

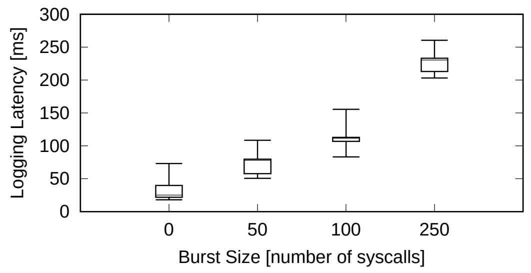


Figure 3: Latency to log a single event under different system loads. The results are computed over 200 repetitions. The logging latency grows larger when the system load is larger.

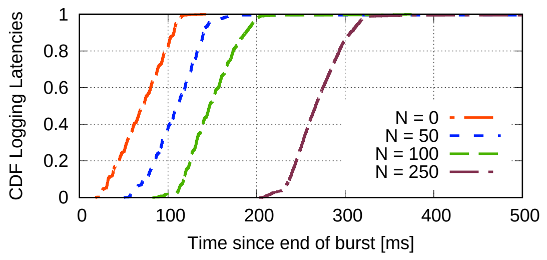


Figure 4: Latency to log 100 events under different loads. The results are aggregated over 200 repetitions. With a burst of 100 events, only 1% of the log trace was logged within 95 ms.

that the machine is in a quiescent state. Alternatively, at the start of the attack there may be unrelated log events in the kernel buffer due to other system activity. Since the events in the buffer are processed by the logging framework in a FIFO manner, these unrelated events could create a backlog that increases the logging latency for events in the attack trace. Among the factors that affect logging latency, we hypothesize that system load is the most impactful; in this section, we experimentally validate this hypothesis by characterizing how the logging latency changes under different system loads.

3.2.1 Experimental Methodology. We perform our experiments on Linux Audit, the standard system logging framework on Linux (cf. Section 2). We instrument the Linux kernel to record the execution time of each system call and `auditd` to record the time its associated log event is received for processing in user space, allowing us to conservatively calculate $\delta(x_i, m_i)$. We model the system load as a *burst* of N system calls executed serially by a script with configurable N. We then measure the logging latency $\delta(x_i, m_i)$ for events executed after a burst. The purpose of the burst is to model the fact that the logging buffer may be non-empty when the adversary begins their attack. Because we are only interested in a snapshot of the system state immediately before the attack, it is immaterial *how* the buffer is filled (whether by a burst of system calls or by events accumulated over time). To simplify our experimental design and to better isolate our experiments from system factors we cannot control, we thus choose to model the system load as a burst.

3.2.2 Experimental Setup. We run our experiments on a bare-metal server with 8 logical CPU cores (4.20 GHz Intel Core i7-7700K) and 64 GB of RAM, running Ubuntu Server 16.04 64 bit (Linux 4.4.0-116). We configure Linux Audit to log all forensically relevant system calls (using the ruleset employed in [32, 60, 68, 89]), and use a large buffer capacity of 2^{20} events. For our burst implementation, we use the system call `getuid`. We use `getuid` because it is a low-latency syscall—it is non-blocking (i.e., it does not wait on a response from the disk or network) and it also does not have any arguments that need to be processed, minimizing the time between the syscall invocation and a new log event being created. Further, `getuid` generates short log records, making our analysis of logging latency conservative for a given burst size.

3.2.3 Experimental Results. Our first experiment measures $\delta(x_i, m_i)$ for a single system call x_i after a burst of size N. We execute a burst of N consecutive system calls followed by x_i, and then measure the latency $\delta(x_i, m_i)$ of logging x_i. The results are shown in Figure 3. In idle conditions ($N = 0$), the median logging latency is 25 ms. After a burst of size $N = 100$, the median logging latency is 111 ms. Most notably, after a burst of size $N = 250$, the median logging latency is 230 ms. These results confirm our hypothesis that the logging latency grows with the system load.

Next, we measure $\delta(x_i, m_i)$ for an attack trace of K system calls executed after a burst of size N. That is, we execute a burst of N consecutive system calls followed by K system calls $\langle x_1, ..., x_K \rangle$, and then measure the logging latency $\delta(x_i, m_i)$ for each $1 \leq i \leq K$. In particular, we pick $K = 100$ to model an attack trace composed of 100 system calls. Figure 4 reports the aggregated results over 200 runs. Under idle conditions ($N = 0$), 82% of the attack trace was logged within 100 ms. Conversely, when the burst size was $N = 100$, less than 2% of the attack trace was logged within 100 ms.

The cause of these behaviors is the larger backlog of events that are queued in the buffer after a longer burst. That is, since the events in the buffer are processed one by one and in a FIFO manner, the latency of logging new events will be affected by the number of events in the buffer. We established this by measuring the size of the logging buffer immediately after executing bursts of different sizes and observed that it was in fact larger when N was larger. This means that the rate at which events are appended to the buffer during a burst is larger than the rate at which `auditd` can process them. We refer the reader to prior work for further explanation of the performance bottlenecks in Linux Audit [67].

4 RACE CONDITION ATTACKS

Recall that the goal of our adversary is to leave no traces of their intrusion in the system logs. In the previous section, we described and characterized a race condition vulnerability on system logging frameworks. We saw that if the system load is high enough, the logging latency of individual log events and of attack traces of K log events is non-negligible, which makes attack-related log events vulnerable to tampering. However, we have not yet discussed how the adversary can exploit this vulnerability in practice to intercept and suppress log events before they are processed. This operation is crucial to the intrusion and needs to be executed immediately after achieving full system compromise, to prevent the logging framework from processing any subset of the attack trace.

4.1 Intercepting a Vulnerable Attack Trace

Suppose that the adversary has just gained root access and that the attack trace is still vulnerable (i.e., its events have not been committed yet). One possible approach for the adversary to prevent the vulnerable attack trace from being logged consists of force killing the logger process immediately after full compromise. This would successfully prevent events currently in the buffer from being processed. However, previous work has shown that this type of tampering can be detected because it will force the logger to skip its shutdown routine and that will raise an alert [54, 89]. Furthermore, killing the logger process would prevent benign events from being logged as well, which would create a suspicious gap in the log file.

To remain stealthy and undetected, an astute adversary will instead attempt to intercept only log events related to the intrusion and their subsequent exfiltration, keeping the logging system running not to raise any alerts. An adversary could achieve these goals by seizing control of the kernel buffer and preventing any events describing their actions from being logged, while letting benign events proceed as normal. In the next section, we will describe and evaluate a proof-of-concept implementation of this approach.

4.2 Exploit Evaluation

We consider two attack scenarios of varying difficulty for the adversary. The first is a *local* attacker who already has non-privileged access to the system, reflecting an insider threat or an adversary who has obtained out-of-band access to credentials. We intend for this scenario to be more favorable to the adversary because the first suspicious act they commit will be privilege escalation, reducing the size (in log events) of the attack footprint. In addition, we also consider a *remote* attacker who has no prior access to the system. This attacker must establish a foothold on the machine by exploiting an Internet-facing service (e.g., a web server) and dropping to shell before escalating privilege, increasing the attack footprint.

4.2.1 Attack Implementation. We implement two existing exploits on Linux and pair them up with a *log-interceptor*, a new tool which seizes control of the kernel buffer to remove compromise-related events before they are committed by the logging framework. For the local privilege escalation, we use CVE-2017-16995 [2], which allows an unprivileged user to run code as root on the machine. For the remote code execution, we use CVE-2014-6271 [1] (also known as "Shellshock"), a vulnerability of the Unix Bash Shell which can be used to run arbitrary commands on a remote machine. The remote adversary achieves privilege escalation by combining these two exploits. That is, we let the remote exploit vector be an apache2 web server that uses Common Gateway Interface (CGI) scripts written in Bash to handle requests. The remote adversary will then issue a curl command to the web server with the exploit payload (the code for the local privilege escalation and the Bash commands to run it) embedded in the User-Agent string. As in the prior section, we model the load prior to an attack as a burst of N system calls.

We implement a proof-of-concept *log-interceptor* as a kernel module targeted against Linux Audit. When this kernel module is loaded, it performs the following operations: 1) retrieve the address of `kauditd`'s events buffer queue (this can be done using the lookup function `kallsyms_lookup_name`); and 2) empty the `kauditd` buffer queue, discarding all the events contained in it. In both of our attacks, we embed the compiled log interceptor kernel module directly in the privilege escalation binary.

Table 1: Log-interceptor results. We report the median over 100 runs. In the local attack scenario, a burst of 100 syscalls preceding the attack is always sufficient for log-interceptor to conceal the entire attack trace. In the remote attack scenario, a burst of 0 syscalls preceding the attack is nearly enough for log-interceptor to conceal the entire attack trace.

Burst Size	% Trace Intercepted (Local Attack)	SD	% Trace Intercepted (Remote Attack)	SD
0	0%	10%	99%	23%
50	40%	18%	100%	5%
100	100%	0%	100%	1%
250	100%	0%	100%	0%

4.2.2 Experimental Setup. We use the same experimental setup of Section 3.2 but replace the default Unix Bash Shell with version 4.2 (vulnerable to CVE-2014-6271) and introduce an apache2 web server (version 2.4.18) using CGI to serve requests. We ensure that the vulnerable system is idle prior to the execution of each experiment, except for the artificial bursts discussed above. We also ensure that the capacity of the kernel buffer is large enough not to fill up during our experiments, causing `kauditd`–depending on its configuration (cf. Section 2)–to drop events or to kernel panic.[3] We issue the remote attack request from a separate bare-metal server, with an average latency of 452 μs between the two machines.

4.2.3 Experimental Results. First, we measure how many log events each attack generates under our logging configuration. We observe that the local attack using CVE-2017-16995 consists of 25 forensically relevant syscalls, all generating relatively short log records (averaging 182.6 bytes). In contrast, the remote attack that uses both CVE-2017-16995 and CVE-2014-6271 consists of 176 forensically relevant syscalls, the first of which generate large log records (up to 7.6 kilobytes) related to the parsing of the User-Agent string. In receiving a CGI request, the web server also generates 60 benign syscalls prior to parsing the malicious User-Agent string. We do not consider these syscalls as part of the attack trace because they do not contain information incriminating the attacker. As a consequence, the remote attack benefits from both an intrinsic benign burst generated prior to attack execution and an increased logging latency due to the larger size of the log events to process.

Next, we analyze what percentage of each attack trace can be erased by log-interceptor under varying system loads. For the local attack, we first run serially a burst of N syscalls, the privilege escalation exploit and log-interceptor, then analyze how many of the forensically relevant syscalls of the privilege escalation attack were logged. For the remote attack, we additionally perform the remote code execution attack, and analyze how many of the forensically relevant syscalls of both the remote code execution and privilege escalation attacks were logged. Table 1 reports the median results over 100 runs for both attacks. As expected, the adversary has a greater probability of intercepting the attack trace when the system

[3]This factor is not relevant in our scenario: given the relatively small size of our attack traces (cf. Section 4.2.3), a system that is prone to filling up the kernel buffer would likely incur event loss or trigger panics independently of the presence of an attack.

load (and thus the logging latency) is larger. In the local attack, we observe that a burst of 100 syscalls is always enough for the adversary to be able to intercept 100% of the attack trace. Surprisingly, in the remote attack we find that a burst size of 0 (idle state) is sufficient for the attacker to conceal 99% of the attack trace, and a burst of 50 syscalls (which could be generated, for example, by issuing a benign request just before the malicious one) is enough to conceal 100% of the attack trace. This is because, as we noted above, the web server generated an additional burst of benign syscalls when processing web requests, further increasing logging latency.

Another way to evaluate the practicality of our attack is to ask the question "Under what system load can an attacker compromise a host without leaving any log record about the intrusion?" From our results, we conclude that a remote attacker can succeed even in the absence of additional system load. Contrary to our intuition, this suggests that remote exploitation of this vulnerability is highly practical due to logging noise generated by web servers.

5 DOES PRIOR WORK OFFER A DEFENSE?

In this section, we discuss whether existing secure logging solutions can be used to protect against the perils described above.

Commercial Solutions. In industry, log integrity is typically addressed by writing events to a trusted storage device [33]. This could be local, such as a Write Once Read Many (WORM) drive [4, 47, 70, 83], or remote, such as a centralized log server [19, 46, 55, 105, 110, 114]. On Linux, these approaches are usually implemented in user space by reconfiguring `auditd` to write events to a different interface (through, e.g., `auditspd` or `rsyslog`). However, our attack targets the kernel log event buffer and therefore preempts these protections. To defend against our attack using commercial solutions, logging would need to become a fully synchronous, proactive operation. That is, any log event m_i related to syscall x_i would need to be stored onto the trusted storage device before any other syscall x_j (with $t_j > t_i$) is allowed to proceed. However, on hosts that can produce millions of syscalls per second [24], even using the fastest communication mechanisms between the kernel and user space would cause impractically large system overheads.

Cryptographic Solutions. The literature features many cryptographic solutions to the log integrity problem [10, 11, 40, 44, 65, 99, 100, 122, 123]. The idea of these schemes is to record log messages together with integrity proofs so that any log tampering can be subsequently detected by verifying the proofs. Some of these cryptographic schemes have been deployed onto real-world logging systems such as the systemd journal [25] and syslog-ng [3, 73] (in use at Airbus [72]). Systems that combine cryptographic schemes with secure hardware have also been presented [54, 85, 89, 104]. However, to our knowledge, all these systems execute cryptographic operations in user space, asynchronously with event occurrence. As such, they are vulnerable to our race condition attack. To defend against our attack using cryptographic solutions, event commitment would instead need to become a fully synchronous, proactive operation. The main performance consideration would then become commitment speed, whose cost depends on the specific proof generation scheme deployed. Schemes based on asymmetric cryptography [40, 44, 65, 122, 123] are therefore not practical in this domain. Cryptographic data structures such as history trees [20, 95] and hash treaps [95], where insertion speed is a function of log size (i.e., non-constant complexity), can be similarly dismissed. The most efficient approaches seem to be earlier schemes based on forward secure symmetric cryptography, such as [10, 11, 44, 99, 100].

6 DEFENSE DESIGN

We now present the design of the first tamper-evident logging system that protects the logs from our race condition attack.

6.1 Threat Model and Assumptions

The adversary we consider is akin to an Advanced Persistent Threat [77]; namely, after some initial compromise or credential theft grants non-privileged access to a host, the attacker will establish persistence and then escalate privilege, ultimately leading to a *full system compromise* that grants privileged code execution. We assume that each phase of this attack requires interaction with the system's relevant software layers; as a result, the "attack footprint" will include system calls that are being traced by the system's audit framework. However, after escalating privilege, the attacker can engage in anti-forensic countermeasures such as log tampering [9, 38, 71, 86, 112, 115] to cover the tracks of their attack.

Analogous to prior work (cf. Sec. 11), we assume the presence of a trusted *auditor*, who can verify the integrity of the log after its commitment. Further, we assume that the machine under attack is trusted prior to full system compromise; that is, the machine's software and hardware are distributed and configured in a correct state. Physical attackers who tamper with the internal functionality of the machine are considered out of scope. Similarly, side channel attacks that may leak secrets about kernel memory (e.g., [62]) lie outside our threat model. We also assume that erased keys cannot be recovered after their deletion from the host's memory [121]. Lastly, we make the standard cryptographic assumptions that it is not feasible for an adversary to forge message authentication codes (MACs) or find collisions in cryptographic hash functions.

6.2 Design Goals

We set out to design a solution that satisfies the following properties:

G1 - Tamper-Evident Logs. Our system must be able to record logs with provable integrity such that forgeries, omissions, and other forms of log tampering can be detected in an audit. Specifically, if a message m_i was committed to the log at time t_i, an audit should be able to verify that no message was added, removed, or modified between times 0 and t_i. This goal is consistent with prior work, assuring that our solution exposes the same basic functionality as existing secure logging solutions.

G2 - Synchronous Integrity. Our system must assure that any *pre-compromise* event (i.e., every event that occurs prior to the moment of compromise) is associated with a tamper-evident log record. Specifically, if t_n is the moment of full system compromise (when the attacker process elevates to root), audits must be able to verify the integrity of all events x_i with $t_i < t_n$. This goal is distinct from prior work—in addition to guaranteeing the tamper-evidence upon the *commitment* of message m_i to the log, our system must guarantee the tamper-evidence of m_i upon the *occurrence* of x_i.

G3 - Tamperproof Mechanism. Our system must be secure in the face of the root-level adversary described in our threat model. Goals **G1** and **G2** must hold for pre-compromise events against an attacker that can gain unrestricted access to kernel memory.

G4 - Practical Deployability. Our system must be demonstrably efficient under realistic deployments on top of commodity audit frameworks. Further, our system must preserve compatibility with upstream log analysis applications (e.g., [105]).

6.3 Design Challenges

Protecting against our adversary is challenging without trading off performance. As we discussed above, to solve it with commercial solutions based on trusted storage devices, logging would need to become a synchronous, proactive operation. However, synchronous processing of log events would be prohibitively expensive. In Section 3.2, we measured that Linux Audit's median logging latency for a single log event is the order of milliseconds: incurring an overhead of milliseconds on the execution of each syscall (whose latency is in the order of microseconds, as we will see in Section 9) would not satisfy **G4**. An alternative, more practical defense mechanism could be to reduce the race condition rather than eliminate it. For example, the logging latency could be reduced by having multiple threads ingest logs in the user space audit daemon. However, while such a solution would raise the bar for an adversary to carry out a successful attack, it would still not fully satisfy **G2**; further, its security guarantees would depend on additional system overheads, which would not be practical for servers under high utilization, again not satisfying **G4**. When considering cryptographic approaches, it may be possible to *synchronously* commit events while still permitting *asynchronous* transmission and storage, which would be much more practical. However, implementing cryptographic proofs in the kernel's audit subsystem poses its own challenges. First, the performance requirements of such a design are stricter, as adding operations to the kernel control path affects the performance of all applications running on the host. Second, modern operating systems are multi-threaded, meaning that multiple syscalls can execute concurrently. This introduces ordering and synchronization issues that prior, buffered secure logging implementations did not have to deal with. To our knowledge, we are the first to address these challenges in the context of system logging.

6.4 In-Kernel Log Integrity

We now describe the design of KennyLoggings, the first tamper-evident logging system that protects system logs from the race condition attack described above. KennyLoggings satisfies our design goals by introducing a tamper-evidence layer that operates in the kernel control path, synchronously with syscall execution.

The key insight behind KennyLoggings is that past tamper-evident logging system designs falter because they cryptographically commit events at the moment they are stored; referring back to Figure 2, this means that commitment has traditionally occurred on the far right of the timeline when event m_i is processed by the user space audit daemon. KennyLoggings addresses this limitation by decoupling commitment and storage, instead committing events as they occur (e.g., when attack-related syscall x_i executes). Thus, while the logging latency $\delta(x_i, m_i)$ remains unchanged, an attacker attempting to exploit this race condition using the method described above will be detected during an audit.

6.4.1 *Tamper-Evident Logging Protocol.*

To satisfy **G1**, we first require a log commitment protocol that facilitates tamper-evident audits. Because our threat model considers an adversary who can fully compromise the system and hence read any keys from its memory, we need a protocol that provides *forward security*. Forward security is the property that an adversary that gains access to a cryptographic secret at time t_n will be unable to forge integrity proofs generated at times prior to t_n, which in this case means that the integrity proofs for pre-compromise log events withstand forgery attempts after compromise. Recall that corruption of pre-compromise log events is a central goal of the adversary, as these events include vital forensic evidence pertaining to the intrusion.

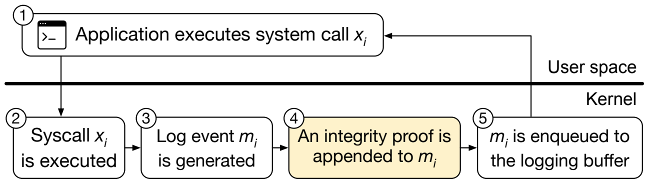
As we mentioned in Section 5, there is expansive prior work on the development of tamper-evident logging protocols. As our paramount consideration is the speed of cryptographic commitment, we select the fastest of these schemes—forward secure message authentication codes (MAC), previously used in [11, 44]. This scheme lacks many of the advancements of subsequent work (e.g., public verifiability, aggregate authentication), but its efficiency makes it ideally-suited for our use case. The protocol begins with a shared secret key, known to both the logger and the auditor. To achieve forward security, the key used by the logger to generate integrity proofs evolves over time and expired keys are securely deleted from memory. Further, the key update mechanism is a one-way function, so that an adversary who learns the current signing key cannot recover past signing keys and forge proofs for past events. Concretely, our protocol exposes the following four functions:

- $\mathsf{KeyGen}: 1^\lambda \rightarrow \{0,1\}^\lambda$. Given security parameter 1^λ, generate an initial λ-bit signing key k_1.
- $\mathsf{KeyUpdate}: \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda$. Given the i-th signing key k_i, generate the $(i+1)$-th signing key $k_{i+1} = \mathcal{H}(k_i)$, where $\mathcal{H}$ is a collision resistant one-way hash function.
- $\mathsf{Sign}: \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^l$. Given a λ-bit key k_i and an arbitrary length message m, compute a l-bit tag (integrity proof). The tag is generated using as a standard MAC algorithm.
- $\mathsf{Verify}: \{0,1\}^\lambda \times [(m_1,p_1),\ldots,(m_n,p_n)] \rightarrow \{0,1\}$. Given an initial signing key k_1 and a list of message-tag pairs, first derive the key sequence $k_1,\ldots,k_n$. If $p_i = \mathsf{Sign}(k_i, m_i)$ for each $1 \le i \le n$, then output 1. Otherwise, output 0.

Notice that, unlike [11], which permits several log entries to be committed with the same signing key, our protocol updates the key for each log event (this is necessary to achieve **G2**). Further, unlike [44], our protocol does not encrypt the log. Our reason for avoiding encryption is that it would break interoperability with upstream applications that analyze log data (e.g., [105]), not satisfying **G4**. With our solution, we are able to provide an identical external interface as the original commodity audit framework.

6.4.2 *Integration into the Operating System.*

The use of forward security for tamper-evident logging is not a new idea. What makes KennyLoggings different from prior work is that KennyLoggings instantiates its tamper-evident logging protocol in the kernel, synchronously committing log events as they occur. The general workflow for our solution is given in Figure 5, which we describe here



1
Application executes system call x_i
User space
Kernel
2
Syscall x_i is executed
3
Log event m_i is generated
4
An integrity proof is appended to m_i
5
m_i is enqueued to the logging buffer

Figure 5: KennyLoggings' workflow for committing log events, with the novel component shaded in yellow. Integrity proofs are generated in the kernel control path.

with respect to our implementation on Linux Audit: an application first invokes an (audited) syscall x_i, prompting a control transfer to kernel space (①); the kernel control path executes the syscall (②) and passes through an `audit_filter` hook (see Figure 1), creating an event m_i (③); within the audit subsystem beneath the hook, we generate a cryptographic commitment of m_i and append it to m_i (④); the extended message m_i is enqueued to the log buffer to be later consumed by the `kauditd` thread (⑤). Finally, the syscall returns control to the application, which is allowed to proceed.

Deploying KennyLoggings in a multi-threaded environment requires additional consideration. Since KennyLoggings' tamper-evident logging protocol uses a new signing key for each log event, we must enforce that each key is used only once. Further, we must ensure that the logger and the verifier agree on the ordering of signing keys and respective log events. If this property did not hold and log events were received by the verifier in an order different from the one at which they were signed, the verifier would not be able to match the key sequence and verification would fail. To ensure that each key is used only once, KennyLoggings runs its commitment operations in a critical section that assures mutual exclusion for event commitment. This critical section envelopes the existing critical section used by `kauditd` to prevent concurrent access to the log buffer and is thus a natural extension to existing audit frameworks. The advantage of this synchronization approach is that the order at which events are signed is the same as the order at which events are enqueued to the log buffer, thus serializing concurrent activity in the system. However, this approach also has potential performance challenges, as it increases the latency required for the kernel to enqueue an event to the log buffer; we will evaluate the overhead of extending this critical section in Section 9.

6.4.3 Optimization: Precomputation of Signing Keys. Forward secure message authentication codes (MAC) allow for highly efficient implementations of the Sign and KeyUpdate functions, which are crucial given that KennyLoggings inserts these into the critical section of the auditing subsystem. Nonetheless, these functions introduce non-negligible latency to log event generation, which is concerning in light of the fact that thousands of events per second can be generated during bursts of system activity. To reduce the overhead of KennyLoggings, we identify the following opportunity to optimize its performance—signing keys may be precomputed as long as they are securely deleted after use.

We extend KennyLoggings to support precomputation of keys as follows. In place of a single signing key, we introduce two sets of signing keys that are populated during system initialization. At

```
void audit_log_end(struct  audit_buffer  *ab) {
  struct  sk_buff  *skb;
  struct  nlmsghdr *nlh;
  char *log_msg;
  u64  integrity_proof ;

  if  (!ab)  return;
  if  ( audit_rate_check () )  {
    nlh  =  nlmsg_hdr(ab->skb);
    log_msg = nlmsg_data(nlh);

    /*  Enter   critical   section   */
    spin_lock_irqsave (&(&audit_queue)->lock,  flags );

    /*  Compute proof  and append it  to  the  log  event  */
    integrity_proof   =  siphash(log_msg,  strlen (log_msg),  &curr_key);
    audit_log_format (ab,  "  p=%llx",   integrity_proof );

    /*  Update  the  key  */
    blake2b (( uint8_t   *)&curr_key,  sizeof(siphash_key_t),
            ( uint8_t   *)&curr_key,  sizeof(siphash_key_t),  NULL, 0);

    skb  = ab->skb;
    ab->skb = NULL;
    nlh->nlmsg_len = skb->len − NLMSG_HDRLEN;

    /*  Enqueue the  event  to  the  logging   buffer  */
    __skb_queue_tail(&audit_queue,  skb);

    /*  Exit   critical   section   */
    spin_unlock_irqrestore (&(&audit_queue)->lock,  flags );
    wake_up_interruptible(&kauditd_wait);
  }  else
    audit_log_lost ( "rate   limit   exceeded");

  audit_buffer_free (ab);
}
```

Listing 1: KennyLoggings' changes to the `audit_log_end` function in `kernel/audit.c`. This code enqueues the syscall's log event to `kauditd`'s buffer (`audit_queue`). Blue lines were previously sequentially executed in a nested function but were moved to support the addition of the new code (in red). Error checks are omitted for brevity.

runtime, when the first set is exhausted, KennyLoggings immediately rotates to the second and requests a background thread to repopulate the empty set. The size of each set is parameterizable; in our implementation, we store 100,000 keys per set at a time. The background thread is able to execute asynchronously and does not require access to the critical section from Step ④ above in order to operate. Because in practice the precomputation of keys is much faster than KennyLoggings's consumption of keys, this optimization effectively eliminates the cost of KeyUpdate from the kernel control path. We argue for the security of this optimization in Section 8 and quantify its effect on system performance in Section 9.

6.4.4 Initialization, Verification and Shutdown. The deployment and usage models for KennyLoggings are similar to past tamper-evident logging systems. At initialization, a system administrator configures the machine with a shared secret, generated using KeyGen, that is used as the initial signing key. This secret is securely kept by the system administrator, who can later use it to perform a tamper-evident audit of the logs. During an audit, the auditor receives the sequence of signed log entries from the KennyLoggings-enabled host and uses them as input to the Verify function together with the initial shared secret. Upon system shutdown, the current signing key is sealed to the host configuration using standard TPM functionalities [104], allowing the host to unseal the key following a correct boot sequence. After the key is unsealed and loaded into kernel memory the sealed key is deleted.

7 IMPLEMENTATION

We implement KennyLoggings on the Linux kernel, version 4.15.0-47, using the Audit subsystem [94] (cf. Section 2). We concretely instantiate the tamper-evident logging protocol with fast symmetric primitives, namely the BLAKE2 cryptographic hash function [6] for KeyUpdate and the SipHash pseudorandom function [5] (specifically, SipHash-2-4, which uses 128-bit keys) for Sign. The KeyGen operation is implemented using the `get_random_bytes` function (a cryptographically secure source of randomness), and keys are securely erased by overwriting (or zeroizing, in case of precomputed keys [121]) their data structures in memory after use. Of the existing kernel source code, we modify the functions `audit_init`, which initializes the kernel components of Linux Audit at system startup, and `audit_log_end`, which is executed at the end of the kernel control path to enqueue the log event to `kauditd`'s kernel buffer if the syscall needs to be logged. The code of the modified `audit_log_end` function (without the key precomputation optimization) is shown in Listing 1. Its existing critical section, implemented using a spinlock, is extended to include the critical section of KennyLoggings. We implement the precomputation of signing keys as a separate kernel task, scheduled using a wait queue.

8 SECURITY ANALYSIS

We now consider how KennyLoggings' design and implementation assure the intended security and design goals. With respect to **G1**, observe that our signing mechanism (SipHash) is a secure message authentication code. The only way an adversary can forge integrity proofs for pre-compromise log events is to recover expired keys. However, because used keys are securely deleted from the system, and because the chosen key update function is a collision (and preimage) resistant hash function (BLAKE2), a signing key residing on the system when compromised cannot be used to recover prior keys. We consider the feasibility of log tampering attacks below:

- *Log Modification.* An adversary may not modify log entries without being detected in an audit. Any modification would cause Verify to fail for the tampered entry.
- *Log Deletion.* An adversary cannot delete log events. Removing any subset of events from a stream would cause the ordered sequence of keys used to commit the events not to match the ordered sequence of signed events, causing Verify to fail.
- *Log Insertion/Reordering.* Similarly, an adversary cannot insert events or reorder log entries, as this would invalidate the ordering between signing and verification of entries, causing Verify to fail.
- *Log Truncation.* Truncation attacks can be detected by checking for a known message at the end of the log, which the administrator can achieve by having live hosts sign a new message immediately prior to an audit.

The novel security property in our design is **G2**, which assures synchronous integrity for all the events that occur before full system compromise. KennyLoggings achieves this goal through instrumenting the `audit_log_end` function in the audit subsystem. This function is invoked in the kernel control path before the invoking process is allowed to proceed; the process is thus unable to violate the integrity of the event. This property also holds in an attack featuring multiple processes/threads; if a malicious running kernel thread is able to modify or erase the event m_i before it reaches the critical section of `audit_log_end`, this implies that the system has already been compromised and thus the property holds for all events occurring prior to the time t_n of full system compromise. This remains a significant security guarantee because all events describing how the first malicious thread violated kernel integrity will be committed. Further, **G2** is preserved when keys for future events are precomputed because each key is deleted immediately after use. Even with our optimization enabled, an attacker will only be able to access keys for events that occur after t_n.

KennyLoggings provides a tamperproof security mechanism (**G3**) because its runtime trusted computing base resides entirely in kernel memory; the attacker must compromise the system before they can disable KennyLoggings or forge future events, but by that point all events prior to t_n are committed. During initialization, we make the reasonable assumption that the host is configured in a secure environment (e.g., not yet connected to the Internet), and thus deployment does not expose an additional attack surface.

Finally, we will show that KennyLoggings satisfies **G4** by evaluating its performance overheads in Section 9.

9 EVALUATION

Experimental Setup. We use the same experimental setup of Section 3.2, with the unmodified kernel 4.15.0-47 as baseline for our comparisons. We configure KennyLoggings to store two sets of 100,000 precomputed keys, which occupy a total of 3.2 MB.

Microbenchmarks. We start with microbenchmarks that capture the application-independent, raw cost of KennyLoggings' base functionality. In particular, we measure the time that our implementation takes to perform each of the operations described in Sections 6.4.1 and 6.4.2, excluding from the measurement the time spent spinning (waiting to acquire the synchronization lock). We do so by manually invoking each operation in the kernel 1,000 times and report the median execution times in Table 2. The operations we care about the most are the ones executed synchronously, in the kernel control path. For the Sign routine, we observe an average performance of 164 ns per proof, making it the most expensive among the operations we introduce to the kernel control path. Our implementation additionally incurs the synchronous costs of appending the proof to each event, which takes 103 ns, and erasing the used key, which takes 73 ns. For the KeyUpdate routine, we

Table 2: Microbenchmarks results. Operations in the kernel control path are denoted by *. For the Sign/Verify operation, we use as input a log record of length 366 (average length in the remote attack). For the KeyUpdate operation, we start from a random key k_1 and compute all keys up to k_{1000}.

Operation	KennyLoggings
KeyGen	262 ns
Sign/Verify*	164 ns
KeyUpdate	218 ns
Erase the current key*	73 ns
Append proof to log event*	103 ns

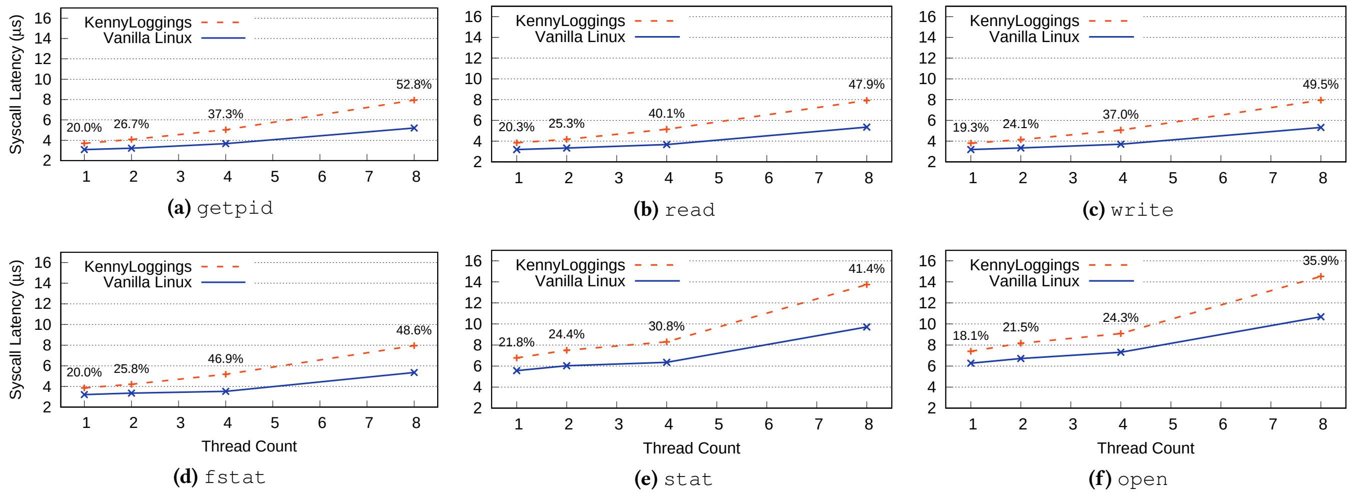


Figure 6: Microbenchmarks on system call latency. We report the median overhead over 100,000 iterations ran on the number of threads reported on the x axis. Due to spin lock contention, the overhead increases when the thread count increases.

Table 3: Application benchmarks results. We report the median values across 20 runs, together with the average number of log events generated per second by each benchmark. Besides 7-zip, smaller numbers are better.

Test Type	Vanilla	KennyLoggings	Overhead	Events/s
nginx	62.5 μs	67.5 μs	8.00%	21,211
apache2	60 μs	65 μs	8.33%	22,621
redis	20.5 μs	22.3 μs	9.12%	44,403
postmark	48 ms	53 ms	10.42%	96,732
7-zip	27,881 MIPS	27,743 MIPS	0.49%	12
openssl	743 μs	743 μs	0.00%	14
blast	968.160 s	973.199 s	0.52%	40
blake2	3.315 cy/B	3.315 cy/B	0.00%	282

Table 4: Application benchmarks with and without key precomputation. The overheads over Vanilla (median of 20 runs) are in parentheses. Smaller numbers are better.

Test Type	Unoptimized	Optimized	Improvement
nginx	73.5 μs (17.60%)	67.5 μs (8.00%)	9.60%
apache2	70 μs (16.67%)	65 μs (8.33%)	8.34%
redis	23.8 μs (16.29%)	22.3 μs (9.12%)	7.17%
postmark	57.5 ms (19.79%)	53 ms (10.42%)	9.37%

observe an average performance of 218 ns. Our optimization allows us to execute this operation asynchronously. That means that the task responsible for key precomputation will take 21.8 ms to refill a set of keys when it is woken up in our experimental configuration.

System Call Benchmarks. Next, we measure how KennyLoggings affects the time required to execute individual system calls when logging is enabled. We first perform this test while running system calls on a single thread. Next, we run system calls on multiple threads, to catch the effect of the spinlock contention. The results are presented in Figure 6. Without additional contention, KennyLoggings average overhead on an individual system call is below 20%. However, even when we use all the CPU cores, maximizing contention, KennyLoggings' overhead on an individual system call remains below 53%. In practice, this overhead could be reduced by using per-core evolving key sequences and spinlocks.

Application Workloads. To evaluate the system-wide impact of KennyLoggings on realistic workloads, we compare our performance to the (insecure) Vanilla kernel on a series of application benchmarks. These benchmarks can be divided into two categories: the first category of applications are I/O-intensive benchmarks, namely NGINX [84], apache2 [111], redis [97], and postmark [56]; the second category are CPU-intensive benchmarks, namely 7-zip [88], openssl [113], blast [27] and blake2 [16]. For apache2, NGINX and Redis, we use the apache bench and redis-benchmark tools, configured to send 30,000 serial requests and measure the average latency per request. For postmark, we use the built-in benchmark with its default configuration and measure its runtime. For 7-zip, we use the built-in benchmark configured to use all CPU cores available and measure the LZMA compression speed (in MIPS, Million Instructions Per Second). For openssl, we use the built-in "speed" benchmark configured to use all CPU cores available and measure the time to compute an rsa4096 signature. For blake2 and blast, we use the built-in single-threaded benchmarks that measure the median performance (in cycles/byte) and runtime, respectively.

Table 3 shows the results. The I/O-intensive benchmarks have the largest overheads at up to 10.42%. This result is unsurprising in that I/O-based workloads generate more system calls, leading to the creation of thousands of log events per second. This overhead is large but manageable, especially when considering that log integrity is achieved at the application layer at the cost of just 2 to 5 μs per request to nginx, apache2, and redis, and that most applications perform additional work that amortizes these costs. The CPU-intensive benchmarks enjoy near-zero overheads due to their limited system call activity, which generates small amounts of log events per second. This result demonstrates that our modifications impose negligible overhead on unrelated kernel functionality.

Key Precomputation. Next, we report on the concrete improvement that precomputing keys allowed us to achieve by measuring the performance of the application benchmarks without the optimization. Table 4 reports the results. We exclude the CPU-intensive benchmarks because we did not observe any significant difference in their results. That is, precomputing keys does not incur an observable overhead on the CPU utilization. On the other hand, the I/O-intensive benchmarks perform on average 8.62% better when KennyLoggings is configured to precompute keys.

Storage Cost. The storage overhead of KennyLoggings is fixed at 19 bytes per log event. These bytes are used for the proof of integrity of 16 hexadecimal characters along with a 3-byte label which is necessary to preserve the semantics of Linux Audit records. In this way, we preserve compatibility with applications that analyze log records; they simply ignore the KennyLoggings-specific field. To put this per-event cost in context, let us consider that Ma et al. [69] profile a web server under realistic conditions and observe an average daily rate of 2.76 million Linux Audit events, yielding a log of size 1.02 GB. From a back-of-the-envelope calculation, we extrapolate that the storage cost with KennyLoggings enabled would add just 52 MB, an overhead of only 5.14%.

10 DISCUSSION AND FUTURE WORK

Protocol Enhancements. KennyLoggings's tamper-evident logging protocol is optimized for concrete log commitment (signing) cost. As a trade-off, it lacks some of the benefits of other cryptographic schemes, such as public verifiability and aggregate authentication. Public verifiability is a desirable property because it can be used to allow third-parties (e.g. Court agents [34, 50, 51]) to verify the authenticity of a given set of logs without needing access to any secret key. Aggregate authentication is useful because it provides constant time verification costs for the auditor, independently of the number of log events in an audit. These techniques are prohibitively costly for synchronous deployment in the kernel control path; however, KennyLoggings can be extended to achieve these properties through the introduction of an intermediate trusted verifier that operates in the typical asynchronous event commitment model used in prior work (e.g., [89]). In such a deployment, KennyLoggings could send log events and MACs to the intermediate verifier, which could verify that no "in-flight" log tampering has occurred. The intermediary could then produce an aggregate signature over the log stream to make subsequent audits faster and verifiable by untrusted third parties. Thus, even when more advanced features are required, KennyLoggings still plays an important role in establishing a secure chain of custody for audit logs.

Performance Improvements. We identify two directions towards improving the performance of KennyLoggings. The first direction is utilizing faster primitives for Sign and KeyUpdate. KennyLoggings currently uses SipHash-2-4 for the Sign function, which, at the time of writing, is arguably the fastest 128-bit secure pseudorandom function that, importantly, has withstood cryptanalysis by experts (e.g., [22]) and is widely deployed in practice (e.g., in the Linux kernel). However, faster pseudorandom functions, such as the nascent, less well-studied HighwayHash [118], or hardware acceleration may be deployed in the future and could be used to reduce the overhead of KennyLoggings. Similarly, faster cryptographic hash functions, such as the recently presented BLAKE3 [17], could be used to speed up KennyLoggings' KeyUpdate phase.

The second direction towards reducing the overheads of KennyLoggings is to explore security-performance trade-offs. Recall that in KennyLoggings, each integrity proof is generated using a unique key. In contrast, the original forward secure message authentication scheme introduced by Bellare and Yee [11] proposed that keys should evolve over time intervals called *epochs* (sometimes referred to as *stages*). That is, in [11], integrity proofs belonging to the same epoch are generated using the same key, which is only updated for subsequent epochs. This reduces the number of key updates required, which can promise gains in performance. One could even imagine generating integrity proofs for a "batch" of log events, further squeezing performance out of the scheme. Observe that these optimizations trade-off security for performance. Suppose an attacker compromises a machine at a time nearing the end of an epoch. Then, the attacker can still forge false integrity proofs for the current epoch, since the corresponding signing key has not yet been securely erased from memory. Should the entire attack trace reside within a single epoch, the attacker could undetectably conceal any forensic evidence of their intrusion. Thus, a one-to-one correspondence between log events, keys, and integrity proofs, as we have implemented in KennyLoggings, provides the strongest security guarantees, satisfying **G2**. Nevertheless, future work could implement an epoch-based scheme by setting appropriate parameters (depending on how fast an attacker can intercept log events in practice) while still maintaining security. Our empirical study of the race condition vulnerability serves as a first step towards understanding this security-performance trade-off in practice.

Applicability to Other Frameworks. KennyLoggings is designed to be generic with respect to the logging framework. In this work, we have only evaluated our attack and defense on Linux Audit. Nevertheless, asynchronous logging mechanisms are ubiquitous and apply to all kernel logging frameworks we are aware of (e.g., [8, 66, 69, 75, 76, 109]). In fact, block-based I/O operations are always asynchronous within operating system kernels unless explicitly configured for synchrony. For example, Event Tracing for Windows (ETW), which is the standard kernel logging facility on Windows, also uses kernel buffering and delegates "a separate thread" to "flush the buffer data to the ETW log file" [75]. Still, future work is needed to investigate our attack and defense methodologies on Windows.

Alternative Attack Strategies. The attack we presented in this paper targets the kernel log buffer to undetectably remove compromise-related events before they are logged. Other attacks to asynchronous audit frameworks are possible. For example, a viable attack towards Linux Audit could be for the attacker to generate a burst to fill up the kernel buffer to its maximum capacity, effectively causing `kauditd`—depending on its configuration (cf. Section 2)—to either drop the subsequent events related to the attack (e.g., hiding the privilege escalation methods from the log) or to kernel panic (causing a denial of service). However, neither of these strategies is stealthy. The denial of service scenario can be detected by existing systems (e.g., [54, 89]) at the next startup, and, importantly, is not in an APT attacker's interests as it will cause the system administrator to intervene and effectively prevent them from completing their mission [77]. The log loss scenario not only leaves a suspicious gap in the log (due to benign events being dropped too) but is also easily detectable by checking the lost record counter of `kauditd`, which is included in all audit reports. There may also exist variants of our race condition attack that target other buffers of the logging pipeline, such as user space queues and I/O queues at the storage or network interface. Future work is needed to explore if these or other variants of our attack are feasible.

11 RELATED WORK

Cryptographic Approaches. The first formalization of the secure logging problem dates to 1997, when Bellare and Yee [10, 11] defined the notion of *forward integrity* (later called *forward security*) for audit logs, which KennyLoggings' tamper-evidence protocol builds on. Forward integrity schemes that use symmetric primitives [10, 11, 99, 100, 104] offer efficient proof generation costs, but need to rely on a trusted verifier sharing a secret with the logger. In contrast, schemes that rely on asymmetric primitives [44] achieve public verifiability at the cost of larger overheads both to generate and to verify proofs. Subsequent works focused on reducing the overhead of the verifier through the use of sequential aggregate signatures [65, 123], but have later been broken [39]. Yavuz et al. [122] further presented an optimized signing protocol which relies on a large public key size (linear with the number of supported log entries). Most recently, Hartung et al. [40] presented a scheme that combines forward-secure sequential aggregate signatures with forward-secure signatures. However, their work it still incurs impractically large computational costs to generate proofs.

The use of append-only data structures has also been proposed for storing logs in a tamper-evident fashion. These solutions include hash-based history trees [20] and authenticated schemes such as Balloon [95]. These schemes call for log events generated by a trusted host to be stored in a remote untrusted server (referred to as "logger"), and their data structures provide an efficient interactive protocol to verify that an event was correctly recorded by the logger.

In our system, we chose to use a simple, non-interactive scheme using only symmetric cryptographic primitives, since our chief concerns are synchronous integrity (**G2**) and performance (**G4**). The more sophisticated schemes described here are interoperable with our system design, but careful vetting is required to ensure that their added overheads do not prohibitively degrade performance.

Hardware Approaches. Beyond cryptographic-only approaches, prior work has proposed to solve the secure logging problem using trusted hardware [54, 85, 89, 103]. One of the benefits of these approaches is their ability to protect the secrecy of signing keys even after full system compromise—this property allows them to avoid frequent and costly key updates. However, because the TEEs they are based on support only Ring 3 execution, none of these designs is able to defend against our in-kernel race condition attack. Nevertheless, as we discussed in Section 10, KennyLoggings could be extended to work in conjunction with these systems to achieve properties such as public verifiability.

Race Conditions. There exist decades of research on highlighting and protecting from the dangers of race conditions in operating systems [15, 30, 31, 79, 116, 119]. These works investigate a type of race condition commonly referred to as time-of-check/time-of-use (TOCTOU) bug, which exists due to lack of synchronization between the enforcement of a security policy for an event (at time-of-check) and its occurrence (at time-of-use > time-of-check). In contrast, our attack exploits an intrinsic design flaw of audit frameworks to tamper with events that have already occurred (i.e., after time-of-use), independently of the presence of concurrency bugs.

Closely related to our work is also PillarBox [18], which is the first work to describe the goal of securing log events before an attacker can intercept them. Our work departs from [18] in several ways. In [18], the race condition occurs at the network level, in which a security analytics source (SAS) transmits logs to a remote server for analysis. In our work, however, the race condition occurs at the OS level, in which log events generated after a syscall travel through an in-kernel buffer to the logging daemon. Because the race condition of [18] sits at a higher level of abstraction, PillarBox is still vulnerable to the attack we described. Another difference is that, in addition to log integrity, PillarBox is designed to provide a stealth property, which conceals when the SAS has generated compromise-related alerts. In our work, we only consider log integrity, which allows us to arrive at a more efficient and modular solution that is interoperable with existing logging frameworks.

Attack Investigation. Related to system auditing are also several attack investigation works that derive insights from audit logs by parsing events into dependency (or *provenance*) graphs [8, 32, 57, 69, 90, 93]. Various methods have been proposed to automatically identify security insights [12, 23, 35, 37, 41, 45, 63, 77, 78, 87, 92, 101, 102, 117, 124], interpret the event stream in a way that more accurately explains application-layer semantics [7, 43, 59, 61, 66, 68, 80, 120], or to more quickly and expressively process queries on dependency graphs [28, 29, 52, 53, 64, 91]. All of the above work fully trusts the integrity of the audit logs used as inputs to their systems. KennyLoggings complements this line of work by providing log integrity for all events as they occur on the system.

12 CONCLUSION

Existing secure logging systems operate asynchronously—actions associated with an ongoing attack are permitted to proceed before their events are securely *committed* to the system log. In this work, we empirically demonstrated a race condition attack that exploits this vulnerability to tamper with compromise-related events before their commitment, thus bypassing all existing protections. We proposed a solution, KennyLoggings, that overcomes this limitation by decoupling an event's commitment from its storage, guaranteeing synchronous integrity for attack-related events upon their occurrence. KennyLoggings can thus give investigators insight into the window of time during which existing secure logging systems are vulnerable. We implemented our solution on the Linux kernel, and demonstrated that it introduces between 7% and 11% overhead on log-intensive application workloads.

ACKNOWLEDGMENTS

This work was funded in part by NSF award #1750024. We thank our shepherd Venkat Venkatakrishnan as well as the anonymous reviewers for their constructive feedback. We also thank Joshua Reynolds for his very useful suggestions on previous drafts of this paper.

REFERENCES

[1] 2014. CVE-2014-6271. Vulnerability reference: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271. Accessed on 08.17.2020.

[2] 2017. CVE-2017-16995. Vulnerability reference: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995. Accessed on 08.17.2020.

[3] Airbus Commercial Aircraft. 2020. Forward integrity and confidentiality of logs - syslog-ng/syslog-ng. https://github.com/syslog-ng/syslog-ng/pull/3121. Accessed on 08.17.2020.

[4] AT&T Cybersecurity. [n.d.]. PCI DSS log management & monitoring. https://cybersecurity.att.com/solutions/pci-dss-log-management-monitoring. Accessed on 08.17.2020.

[5] Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. SipHash: a fast short-input PRF. In *Proc. of the International Conference on Cryptology in India (INDOCRYPT)*.

[6] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. 2013. BLAKE2: simpler, smaller, fast as MD5. In *Proc. of the International Conference on Applied Cryptography and Network Security (ACNS)*.

[7] Adam Bates, Wajih Ul Hassan, Kevin R.B. Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. 2017. Transparent Web Service Auditing via Network Provenance Functions. In *Proc. of the International World Wide Web Conference (WWW)*.

[8] Adam Bates, Dave Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proc. of the USENIX Security Symposium (USENIX)*.

[9] Greg Belding. [n.d.]. Ethical Hacking: Log Tampering 101. https://resources.infosecinstitute.com/category/certifications-training/ethical-hacking/covering-tracks/log-tampering-101/. Accessed on 08.17.2020.

[10] Mihir Bellare and Bennet Yee. 1997. *Forward Integrity For Secure Audit Logs*. Technical Report. Computer Science and Engineering Department, University of California at San Diego.

[11] Mihir Bellare and Bennet Yee. 2003. Forward-Security in Private-Key Cryptography. In *Proc. of the Cryptographers' Track at the RSA Conference (CT-RSA)*. Springer.

[12] Konstantin Berlin, David Slater, and Joshua Saxe. 2015. Malicious Behavior Detection using Windows Audit Logs. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*.

[13] Tara Siegel Bernard, Tiffany Hsu, Nicole Perlroth, and Ron Lieber. 2017. *Equifax Says Cyberattack May Have Affected 143 Million in the U.S.* https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html

[14] Chris Bing. 2017. Shadow Brokers' latest leak could have come from beyond NSA staging servers. https://www.cyberscoop.com/shadow-brokers-nsa-microsoft-windows-exploits-2017/. Accessed on 08.17.2020.

[15] Matt Bishop and Michael Dilger. 1996. Checking for Race Conditions in File Accesses. *Computing systems* 9, 2 (1996), 131–152.

[16] BLAKE2. [n.d.]. BLAKE2 – fast secure hashing. https://blake2.net/. Accessed on 08.17.2020.

[17] BLAKE3. [n.d.]. BLAKE3: official implementations of the BLAKE3 cryptographic hash function. https://github.com/BLAKE3-team/BLAKE3/. Accessed on 08.17.2020.

[18] Kevin D. Bowers, Catherine Hart, Ari Juels, and Nikos Triandopoulos. 2014. PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[19] Armando Ortiz Cornet and Joan Miquel Bardera Bosch. 2013. Method and system of generating immutable audit logs. US Patent 8,422,682.

[20] Scott A. Crosby and Dan S. Wallach. 2009. Efficient Data Structures For Tamper-Evident Logging. In *Proc. of the USENIX Security Symposium (USENIX)*.

[21] dark laboratorys. [n.d.]. A better generation of logcleaners. https://web.archive.org/web/20070218231819/http://darklab.org/~jot/logclean-ng/logcleaner-ng_1.0_lib.html. Accessed on 08.17.2020.

[22] Christoph Dobraunig, Florian Mendel, and Martin Schläffer. 2014. Differential Cryptanalysis of SipHash. In *Proc. of the International Conference on Selected Areas in Cryptography (SAC)*.

[23] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[24] Michael Dymshits, Benjamin Myara, and David Tolpin. 2017. Process Monitoring on Sequences of System Call Count Vectors. In *Proc. of the International Carnahan Conference on Security Technology (ICCST)*.

[25] Jake Edge. [n.d.]. Forward secure sealing. https://lwn.net/Articles/512895/. Accessed on 08.17.2020.

[26] European Parliament and of the Council. 2016. Regulation (EU) 2016/679 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* L119 (2016).

[27] Fiehn Lab. [n.d.]. blast 2.7.1. http://fiehnlab.ucdavis.edu/staff/kind/collector/benchmark/blast-benchmark. Accessed on 08.17.2020.

[28] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. 2018. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *Proc. of the USENIX Security Symposium (USENIX)*.

[29] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. 2018. AIQL: Enabling Efficient Attack Investigation from System Monitoring Data. In *Proc. of the USENIX Security Symposium (USENIX)*.

[30] Tal Garfinkel. 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[31] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[32] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proc. of the International Middleware Conference (Middleware)*.

[33] Peter H. Gregory. 2015. *CISSP Guide to Security Essentials* (2nd ed.). Course Technology Press.

[34] Roger A. Grimes. 2016. Why it's so hard to prosecute cyber criminals. https://www.csoonline.com/article/3147398/why-its-so-hard-to-prosecute-cyber-criminals.html. Accessed on 08.17.2020.

[35] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2015. LEAPS: Detecting Camouflaged Attacks with Statistical Learning Guided by Program Analysis. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*.

[36] Steve Hales. [n.d.]. Last Door Log Wiper. https://packetstormsecurity.com/files/118922/LastDoor.tar. Accessed on 08.17.2020.

[37] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. 2020. UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. *Proc. of the Symposium on Network and Distributed System Security (NDSS)* (2020).

[38] Kabot Haniradi. [n.d.]. mig-logcleaner-resurrected. https://github.com/Kabot/mig-logcleaner-resurrected. Accessed on 08.17.2020.

[39] Gunnar Hartung. 2017. Attacks on Secure Logging Schemes. In *Proc. of the International Conference on Financial Cryptography and Data Security (FC)*.

[40] Gunnar Hartung, Björn Kaidel, Alexander Koch, Jessica Koch, and Dominik Hartmann. 2017. Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures. In *Proc. of the International Conference on Provable Security (ProvSec)*.

[41] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[42] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[43] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. 2020. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[44] Jason E. Holt. 2006. Logcrypt: Forward Security and Public Verification for Secure Audit Logs. In *Proc. of the Australasian Information Security Workshop (AISW-NetSec)*.

[45] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott Stoller, and V.N. Venkatakrishnan. 2017. SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data. In *Proc. of the USENIX Security Symposium (USENIX)*.

[46] Ryan Huber. 2016. Syscall Auditing at Scale—The Slack Engineering Blog. https://slack.engineering/syscall-auditing-at-scale-e6a3ca8ac1b8. Accessed on 08.17.2020.

[47] IBM Knowledge Center. [n.d.]. Storage and analysis of audit logs. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.sec.doc/doc/c0052328.html. Accessed on 08.17.2020.

[48] IDC. 2019. Worldwide IT Operations Management Software Forecast, 2019–2023. https://www.idc.com/getdoc.jsp?containerId=US44896118. Accessed on 08.17.2020.

[49] (ISC)^2. [n.d.]. Cybersecurity Certification - CISSP, Certified Information Systems Security Professional. https://www.isc2.org/Certifications/CISSP. Accessed on 08.17.2020.

[50] Marshall Jarrett, M Bailie, E Hagen, and E Etringham. 2010. Prosecuting Computer Crimes. *United States. Department of Justice. Office of Legal Education* (2010).

[51] Marshall Jarrett, M Bailie, E Hagen, and N Judish. 2009. Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations. *United States. Department of Justice. Office of Legal Education* (2009).

[52] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation

with On-demand Inter-Process Information Flow Tracking. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[53] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2018. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *Proc. of the USENIX Security Symposium (USENIX)*.

[54] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs With SGX.

[55] Kent Karen and Souppaya Murugiah. 2006. NIST Special Publication 800-92, Guide to Computer Security Log Management.

[56] Jeffrey Katcher. 1997. *Postmark: A new file system benchmark.* Technical Report.

[57] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*.

[58] Brendan I. Koerner. 2016. Inside the Cyberattack That Shocked the US Government. https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/. Accessed on 08.17.2020.

[59] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[60] Kyu Hyung Lee. [n.d.]. Ubsi. https://github.com/kyuhlee/UBSI. Accessed on 08.17.2020.

[61] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[62] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proc. of the USENIX Security Symposium (USENIX)*.

[63] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. 2019. Log2vec: A Heterogeneous Graph Embedding Based Approach for Detecting Cyber Threats within Enterprise. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[64] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[65] Di Ma and Gene Tsudik. 2009. A New Approach to Secure Logging. *ACM Transactions on Storage (TOS)* 5, 1 (2009).

[66] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[67] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. 2018. Kernel-Supported Cost-Effective Audit Logging for Causality Tracking. In *Proc. of the USENIX Annual Technical Conference (ATC)*.

[68] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple Perspective Attack Investigation with Semantics Aware Execution Partitioning. In *Proc. of the USENIX Security Symposium (USENIX)*.

[69] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[70] Gregory Machler. [n.d.]. Protecting data with WORM drives. https://www.csoonline.com/article/2131925/protecting-data-with-worm-drives.html. Accessed on 08.17.2020.

[71] maldevel. [n.d.]. ClearLogs. https://sourceforge.net/projects/clearlogs/. Accessed on 08.17.2020.

[72] Stephan Marwedel. 2020. Protecting log records at 35,000 feet - APNIC Blog. https://blog.apnic.net/2020/03/27/protecting-log-records-at-35000-feet/. Accessed on 08.17.2020.

[73] Stephan Marwedel. 2020. Secure logging with syslog-ng. https://fosdem.org/2020/schedule/event/security_secure_logging_with_syslog_ng/. Accessed on 08.17.2020.

[74] Kayle Matthews. 2019. *Incident Of The Week: Historic Capital One Hack Reaches 100 Million Customers Affected By Breach.* https://www.cshub.com/attacks/articles/incident-of-the-week-historic-capital-one-hack-reaches-100-million-customers-affected-by-breach

[75] Microsoft. [n.d.]. ETW Framework Conceptual Tutorial. https://docs.microsoft.com/en-us/message-analyzer/etw-framework-conceptual-tutorial. Accessed on 08.17.2020.

[76] Microsoft. [n.d.]. Process Monitor v3.50. https://docs.microsoft.com/en-us/sysinternals/downloads/procmon. Accessed on 08.17.2020.

[77] Sadegh Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and V. N. Venkatakrishnan. 2019. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[78] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2019. Poirot: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[79] James Morris, Stephen Smalley, and Greg Kroah-Hartman. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. of the USENIX Security Symposium (USENIX)*.

[80] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-Aware Storage Systems. In *Proc. of the USENIX Annual Technical Conference (ATC)*.

[81] National Institute of Standards and Technology. 2013. NIST Special Publication 800-53 (Rev. 4), Security Controls and Assessment Procedures for Federal Information Systems and Organizations.

[82] National Security Agency. 1999. Controlled Access Protection Profile, Version 1.d. https://www.niap-ccevs.org/Profile/Info.cfm?PPID=14&id=14.

[83] NetApp. [n.d.]. SnapLock: WORM Compliance – Data Compliance. https://www.netapp.com/us/products/backup-recovery/snaplock-compliance.aspx. Accessed on 08.17.2020.

[84] NGINX Inc. [n.d.]. NGINX 1.10.3. https://www.nginx.com/. Accessed on 08.17.2020.

[85] Hung Nguyen, Bipeen Acharya, Radoslav Ivanov, Andreas Haeberlen, Linh T.X. Phan, Oleg Sokolsky, Jesse Walker, James Weimer, William Hanson, and Insup Lee. 2016. Cloud-Based Secure Logger for Medical Devices. In *Proc. of the IEEE International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*.

[86] OccupytheWeb. 2013. How to Cover Your Tracks & Leave No Trace Behind on the Target System. https://null-byte.wonderhowto.com/how-to/hack-like-pro-cover-your-tracks-leave-no-trace-behind-target-system-0148123/. Accessed on 08.17.2020.

[87] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H. Chin, and Sumayah Alrwais. 2015. Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data. In *Proc. of the Conference on Dependable Systems and Networks (DSN)*.

[88] p7zip. [n.d.]. p7zip 16.02. https://sourceforge.net/projects/p7zip/. Accessed on 08.17.2020.

[89] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Christopher W. Fletcher, Adam Bates, Andrew Miller, and Dave Tian. 2020. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[90] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-system Provenance Capture. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[91] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, , and Margo Seltzer. 2018. Runtime Analysis of Whole-System Provenance. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[92] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. HERCULE: Attack Story Reconstruction via Community Discovery on Correlated Log Graph. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[93] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[94] Red Hat Customer Portal. [n.d.]. System Auditing. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing. Accessed on 08.17.2020.

[95] Tobias Pulls and Roel Peeters. 2015. Balloon: A Forward-Secure Append-Only Persistent Authenticated Data Structure. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*.

[96] Rapid7. [n.d.]. Metasploit, the world's most used penetration testing framework. https://www.metasploit.com/. Accessed on 08.17.2020.

[97] Redis Labs. [n.d.]. Redis 3.0.6. https://redis.io/. Accessed on 08.17.2020.

[98] Michael Riley, Ben Elgin, Dune Lawrence, and Carol Matlack. 2014. Missed Alarms and 40 Million Stolen Credit Card Numbers: How Target Blew It. https://www.bloomberg.com/news/articles/2014-03-13/target-missed-warnings-in-epic-hack-of-credit-card-data. Accessed on 08.17.2020.

[99] Bruce Schneier and John Kelsey. 1998. Cryptographic Support for Secure Logs on Untrusted Machines. In *Proc. of the USENIX Security Symposium (USENIX)*.

[100] Bruce Schneier and John Kelsey. 1999. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)* 2, 2 (1999), 159–176.

[101] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. 2018. Tiresias: Predicting Security Events Through Deep Learning: Template Based Efficient Data Reduction For Big-Data Causality Analysis. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[102] Yun Shen and Gianluca Stringhini. 2019. Attack2vec: Leveraging Temporal Word Embeddings to Understand the Evolution of Cyberattacks. In *Proc. of the USENIX Security Symposium (USENIX)*.

[103] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. 2017. EmLog: Tamper-Resistant System Logging for Constrained Devices with TEEs. In *Proc. of the International Conference on Information Security Theory and Practice (WISTP)*.

[104] Arunesh Sinha, Limin Jia, Paul England, and Jacob R Lorch. 2014. Continuous Tamper-Proof Logging Using TPM 2.0. In *Proc. of the International Conference on Trust and Trustworthy Computing (TRUST)*.

[105] Splunk Inc. [n.d.]. Splunk. https://www.splunk.com. Accessed on 08.17.2020.

[106] Splunk Inc. 2018. Splunk Ranked No. 1 in IDC Worldwide SIEM Market Share 2018. https://www.splunk.com/en_us/form/splunk-ranked-no1-in-idc-worldwide-siem-market-share-2018.html. Accessed on 08.17.2020.

[107] Swaminathan Sundararaman, Gopalan Sivathanu, and Erez Zadok. 2008. Selective Versioning in a Secure Disk System. In *Proc. of the USENIX Security Symposium (USENIX)*.

[108] SUSE Linux AG. 2004. Linux Audit-Subsystem Design Documentation for Linux Kernel 2.6, v0.1. http://uniforum.chi.il.us/slides/HardeningLinux/LAuS-Design.pdf.

[109] Sysdig. [n.d.]. Sysdig – Open Source System Capturing. https://sysdig.com/opensource/inspect/. Accessed on 08.17.2020.

[110] syslog-ng. [n.d.]. Log Management Solutions. https://www.syslog-ng.com/. Accessed on 08.17.2020.

[111] The Apache Software Foundation. [n.d.]. httpd 2.4.18. https://httpd.apache.org/. Accessed on 08.17.2020.

[112] The MITRE Corporation. [n.d.]. CAPEC-268: Audit Log Manipulation. https://capec.mitre.org/data/definitions/268.html. Accessed on 08.17.2020.

[113] The OpenSSL Project. [n.d.]. OpenSSL 1.1.1. https://www.openssl.org/. Accessed on 08.17.2020.

[114] Tripwire. [n.d.]. Log Management. https://www.tripwire.com/solutions/log-management/. Accessed on 08.17.2020.

[115] VMware Carbon Black. 2019. *Global Incident Response Threat Report - The Ominous Rise of "Island Hopping" & Counter Incident Response Continues*. Technical Report. Accessed on 08.17.2020.

[116] David A. Wagner. 1999. *Janus: an Approach for Confinement of Untrusted Applications*. Master's thesis. University of California, Berkeley.

[117] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[118] Jan Wassenberg and Jyrki Alakuijala. [n.d.]. HighwayHash. https://github.com/google/highwayhash. Accessed on 08.17.2020.

[119] Robert N. M. Watson. 2007. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*.

[120] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. 2020. UIScope: Accurate, Instrumentation-free, and Visible Attack Investigation for GUI Applications. (2020).

[121] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead Store Elimination (Still) Considered Harmful. In *Proc. of the USENIX Security Symposium (USENIX)*.

[122] Attila A. Yavuz and Peng Ning. 2009. BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[123] Attila A. Yavuz, Peng Ning, and Michael K. Reiter. 2012. Efficient, Compromise Resilient and Append-only Cryptographic Schemes for Secure Audit Logging. In *Proc. of the International Conference on Financial Cryptography and Data Security (FC)*.

[124] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-Scale Log Analysis for Detecting Suspicious Activity in Enterprise Networks. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[125] Kim Zetter. 2015. Health Insurer Anthem Is Hacked, Exposing Millions of Patients' Data. https://www.wired.com/2015/02/breach-health-insurer-exposes-sensitive-data-millions-patients/. Accessed on 08.17.2020.