VMBLS: Virtual Machine Based Logging Scheme for Prevention of Tampering and Loss

Masaya Sato, Toshihiro Yamauchi

VMBLS: Virtual Machine Based Logging Scheme for Prevention of Tampering and Loss

Masaya Sato and Toshihiro Yamauchi

Graduate School of Natural Science and Technology, Okayama University,
3-1-1 Tsushima-naka, Kita-ku, Okayama, 700-8530 Japan
m-sato@swlab.cs.okayama-u.ac.jp,yamauchi@cs.okayama-u.ac.jp

Abstract. Logging information is necessary in order to understand a computer's behavior. However, there is a possibility that attackers will delete logs to hide the evidence of their attacking and cheating. Moreover, various problems might cause the loss of logging information. In homeland security, the plans for counter terrorism are based on data. The reliability of the data is depends on that of data collector. Because the reliability of the data collector is ensured by logs, the protection of it is important problem. To address these issues, we propose a system to prevent tampering and loss of logging information using a virtual machine monitor (VMM). In this system, logging information generated by the operating system (OS) and application program (AP) working on the target virtual machine (VM) is gathered by the VMM without any modification of the OS. The security of the logging information is ensured by its isolation from the VM. In addition, the isolation and multiple copying of logs can help in the detection of tampering.

Keywords: Log, security, virtualization, virtual machine monitor, digital forensics.

1 Introduction

The countermeasure for terrorism is one important topic in homeland security. In the field of counter-terrorism, enormous quantity of data is gathered and analyzed for the planning of countermeasures. Computers and networks are used to gather and analyze data, computer science is deeply committed to homeland defence and security. In the field of computer science, countermeasures are considered for cyber terrorism as an activity in homeland security. Recently, information technology is used as a tool to control infrastructures. Cyber terrorism is able to cause critical damage on infrastructures in low cost. Thus, the countermeasure for cyber terrorism have been discussed.

The countermeasures might be weakened by attacking on the data gathered for homeland security. Therefore, the protection of the data is important. The protection of the logs of the APs is also necessary to ensure the validity of gathered information.

The computer terrorism has two characters: anonymity and the lack of evidences of attacks. In computer terrorism, it is difficult to acquire the information

that specifies the attacker. Because there are no evidences left on attacks using network, the logs that records the behavior of the systems are important. For this reason, the protection of the information is necessary for the prevention and investigation of computer terrorism.

Digital forensics is a method or technology for addressing these problems. This is a scientific method or research technology for court actions, which allows us to explain the validity of the electronic records. Many researchers are working in this area of the protection of logging information[2, 4, 8, 10, 11].

Syslog is commonly used as a logging program in Linux. In this case, the logging information generated by the AP (user log) and kernel (kernel log) is gathered by syslog. Syslog writes logs to file according to the policy, so attackers can tamper with logs by modifying the policy. Moreover, if the syslog program itself is attacked, the log files written are not reliable. In addition, the kernel log is stored in a ring buffer, and therefore, since the kernel log is gathered on a regular schedule, if many logs are generated and stored in the ring buffer before the next gathering time, old logs may be overwritten by new logs. As described above, the user log and kernel log can be tampered with or lost.

In this paper, we propose a logging system to prevent tampering and loss of logs with the virtual machine monitor (VMM). In this system, the OS that should be monitored (monitored OS, MOS) works on the virtual machine (VM). Logs in the MOS are gathered by the VMM without any modification of the MOS's kernel source codes. The VMM gathers user logs by hooking the system calls invoked in the MOS. Because the system gathers logs just after the output of logs, any possibilities for tampering are excluded. The VMM gathers current kernel logs from the buffer before new kernel logs have accumulated. Therefore, the system can gather current kernel logs in conjunction with the accumulation of new ones. Thus, no logs are lost through the buffer being overwritten by new kernel logs.

As mentioned above, the system gathers logs using the VMM. The VMM is independent of and invisible to the MOS, so it is difficult to detect and attack this system. Thus, the system itself is secure. In addition, because the logs gathered by the system are copied to the logging OS (LOS), it is easy to determine which part has been modified by attackers. Moreover, with the isolation of logs, any attacks on the MOS have no effects on the logs gathered by the system. These features mean that this system provides secure logging.

The contributions of this paper are as follows:

(1) The logging scheme for prevention of tampering and loss of logging information using VMM is proposed. The scheme can solve the problems in existing schemes and researches described in Section 2. The implementation has no modification of guest OSes and easy to introduce in existing systems.
(2) Evaluations are described and they show that the system is effective on protection of logs. The measurement of the performance with the system shows the overheads in the system calls that are related to logging is 50μs, and are not related to logging is only 2$\sim$5μs.

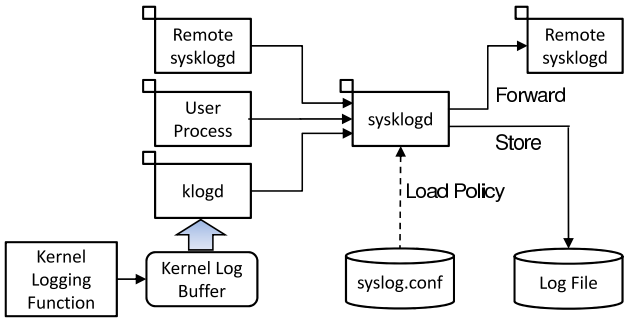

Fig. 1. Architecture of syslog.

2 Existing logging schemes

This section describes the architecture and problems of existing logging schemes. Section 6 also refers to these schemes, the comparisons between these schemes and our proposed system are in there.

2.1 Syslog

Syslog is a protocol for system management and security monitoring standardized by Internet Engineering Task Force (IETF)[6]. Syslog consists of a library and a daemon. The syslog library provides interfaces for logging, and the syslog daemon gathers logs and stores them as a file. Figure 1 shows the architecture of syslog. User and kernel logs are gathered as follows.

The syslog library provides functions for user program to send log messages to the syslog daemon. The syslog function sends messages to /dev/log with the `send` or `write` system call, and the syslog daemon gathers logs from /dev/log with the `read` system call.

The kernel accumulates logs in internal buffer (kernel log buffer). The kernel logging daemon (klogd) gathers logs from the kernel log buffer, and afterwards, klogd similarly sends logs to the syslog daemon.

Syslog also has a filtering function. Its policies are described in the configuration file (syslog.conf).

Syslog has the following problems:

(1) The behavior of the syslog daemon can be modified by tampering with the configuration file. In addition, if the syslog daemon itself is tampered with, its output can be unreliable.
(2) Users who have permission to access logs can tamper with them intentionally.
(3) Kernel logs in Linux are accumulated in the ring buffer and are gathered at fixed intervals. Thus, if the logs are not gathered for a long time, old logs can be overwritten by new ones. Old logs will also be overwritten if many logs are accumulated in a time that is shorter than the gathering interval.

Although improved syslog daemons have been developed (e.g., rsyslog, syslog-ng), these problems have not been addressed.

2.2 Protection of logs

Some research has been carried out on the protection of files by the file system. The system NIGELOG has been proposed for protecting log files [10]. This method has a tolerance for file deletion. It produces multiple backups of a log file, keeps them in the file system, and periodically moves them to other directories. By comparing the original file and the backups, any tampering with the log file can be detected. Moreover, if any tampering is detected, the information that has been tampered with can be restored from these backups.

The protection of files with the file system is still vulnerable to attacks that analyze the file system. Therefore, a log-protection method using virtualization has been proposed [11]. This method protects logs by saving them to another VM, so it is impossible to tamper with the logs from other VM. However, this method aims to protect the log of a journaling file system, so the scope of the protection target is different from that in our research.

The hysteresis signature is used to achieve the integrity of files. However, it is known that the algorithm of the hysteresis signature has a critical weak point. Although the hysteresis signature can detect the tampering and deletion of files, it cannot prevent tampering and deletion. Moreover, the manager of the signature generation histories can tamper with the histories and files. Therefore, a mechanism to solve this problem using a security device has been proposed [2]. This method constructs a trust chain from the data in the tamper-tolerant area of the security device, the source of the trust chain is protected from attackers. Nevertheless, this method is not versatile because it uses the special device.

2.3 Protection of syslog

The methods mentioned above are protecting log files. However, they cannot protect logs before storing of them. Thus, a method to guarantee syslog's integrity has been proposed [4], which uses a Trusted Platform Module (TPM) and a late launch by a Secure Virtual Machine (SVM) to ensure the validity of syslog. The validated syslog receives logs and sends them to a remote syslog.

2.4 Original logging method

An original logging method, independent of syslog, has been proposed for audit [8]. This method uses Linux Security Modules (LSM) to gather the logs, and Mandatory Access Control (MAC) to ensure their validity. The system also uses SecVisor [9], and DigSig [1]. SecVisor ensures the security of the logging framework, and DigSig prevents rootkit from making modifications to access permissions. DigSig adds a signature to a program, and prevents the execution of an unknown program by verifying its signature. This method gathers logs in

its own way, but the method modifies the kernel source codes. In general, kernel modification is difficult and complex, so the method lacks versatility. And the method uses variety of mechanisms, the overheads arising from them have large effect on daily operations on computers.

2.5 Problems of existing logging schemes

From the descriptions above, we find that there are three problems for logging:

(1) Attacks on logging information.
(2) Attacks on logging mechanism.
(3) Loss of kernel log.

The security of logging information is the main focus of the current research, and is of utmost important in digital forensics. However, this is not enough to ensure secure logging. For the protection of logging information, it is necessary to protect the logging mechanism itself. The reliability of logs generated by a program is determined by the reliability of the generator, so the security of the logging mechanism is also important. The third problem depends on the architecture of the Linux kernel log buffer. We currently assume that the guest is Linux, and so this problem needs to be addressed.

3 VMBLS: Virtual Machine Based Logging Scheme

3.1 Requirements and Approaches

To address problems in Section 2, we propose a system to prevent tampering and loss of logging information. There are three requirements for addressing the problems:

(1) Detection of all outputs of log (user log and kernel log).
We need to detect and gather all logs to keep them secure. Where we support only user logs and kernel logs. We do not support logs not sent to syslog.
(2) Isolation of log.
In order to secure a log, we obtain a copy of the log and isolate it from the MOS. Since the protection method using the file system is not secure, we isolate the logs in the LOS, assuming that the working environment is on a VM. As the LOS are isolated, attacks on the working environment have no effect on the logs. This isolation also enables us to detect loss of and tampering with logging information, because a comparison between copied logs and original ones will show any differences arising from such attacks.
(3) Security of logging mechanism.
To achieve security of the logging mechanism, we use a VMM. Since the VMM is independent of the VM, it is generally difficult to detect its existence and to attack it from the VM. A logging method in kernel space (e.g., LSM or kernel modification) is weakly defended from attacks. If attackers obtain

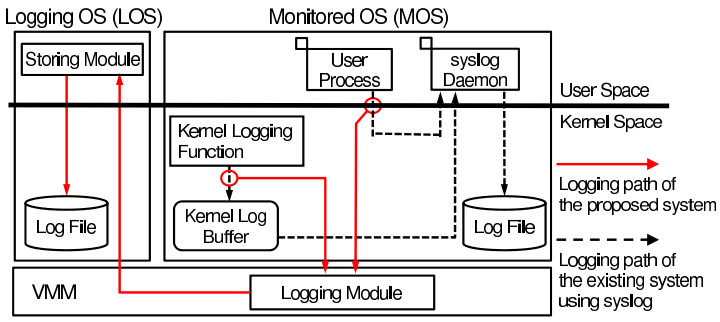

Fig. 2. Architecture of the proposed system.

root privileges, they can use bugs in kernel or APs to attack some programs and tamper with the logs to hide their activities. Thus, the isolation of the logging mechanism provides greater security than the methods in kernel.

Furthermore, simplicity and ease of introduction is demanded. The mechanism using a VMM makes our proposed system easier to introduce. The original logging method and security mechanism with kernel modification is difficult to apply to a new version of the kernel because of the kernel's complexity. The VMM, for which no modification of the kernel of the MOS is required, makes our method more flexible than methods that need kernel modification. Thus, we choose the use of a VMM as our method for the protection of logging information.

3.2 Architecture

Figure 2 shows the architecture of the proposed system. The MOS and LOS work on the VM. The LOS is a guest OS for storing logs gathered by the logging module. The logging module works in the VMM, and its details are described below. It gathers logs and sends them to the LOS, which then stores them in files.

3.3 Logging module

User log collector The collector acquires logs when the requirement for sending logs occurs. As shown in Fig. 2, the VMM hooks the system call that was invoked for sending logs from the user process to the syslog daemon.

The procedures for user log collection are detailed below:

(1) Detect a connect system call for a socket of /dev/log
(2) Determine the socket number from the first argument of the connect system call invoked at (1)
(3) Detect a send system call for the socket used at (2)

(4) Acquire the string as the log specified in the second argument of the send system call invoked at (3)

To send logs, the user process creates a socket and invokes a connect system call for /dev/log. The user log collector detects this connect system call and specifies the socket number that will be used to send the logs. Finally, the user log collector acquires logs by detecting the send system call for that socket.

However, the collector cannot recognize the process that sends the logs. To address this issue, the collector uses CR3 control register to determine the process; CR3 stores a unique value for each process since it shows the page directory.

Kernel log collector The collector acquires logs when the kernel logging function is called in a guest OS. Typically, the VMM cannot detect a function call in a guest OS. To solve this problem, the system sets a breakpoint in the guest OS. Breakpoint exception occurs when some process reaches this breakpoint. In the proposed system, since the guest OSes are fully virtualized, breakpoint exception is processed by the VMM. Using the exception as an opportunity to acquire logs, the VMM can gather kernel logs.

When the processing is brought to the VMM, the logging module checks the state of the kernel log buffer of the MOS. If new logs have been accumulated in the buffer, the logging module gathers them. After that, the VMM returns the processing to the guest OS. Since these processes have no effect on the state of the guest OS, the guest OS can continue to write the kernel log.

In this method, since kernel logs are gathered when a kernel logging function is called, old logs are never overwritten by new ones.

3.4 Storing module

The storing module stores logs gathered by the logging module. The storing module is an AP working on the LOS, and stores the logs as log files. The storing module actively gathers logs from the buffer of the VMM.

4 Implementation

4.1 Environment

We implement the system with Xen [3] as the VMM and Linux as the MOS. This version of Xen supports full virtualization. We did not modify the kernel source codes. However, the system needs the System.map file of the MOS, so this information must be provided beforehand. The reason for the system requiring this System.map information is that the system uses the address of the kernel log buffer and the kernel logging function. Because of the need for full virtualization, we prepared a CPU that supports virtualization extension.

4.2 Detecting a system call

The user log collector detects the invoking of system calls; such a mechanism is necessary because the proposed system is implemented with a VMM.

Therefore, in the proposed system, we applied a mechanism that causes a page fault when a system call is invoked [5]. In a fully virtualized environment, if a page fault occurs on the VM, then the VMM is raised (VM exit) [7]. After the VMM has been raised, the logging module acquires the user logs and hides the occurrence of the page fault. Finally, the VMM raises the guest OS, which works as if no event has occurred.

In this method, to cause a page fault, we modified some registers of the MOS. A system call using the sysenter (fast system call) refers the value in `sysenter_eip_msr` and jumps to its address to execute the system call function (`sysenter_eip_msr` is one of the machine-specific register (MSR)). Through modification of this value to another address to which access is not permitted from the MOS, a page fault is made to occur when a system call is invoked.

4.3 Setting a breakpoint

In the proposed system, we set a breakpoint in the kernel logging function of the MOS to gather kernel logs. The breakpoint is realized by embedding of a INT3 instruction. VM exit appears if a breakpoint exception occurs in the MOS, and the VMM gathers kernel logs with this exception.

In the following, we detail the procedure for setting breakpoints and gathering kernel logs. Here, we assume that the first INT3 instruction is already embedded.

(1) Breakpoint exception occurs and switches to the VMM.
    Kernel logs are gathered after the processing is switched to the VMM.
(2) Embed INT3 instruction to the next one.
(3) Restore the value at the address at which the exception occurred.
(4) Restart the processing from the restored instruction.
(5) Breakpoint exception occurs and switches to the VMM.
(6) Embed INT3 instruction to the firstly embedded address
(7) Restore the value on the address at which the exception occurred.
(8) Execute instruction from restored point.

5 Evaluation

5.1 Simplicity and ease of introduction

We implemented a prototype of the proposed system by modifying the Xen hypervisor. The total amount of source codes that we added and modified on Xen are only about 1,000 lines. The source codes of the MOS is not modified.

5.2 Purposes and environment

We evaluated the system from two points of view: the prevention and detection of tampering, and the loss of logging information. This paper also describes the overheads of the system. Table 1 details the environment used for this evaluation.

Table 1. Environment used for evaluation.

OS	Domain0	Linux 2.6.18-xen
	Fully virtualized domain	Linux 2.6.26
VMM		Xen 3.4.1
syslog		rsyslogd 3.18.6
CPU		Intel Core 2 Duo E6600
Memory	Physical	2,048 MB
	Domain0	1,024 MB
	Fully virtualized domain	1,024 MB

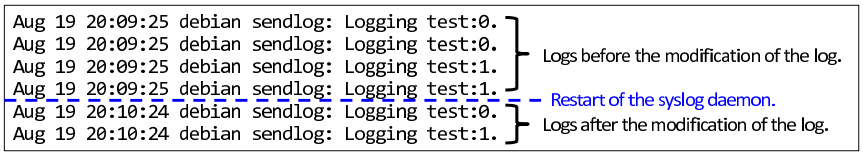

Fig. 3. A user log gathered by syslog.

5.3 Prevention of tampering

A log gathered by the proposed system is kept in a place where it is independent of the MOS. Thus, with the MOS working on a VM, it is difficult for attackers to tamper with the log even if they obtain root privileges.

5.4 Prevention of loss

Loss of user log We assume that an attacker tampers with the policy of syslog to suppress some parts of the log. We show that the proposed system can gather logs even if the policy has been tampered with. Figure 3 shows the logs gathered by syslog, and Fig. 4 shows those gathered by the proposed system. We compared both logs around the time when the policy was manipulated, as shown in Fig. 5. In the logs gathered by syslog, the logs decreased after the manipulation of the policy because of the exception of the mail facility. On the other hand, there are no changes in the logs gathered by our proposed system around the time of the manipulation of the policy. Thus, it is proved that the proposed system gathers logs regardless of the behavior of syslog.

Loss of kernel log We show that the proposed system can gather logs under the condition of a massive output of kernel logs even if the existing system cannot.

The size of the kernel log buffer of the standard kernel on Debian 5.0.3 is 131,072 bytes. To exhaust this buffer, the program outputs logs with a total size greater than that of the buffer by printk (kernel logging function). One output has a size of 21 bytes, whereas in printk format, the size of the output is 38

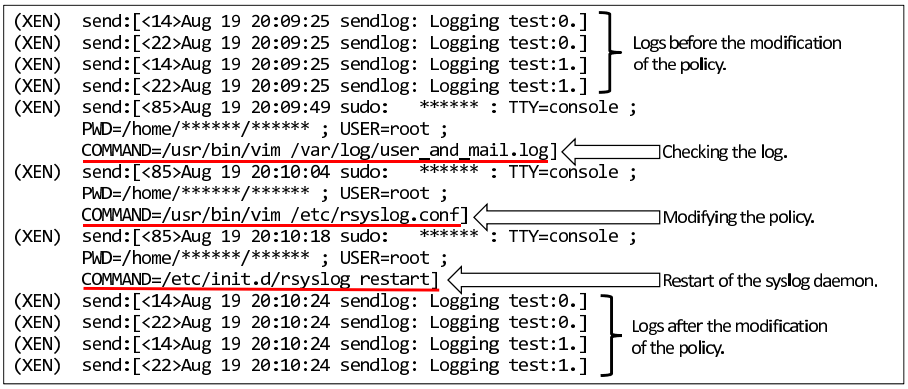

Fig. 4. A user log gathered by the proposed system.

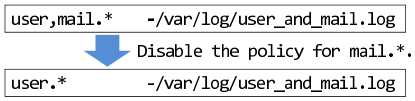

Fig. 5. Manipulation of configuration.

```
Nov 29 20:12:42 debian kernel: [   17.398956] lp0: using parport0 (interrupt-driven).
Nov 29 20:12:42 debian kernel: [   17.419593] ppdev: user-space parallel port driver
Nov 29 20:14:24 debian kernel: th world.
Nov 29 20:14:24 debian kernel: [  118.900091] Hello   51th world.
Nov 29 20:14:24 debian kernel: [  118.900098] Hello   52th world.
```

Fig. 6. A kernel log gathered by the existing system.

bytes. Thus, in order to exhaust the buffer, the program needs to output the log 3,450 times. In this experiment, we compared the logs gathered by the existing system and our proposed system after 4,000 outputs of the log.

Figure 6 shows the log in the MOS, and Fig. 7 shows the log gathered by the proposed system. In the third row of Fig. 6, the log is unusual because something has been lost through overwriting. In contrast, Fig. 7 shows that the proposed system has correctly gathered all the logs with no interruption. Whereas the existing system could not gather logs before the fiftieth output, the proposed system gathered all the logs. Thus, we have shown that the proposed system can gather all logs, even if they are lost by the existing system.

5.5 Detection of tampering and loss

The comparison of logs gathered by the existing and proposed systems enables us to detect any tampering with or loss of logs. Therefore, we show that it is possible to detect tampering by comparing these logs.

```
(XEN) KERNLOG:<6>[   17.398956] lp0: using parport0 (interrupt-driven).
(XEN) KERNLOG:<6>[   17.419593] ppdev: user-space parallel port driver
(XEN) KERNLOG:<4>[  118.899567] Hello    0th world.
(XEN) KERNLOG:<4>[  118.899567] Hello    1th world.
(XEN) KERNLOG:<4>[  118.899567] Hello    2th world.
                                  ⋮
(XEN) KERNLOG:<4>[  118.900085] Hello   50th world.
(XEN) KERNLOG:<4>[  118.900091] Hello   51th world.
(XEN) KERNLOG:<4>[  118.900098] Hello   52th world.
```

Fig. 7. A kernel log gathered by the proposed system.

In preparation, we tamper with the log file. After that, we compare the logs gathered by the existing and proposed systems. This comparison indicates that the proposed system is able to detect tampering.

For this comparison, we use `sudo` as a program that uses syslog to output user logs. If a command is executed with `sudo`, the user name, tty, and the name of the command are sent to the syslog daemon. At this point, the information in the log gathered by the existing system and the proposed system is the same.

To hide the executed command, we modify the name of the command in a log in the existing system. Now, comparison between the logs allows a difference in the name of the command to be detected.

As a result, it is seen that tampering with a log in the MOS has no effect on the logs in our proposed system. Moreover, the experiment shows that a part tampered with in the MOS can be detected through comparison between the logs in the MOS and in the proposed system.

5.6 Overheads

The user log collector hooks all system calls in the MOS. In this mechanism, the VMM is raised upon each system call, and therefore, the processing time of the system call increases. To determine the overheads of the system, we measured the overheads in some of the system calls that are mainly invoked in the syslog function and the function itself. The results are shown in Tables 2 and 3.

Table 2 shows the overheads of connect and write system calls that were invoked in syslog function. To show the overhead when switching between the MOS and VMM, we also measured the overhead of the getpid system call. From the margin between the overhead of getpid and other system calls, it is found that switching between the VMM and MOS takes about 2 μs. In invoking a connect system call, our system decides the target of the connect system call. The measurement shows that this decision takes about 3 μs. In the write system call, the proposed system takes about 47 μs. This overhead derives from copying of the message between the MOS and the VMM.

In addition, since the difference between the overhead in the syslog function in Table 3 and the write system call is 5 μs, it can be concluded that the main reason for the overhead in the syslog function is the write system call. Table 3

Table 2. Overheads in the proposed system when a system call was invoked (μs).

	connect		write		getpid	
	ave	overhead	ave	overhead	ave	overhead
unmodified	1.16	–	92.90	–	0.23	–
modified	6.55	5.39	142.20	49.30	2.23	2.00

Table 3. Overheads in the proposed system when a syslog and printk are invoked (μs).

	syslog		printk	
	ave	overhead	ave	overhead
unmodified	102.39	–	9.89	–
modified	156.56	54.17	67.34	57.45

also shows the overheads of the kernel log collector. The overhead of the proposed system in the printk function is 57 μs. This result is similar to that for the syslog function. The reason is thought to be that the printk function copies data to the memory in the same way as the syslog function. It can be estimated that the 10 μs overhead derives from the breakpoint exception.

From these measurements, such overheads have little effect on performance.

6 Discussion

6.1 Opportunity for gathering kernel logs

The proposed system sets a breakpoint at the starting point of the kernel logging function for the opportunity to gather kernel logs. Thus, the system gathers old logs just before the output of current logs, the logs are older than current ones.

In order to acquire the latest logs, it is necessary to set the breakpoint immediately after the output. The current implementation uses the starting address of the kernel logging function from the System.map. Starting points of functions are described here, so it is easy to set a breakpoint. In order to gather logs immediately after the output of a kernel log, we need the returning point of the function. But to acquire the point, we need to analyze the kernel. Although the method can gather latest logs, there is a problem in terms of analyzing the kernel each time it is updated. The method of the proposed system only requires System.map, so there is no difficulty arising from kernel updates.

6.2 Security of logs in logging path

To guarantee the integrity of a log, it is necessary to ensure the security of the logging path from its output to the time it is stored in a file. Here, we compare the security of the logging paths of the existing and proposed systems.

Firstly, we analyze the logging path of a user log. A user log might be attacked at the following points:

(1) The time when a user process generates a log
(2) The time between the sending of a log and its receipt by syslog
(3) The time between the receipt of a log and storing it to a file
(4) After the output of a log

The existing system cannot detect and prevent tampering or the loss of logging information at any time. In the proposed system, time (1) is the only possible time when attacks might be suffered. An example of an attack at time (1) would be someone tampering with the program itself. To protect the logs from tampering in this case, it is necessary to ensure the integrity of all programs that generate logs. DigSig [1] is a method that ensures the integrity of a program by assigning a signature to the program. However, this method does not satisfies our demand because it modifies the kernel codes. Moreover, the method causes a large overhead. For time (2) to (4), some research has been carried out, but those methods do not satisfy our demand. The method ensures the integrity of syslog has proposed [4], but the method needs logging server, so if the network is down, it is unavailable. For time (4), hysteresis signature enables us to detect tampering with a log, but it has a problem mentioned in Section 2. In addition, to prevent such tampering, protection methods using the file system already exist [10]. However, these methods modify the source codes of the kernel, so our demand is not satisfied.

Secondly, we analyze the logging path of a kernel log. A kernel log might be attacked at the following moments:

(1) The time to generate a kernel log in a kernel
(2) The time to output the log to a kernel log buffer
(3) While stored in the kernel log buffer
(4) The time during which a kernel logging daemon gathers a log
(5) While the kernel logging daemon sends the log to syslog
(6) While syslog stores the log to a file
(7) Afther the output of a log

In the existing system, it is impossible to protect a log from an attack by a rootkit at any time. Furthermore, there is a possibility of attack similar to the logging of a user log if the kernel is safe. The proposed system gathers a log at time (2). We can consider tampering with the kernel logging function as an example of an attack at time (2). However, the log gathered by the proposed system is the previous one. Therefore, the logs might be attacked at time (3). Thus, the proposed system can address attacks on and after time (4). The improvement of the proposed system for gathering a log immediately after its output enables it to address time (3) as well.

However, to prevent attacks before time (3), it is necessary to modify the parts related to kernel logging. In this situation, our demand is not satisfied.

SecVisor [9] is a technology that ensures the integrity of the kernel, and is effective for the prevention of attacks at times (1) and (2). However, it is impossible to prevent attacks after time (3) with SecVisor alone. Thus, since the proposed system can prevent attacks on or after time (4), the security of logging is ensured unless the kernel logging function itself is attacked.

6.3 Applicability for other OSes

No modification of the kernel code in the MOS is necessary because the proposed system is implemented in a VMM. For this reason, the system can be applied to any kind of OS. Furthermore, if an OS fulfills the following requirements, the proposed system can be applied to an OS that is not open-source software. Here, we discuss the possibility of applying the system to other OSes.

Regardless of how the system is constructed in terms of the logging and storing modules, the storing module is independent from the MOS. Thus, in the adaption of the system, we need to consider the requirement for implementing the logging module. Considering the case of Windows, a widely used OS, the following requirements can be noted:

(1) Use of sysenter instruction in invoking a system call.
(2) Identification of the system call is used in logging.
(3) Identification of the starting address of the kernel logging function.
(4) Identification of the area of the kernel log buffer.

Requirements (1) and (2) are necessary for the gathering of user logs, and (3) and (4) for the gathering of kernel logs.

In Windows XP or later, system calls are generally implemented with sysenter. Moreover, since the identifier of the system call is stored in the EAX register in Windows, the detection of a system call for logging is available with its value. Requirements (3) and (4) can be achieved by analyzing the kernel.

On the basis of these considerations, it is concluded that the proposed system can be applied to various OSes (e.g., Windows) if the requirements are fulfilled.

7 Conclusion

This paper describes a logging system with a VMM to prevent log tampering and loss. In the user log collector, the VMM detects an output of the log of the MOS and gathers it before it is gathered by syslog. Because the proposed system gathers logs immediately after the logging request, there is no opportunity for tampering. Also, because the kernel log collector gathers logs in conjunction with the kernel log's output, the system can prevent the loss of logging information caused by the overwriting of old kernel logs. These functions enable the VMM to gather logs of the MOS with no modification of its source codes. Moreover, the system is independent of the MOS because it is implemented as a VMM. Thus, it is difficult to attack the proposed system.

This paper also presents an evaluation of the proposed system, assuming that tampering and loss of logging information occurs. Considering the results of these evaluations, it is proved that it is possible to detect and prevent tampering with log files. Furthermore, the proposed system can address the problem caused by the structure of the kernel log buffer.

This paper also describes the evaluation of the overheads caused by the system. The results show that the overheads associated with the proposed system is only about 50μs at most.

Acknowledgements This research was partially supported by Grant-in-Aid for Scientific Research 21700034 and a grant from the Telecommunications Advancement Foundation (TAF).

References

1. Apvrille, A., Gordon, D., Hallyn, S., Pourzandi, M., Roy, V.: Digsig: Runtime authentication of binaries at kernel level. In: Proceedings of the 18th USENIX Conference on System Administration. pp. 59–66 (2004)
2. Ashino, Y., Sasaki, R.: Proposal of digital forensic system using security device and hysteresis signature. In: Proceedings of the Third International Conference on International Information Hiding and Multimedia Signal Processing (IIH-MSP 2007) - Volume 02. pp. 3–7 (2007)
3. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles. pp. 164–177 (2003)
4. Bock, B., Huemer, D., Tjoa, A.: Towards more trustable log files for digital forensics by means of "trusted computing". In: 24th IEEE International Conference on Advanced Information Networking and Applications. pp. 1020–1027 (2010)
5. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and Communications Security. pp. 51–62 (2008)
6. IETF Syslog Working Group: IETF Syslog Working Group Home Page. `http://www.employees.org/~lonvick/index.shtml`
7. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. `http://www.intel.com/Assets/PDF/manual/253669.pdf` (2009)
8. Isohara, T., Takemori, K., Miyake, Y., Qu, N., Perrig, A.: Lsm-based secure system monitoring using kernel protection schemes. In: International Conference on Availability, Reliability, and Security. pp. 591–596 (2010)
9. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles. pp. 335–350 (2007)
10. Takada, T., Koike, H.: Nigelog: Protecting logging information by hiding multiple backups in directories. International Workshop on Database and Expert Systems Applications pp. 874–878 (1999)
11. Zhao, S., Chen, K., Zheng, W.: Secure logging for auditable file system using separate virtual machines. In: IEEE International Symposium on Parallel and Distributed Processing with Applications. pp. 153 –160 (2009)