Analysis of the Linux Audit System[1]

Authors
Bruno Morisson, MSc (Royal Holloway, 2014)
Stephen Wolthusen, ISG, Royal Holloway

Overview

Audit mechanisms on an operating system (OS) record relevant system events to provide information for analysing the trustworthiness of the system. This is especially important for detecting or investigating potential compromises of a system.

In Linux based Operating Systems, the standard framework for auditing is the Linux Audit Subsystem. It generates, processes, and records relevant audit events either from within the kernel or from user-space programs. In this article we identify serious flaws due to architectural limitations of the Linux kernel which cast doubts on its ability to provide forensically sound audit records. We also examine these limitations and discuss possible mitigation methods.

The importance of operating systems auditing

Audit mechanisms on an operating system (OS) are a set of functionalities that record relevant system events that can be used to understand the status of that system regarding its assurance in a certain point in time. It is a critical mechanism that should be able to provide enough information to allow for the analysis of the trustworthiness of the system, and therefore the reliability of information it processes or stores. They become especially relevant when detecting or investigating potential compromises of a system.

The logs generated by the audit mechanisms allow the accomplishment of several security objectives, such as individual accountability, reconstruction of events, intrusion detection and problem analysis. They can work as a deterrent, since they provide individual accountability, and the knowledge that an effective audit mechanism is in place may be enough to keep potential attackers out. They also work as a detection mechanism - by providing the ability to reconstruct events, audit records are typically the only possible way to obtain insight into the events that preceded a security breach, and to understand the initial point of compromise. Since the monitoring of audit records may reveal the existence of abnormal activities that can be further investigated and acted upon, audit mechanisms also serves as a preventative mechanism.

[1] This article is to be published online by Computer Weekly as part of the 2015 Royal Holloway info security thesis series. The full MSc thesis is published on the ISG's website.

The Linux Audit System

In Linux-based operating systems, the standard framework for auditing is the Linux Audit System (LAS). The LAS is comprised of a subsystem in the Linux kernel and a set of user-space tools. It handles the generation, processing, and recording of relevant audit events from within the kernel or from user-space programs, according to a defined policy.

The Linux kernel includes the audit subsystem originally developed by RedHat for the RedHat Enterprise Linux 4 EAL certification, which was added to the main kernel development in 2004, and originally named ”lightweight auditing framework” (in the Common Criteria TOE documents this nomenclature is still used). This subsystem aims to provide auditing capabilities to the kernel, enabling security events to be generated and passed on to user-space for logging. It performs the logging of system calls (*syscalls*), and provides a framework for generating audit events that can also be used by Linux security modules, such as SELinux. The subsystem implements hooks on the kernel *syscalls*, so that when they are invoked by a user-space process an audit event is generated according to the policy defined in the system. Events are then passed on to user-space, and logged on the filesystem as simple ASCII files.

The LAS also provides a user-space library, *libaudit*, that allows trusted programs in user-space to perform logging using this framework. By invoking the functions in the library, these programs may send audit messages into the kernel and allow them to be logged according to the same policy and mechanism.

Since the actual logging (writing of audit events into a file) itself is performed in user-space, and since trusted user-space tools may also use the framework for logging, the kernel provides a *netlink* socket for bidirectional communication between the kernel and the user-space.

As such, the LAS is more than simply a kernel functionality. It is made up of the kernel functionality it provides, as well as a library and tools to process the events, interact with the kernel, and analyse the recorded events log files. A diagram of the audit system (from “Track KVM guests with libvirt and the Linux audit subsystem”, Cerri, 2012) is shown in Figure 1.

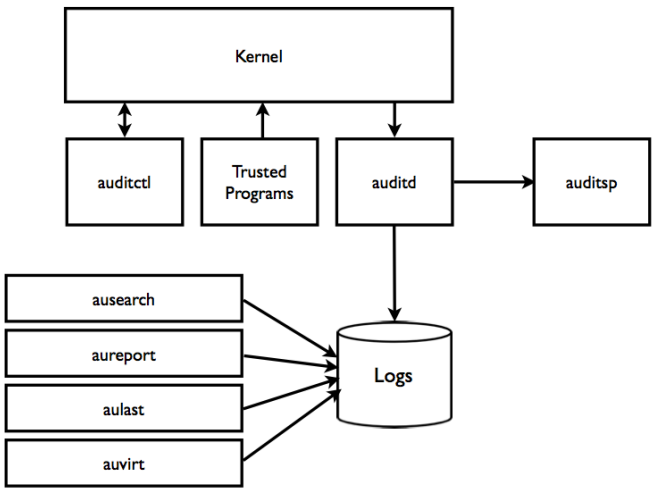

Figure 1 - The Linux Audit System

According to its developers, the LAS should be able to record the following:

- Date and time of event, type of event, subject identity, outcome
- Sensitivity labels of subjects and objects
- Be able to associate event with identity of user causing it
- All modifications to audit configuration and attempted access to logs
- All use of authentication mechanisms
- Changes to any trusted database
- Attempts to import/export information
- Be able to include/exclude events based on user identity, subject/object, labels, other attributes

In practice it simply provides a framework with the ability to record those events and information, which in some cases is only possible when using a Linux security module (LSM). For instance, the recording of sensitivity labels is not even applicable without SELinux, since there exists no labels on a typical Linux system based on discretionary access control (DAC).

Kernel Functionality

Inside the kernel, the LAS implements a set of functionalities that are the core of the framework:

- Audit functions available inside the kernel
- System call auditing
- *Netlink* socket for communication with user-space
- Audit policy enforcement

The subsystem provides functions that can be used by other parts of the kernel, such as loadable modules or Linux security modules.

User-Space Functionality

User-space auditing functionality is based on a few components that provide control over the audit subsystem and a framework that allows any trusted program to use the audit subsystem's functionality and features. All communication to and from the kernel is done through the *netlink* socket.
Figure 2 provides an overview of the main user-space components we addressed in our research.

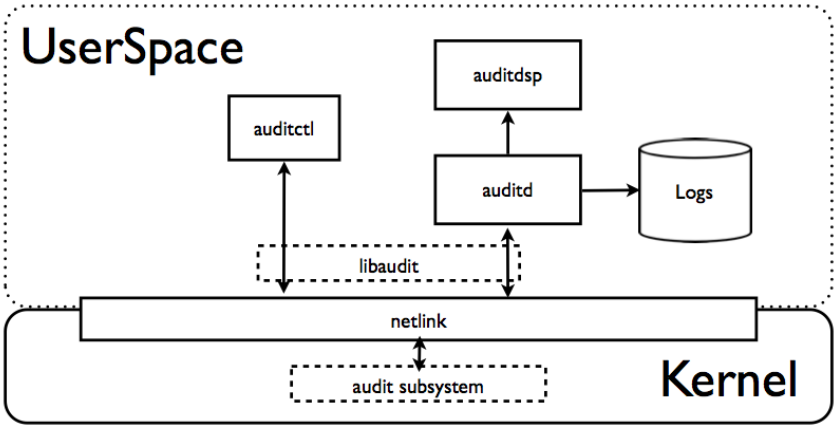

Figure 2 - Linux Audit System – main components

Silently disabling the audit system

Auditing can be enabled and disabled by using the *auditctl* tool, or by communicating directly with the kernel through the audit *netlink* socket. The communication primitives are available by way of the *libaudit* library, which facilitates integration of applications with the audit subsystem. Enabling or disabling auditing will produce an audit event which is logged.

However, we have identified two methods for disabling the auditing without generating any event. Both methods require that the attacker has already somehow compromised the system and obtained root (user ID 0) privileges, which allows him to interact with the audit subsystem, and enable or disable it. These attacks are relevant since even if the log files are being securely stored in a *WORM* (*Write Once Read Many*) media or on a remote system, an attacker is able to disable the auditing and only then perform actions without leaving any evidence in the audit logs. One of the attacks is performed purely on user-space by using the functions provided by *libaudit*, and allows for silently disabling the audit subsystem. The second attack, however, takes advantage of the Linux loadable modules functionality, effectively disabling the audit subsystem directly from kernel-space. It has the added advantage of being able to disable the auditing even if it is ”locked”, which according to the documentation should only allow for changes (including disabling) after a reboot.

We now detail both the attacks in the following subsections.

Silently disabling the audit system from user-space

For this attack, we have developed a custom tool linked to *libaudit* that uses the regular functions from this library to disable the audit subsystem from user-space, ensuring that no audit events are logged.

The process takes four steps, as shown in the following figure:

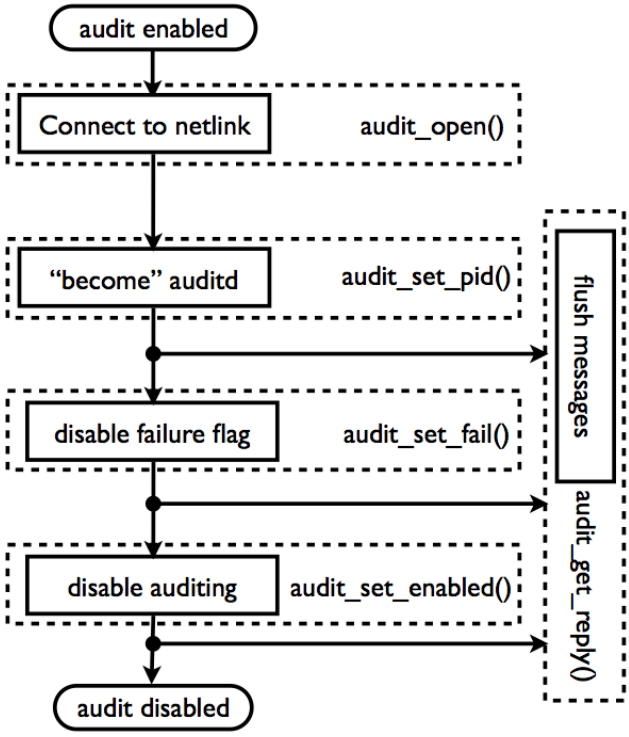

Figure 3 - disabling from userspace

Connecting to the audit *netlink* socket is done by calling the function audit_open(). We then take over the handling of the audit messages sent from the kernel, by calling the function audit_set_pid(). The audit_set_pid() function tells the kernel the process ID (PID) of the *auditd* program, which is the one handling the audit messages. By calling this function and passing the process id from our own tool, *auditd* will stop receiving messages and we will assume the handling of those messages. We then disable the failure flag - this stops the displaying of audit events on the console in case of failure contacting the *auditd* process. This ensures that after our tool ends, no further audit events will get logged. Finally, audit is disabled with the audit_set_enabled() function.

Between all the audit_set_* functions called, we call the audit_get_reply() function. This function ensures the flushing of the audit events generated from our own actions reconfiguring the audit subsystem, also ensuring that they will get read by our tool, and ignored.

Silently disabling the audit system from kernel space

The Linux kernel allows for loading modules into the kernel, which is a functionality typically used by device drivers. These modules are an extension to the kernel itself, allowing access to full kernel memory.

In order to disable auditing from inside the kernel, we have developed a loadable kernel module which will look for the relevant control variables used in the audit subsystem, and set them to 0, so the audit subsystem will effectively be disabled. The technique is similar to the techniques

used by kernel-based rootkits, where the objective is to manipulate the kernel memory in order to control specific parts of the system.

The main variables that control the audit subsystem are the following:

- audit_enabled - This variable is verified before every action in the audit system that would generate an audit message / event. If it is set to 0, the system is disabled, if set to 1 is enabled, and if set to 2 it is locked (i.e., unchangeable);
- audit_fail - The audit fail variable controls the behaviour of the audit system in cases when the *auditd* process is unavailable. If set to 0 lost events are ignored, if set to 1 they are sent to the console using the kernel printk function, and if set to 2 it will generate a kernel panic and stop the operating system;
- audit_pid - This variable holds the process ID of the *auditd* process that is responsible for processing the audit events in user-space.

These variables are not exported to the kernel and are not directly accessible by kernel modules by name. To access them, kernel memory must be searched for all the symbols and the addresses identified. The Linux kernel provides a function, kallsyms_lookup_name, that will return the memory address of a given symbol name (function of variable). However, the kernel, depending on the version, does not always export this function. To find it, we implemented a similar function that will look for kernel symbols, using a different function that is exported and that provides roughly the same functionality. This function, kallsyms_on_each_symbol, will iterate over all the symbols in the kernel, and for each one will call a handling function. This handling function will then check if the symbol name is the one we are looking for, and if so saves the address to a variable local to the module. This allows us to identify the address of the kallsyms_lookup_name function and proceed with searching for the audit subsystem control variables. After identifying the addresses of the variables, we are then able to set them to 0, thus completely disabling the audit subsystem, which will no longer generate audit events. This flow is shown in the following figure.

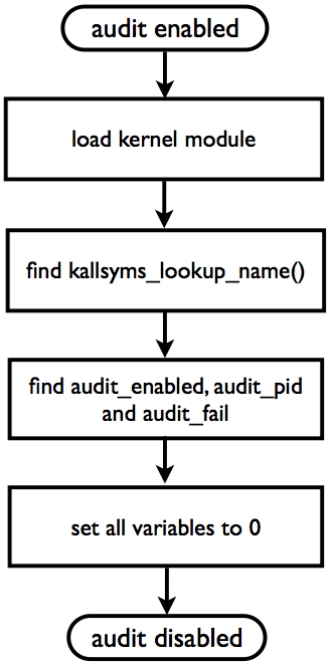

Figure 4 - Disabling the audit system from kernelspace

It is important to notice that although in our proof of concept we perform the search for the symbols' addresses in kernel-space, kernel symbols addresses may be found from user-space via the proc pseudo-filesystem, in /proc/kallsyms or /proc/ksyms, in the System.map file (if available) or by analysing the kernel image (vmlinuz/vmlinux).

With this approach, it becomes possible to disable auditing even if the audit subsystem is in the "locked" state (i.e. immutable). As previously mentioned, in this state it should not be possible to disable auditing without rebooting.

Improving The Linux Audit System

Authenticating trusted processes

One of the issues that we first noticed while analysing the architecture and functionality of the LAS, was the lack of any authentication of user-space tools when communicating with the kernel. As we detailed previously, communication to and from the kernel is performed through a *netlink* socket that handles all the messages between user-space and kernel-space. Furthermore, the LAS also doesn't validate or authenticate the user-space programs that communicate with the kernel. Any program running with user id 0 can perform any action on the audit subsystem, not only send and receive data (audit events), but also control the audit subsystem, being able to stop it or change the audit policy, for example.

Since version 3.7 the Linux kernel includes the Integrity Measurement Architecture (IMA), which aims to provide support for authentication of binaries on the system by validating hashes or digital signatures of those binaries, so that tampering can be detected. This implementation also supports the use of a Trusted Platform Module for key storage and signing. By using the functionality provided by the IMA, the LAS could implement an authentication mechanism for the processes interacting with it, and not simply trust any program running under user id 0.

Protecting the audit state

The attack we detailed allowing for silently disabling auditing using a loadable kernel module (LKM), is only possible since a LKM is able to access exported kernel symbols and have full control over them, being able to change their contents without any restriction or even being detected.

Hiding kernel symbols

An approach that mitigates our attack is available in the grsecurity (GRSec) kernel patches. One of these patches allows for hiding the kernel symbols from loadable kernel modules. By implementing this kernel patch, loadable kernel modules will not be able to find the correct addresses for symbols, and in the case of the attack performed, it will not be possible to find the needed symbols in order to change their values and effectively disable auditing.

Other approaches for protecting kernel symbols from attackers, specifically rootkits, have been published. These approaches include the randomization of kernel data structures at compile time, making it harder for loadable kernel modules to identify the correct addresses of kernel structures they wish to access or modify. We believe that these techniques may mitigate the attack we presented.

Monitoring the audit status

For the purpose of mitigating this attack, using a different approach, we have developed a proof of concept LKM. The objective of this module is to monitor critical symbols in the kernel, specifically audit_enabled, in order to detect and record any changes to the audit state, and also ensure and enforce the immutability of the locked state. For this purpose we used the hardware breakpoint functionality available in the Linux kernel. This functionality allows for setting a watch on a specific kernel symbol, and define a handler for when that symbol is accessed and/or modified which will provide an alert and/or reset the audit state to the previous one.

Although using hardware breakpoints may be an effective approach, the number of debug registers on a processor is limited, and varies according to the processor. Since debug registers are used for setting the breakpoints for the symbols we want to protect, the number of debug registers effectively limits the number of symbols possible to watch. For instance, an x86 processor is limited to 4 debug registers. Also, not all processor architectures support hardware breakpoints.

Additionally, an attack such as the one implemented by our proof of concept loadable kernel module could be performed on the protecting mechanism itself. An attacker could access the kernel structures holding information regarding the set breakpoints, and disable all breakpoints configured, effectively rendering the protection useless.

For this reason, monitoring the protection mechanism itself is required, in order to ensure its own integrity and reliability.

Conclusion

Audit systems are an essential part of any operating system, and should provide enough information that is both reliable and accurate about events that have happened in the system to be analysed if and when required.

In the case of Linux systems, we have identified that any privileged user or process can control the audit system, effectively impacting its ability to provide reliable audit events. Privileged users and processes can subvert the audit system by disabling it without generating any audit event, and as such there will be no evidences of this action, and subsequent events will not be recorded and logged.

The issues identified are architectural problems in the Linux kernel itself that are not easily solved. For instance, the fact that loadable kernel modules have unrestricted access to all kernel symbols, or the fact that all programs running with user id 0 (”root”) end up having full control over the whole operating system.

Based on the results observed in this research, we believe that the Linux Audit System is currently unable to provide audit records that are reliable or accurate enough to forensically analyse a system in the cases where that system was compromised, and unable to prevent or aid in identifying fraudulent or malicious actions performed by privileged users.

Biographies

Bruno Morisson is a Partner and Director at INTEGRITY S.A. where he is responsible for the penetration testing services. He has over 14 years experience in the industry, and has been involved in the infosec community for as long as he can remember. He founded and still manages the only active portuguese infosec mailing list, organizes monthly meetings of infosec professionals and the BSidesLisbon security conference, and has contributed modules to the Metasploit Framework. Bruno holds a MSc. with Distinction in Information Security from Royal Holloway, University of London, and several industry certifications such as the CISSP-ISSMP, OSCP, CISA and ISO27001LA.

Dr Stephen Wolthusen received his Dipl.-Inform. degree in computer science in 1999 and completed his Ph.D. in theoretical computer science in 2003, both at TU Darmstadt. He was with the Security Technology Department at Fraunhofer-IGD from1999-2005 serving as deputy division chief from 2003 onwards and as senior visiting scientist from 2005 onwards. He is currently a Reader in Mathematics with the ISG, and also Full Professor of Information Security (part-time) at Gjøvik University College, Norway. His research focuses on models of adversaries and resilient networks, with applications in defence networks and particularly in critical infrastructure networks and control systems security. He has led a number of national and European projects, including the Internet of Energy project. He is author and editor of several books as well as over 100 peer-reviewed publications. He has served as editor-in-chief of Computers and Security and as vice-chair of the IEEE Task Force on Information Assurance and is currently vice-chair of the IEEE Task Force on Network Science.