Linux auditd for Threat Detection [Part 1]

IzyKnows Follow 11 min read · Jan 26, 2022

Part 2: Linux auditd for Threat Detection [Part 2]

A few years ago, I was asked to define an auditd configuration which would serve as the primary detection technology for a large organization. While I had a fair understanding of Linux systems, I found surprisingly little on utilizing auditd for at-scale security monitoring purposes.

The topics I look to cover in this article are

- Quick intro to the Linux Audit System
- Tips when writing audit rules
- Designing a configuration for security monitoring
- What to record with auditd
- Tips on managing noise

The end audience for this article is the security practitioners, IR, blue teamers who need to define what logs they need in their SIEM to detect bad activity. Going forward, I'm hoping things will get a lot easier with ePBF but we've never been good at completely eradicating older technologies now have we? :)

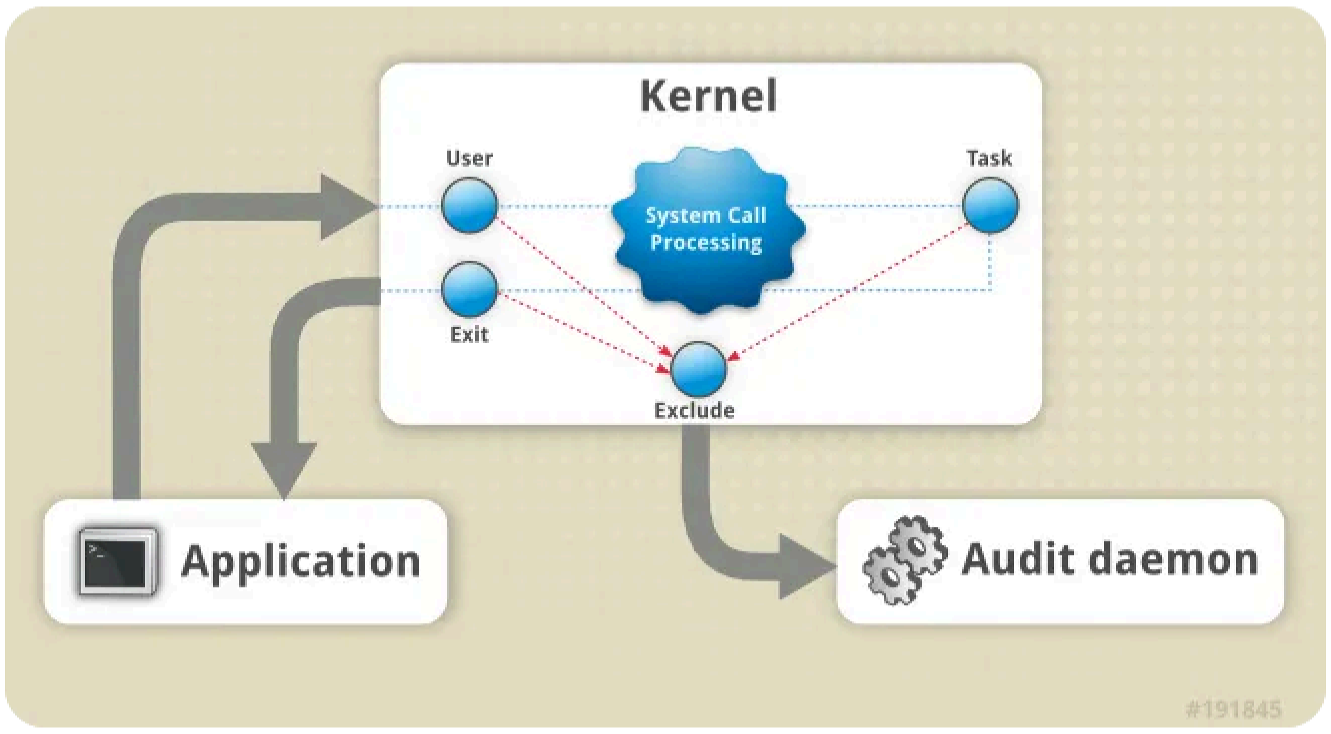
There may be better ways to configure and utilize auditd, so feel free to share your experiences and I'd be happy to incorporate them here.

. . .

The Linux Audit System

The Linux Audit system provides a way to log events that happen on a Linux system. The recording options offered by the Audit system is extensive — process, network, file, user login/logout events, etc. In this series, I only focus on the security-relevant events from a detection standpoint. The entire list of types that can be recorded are listed here. This list will come handy when analyzing logs.

By default, the packages required to use the Linux audit system are installed on many common distros. I won't be covering the installation part here. "auditd" is the **audit d**aemon that leverages the Linux audit system to write events to the disk. User-space applications make system calls which the kernel passes through certain filters and then finally through the "exclude" filter. The filters are important when it comes to writing auditd rules. I really like the diagram from Red Hat explaining it.



Audit System Architecture (source: RedHat)

By default, log files are written to /var/log/audit/audit.log. You may find multiple audit.log files in the above directory all of particular max length.

The two files we particularly care about in the audit system are

- audit.rules: Tells the audit daemon what to record. This is where most of one's time should go — deciding what events are most important to you
- audit.conf: Governs how the audit daemon runs. Log file location, buffer size, log rotation criteria, etc. I would not mess around too much with this file, but we'll get to some important parameters of it later

The above was just a brief intro. There are better explanations of the Audit subsystem out there. The link to RedHat's docs I mentioned above is a good one. Other good documents I can recommend are — Understanding Linux Audit from Suse, The Linux Audit Documentation Project

Writing Audit rules

What you specify in the audit.rules file is what finally lands up in the audit.log. When the audit daemon (auditd) is started (ensure that's on system startup), rules defined in this file is what gives you events in the audit.log.

The audit.rules file is located in /etc/audit/audit.rules. This is the final file that auditd refers when writing events to disk. Why I mention this is because, as mentioned at the top of the etc/audit/audit.rules file,

> *## This file is automatically generated from /etc/audit/rules.d*

That doesn't mean you can't add to it manually. However, in a large environment, you may have different rule files managed by different parties, that need to run together. You can place these different *.rules files within the /etc/audit/rules.d/ directory. A utility named *augenrules* does the job of compiling the different *.rules files (in natural sort order) into the final /etc/audit/audit.rules . It's worth mentioning that augenrules strips away comments and empty lines before generating audit.rules. I mention this as I've been accused of ridiculous things on this point in the past.

Coming to the rule writing. Let's focus on the 2 types of rules that we're interested in configuring

- File watches — can watch read, write, execute or attribute changes
- Syscalls — record syscalls sent to the kernel by the application

I'm not going to go into the depths of writing auditd rules, I recommend reading the man page for that.

Some good references to look into when writing rules

- Inbuilt examples — https://github.com/linux-audit/audit-userspace/tree/master/rules
- MITRE-based rules — https://github.com/bfuzzy1/auditd-attack/tree/master/auditd-attack
- Florian Roth's best practice ruleset — https://github.com/Neo23x0/auditd
- https://slack.engineering/syscall-auditing-at-scale/

A few notes to keep in mind when building rules

- Utilize tagging — audit rules allow for tagging (-k) which is very helpful when analyzing events later. You could use your own custom tags or even tag rules with MITRE IDs (see bfuzzy1's link above)
- When monitoring syscalls, it's often better to monitor the syscall upon exit rather than entry (-always,exit). Important parameters may not be available at the time of function entry because they aren't defined yet, hence you miss it too
- When doing syscall auditing always try to combine rules where you can (-a always,exit -S rmdir -S unlink -S rename). Each syscall rule gets

evaluated for every syscall every program makes. It adds up, thereby impacting performance

- When writing rules, you may come across a filter criteria '-F auid!=4294967295'. This number is the equivalent of 0xFFFFFFFF which is the highest unsigned int number. It evaluates to -1 which in auditd world is equivalent to "unset", i.e, it's not defined yet. It's common for this to happen with processes that initialize before the audit daemon. As in the audit.rules man page —

> *"The audit system considers uids to be unsigned numbers. The audit system uses the number -1 to indicate that a loginuid is not set. This means that when it's printed out, it looks like 4294967295. But when you write rules, you can use either "unset" which is easy to remember, or -1, or 4294967295. They are all equivalent."*

- A note on exclusions — ensure your exclusions (-a never,exclude) are specific and placed on top so they get matched first. There are debates on whether specific includes or excludes should go on top. Specific includes would be the best for performance but I don't find this a feasible option for large environments. Often times, the syscalls (like execve) are where you gain maximum visibility from a security detection perspective. Were you to place them right on top, you'd have little scope for performance tuning thereafter.

Configuration of the Audit Daemon

There are two places you can specify configurations for the audit daemon. One place is directly at the top of the audit.rules file as control rules and the other is in the audit.conf file. The configurations you can modify in both these places differ. Unless for a specific reason, I would not toy with the configurations of auditd except for a few

Backlog size: This can be set by using the -b keyword followed by the number of audit messages to buffer. The backlog option limits the number of messages that can be queued up waiting to be written to the log. Here, it's recommended you start with the default and work your way up. On production level systems, I've seen 8192 as a feasible option. When the limit is breached, you should see a "backlog limit exceeded" in the logs and that can be your indicator to increase this buffer. Keep in mind, the higher buffer value, the more memory consumption on the system

Failure flag: Set with -f, it instructs auditd what to do when the above buffer is full. In production systems, you definitely don't want disruption hence I recommend setting this to 1 so it prints in the log and nothing else

Enabled flag: Set with -e, can help ensure your audit configuration is not modified, i.e, immutable. Setting it to 2 will ensure that any changes will be denied and logged. Only a restart of the system can change the configuration. Remember, ensure this rule is the last rule in your ruleset.

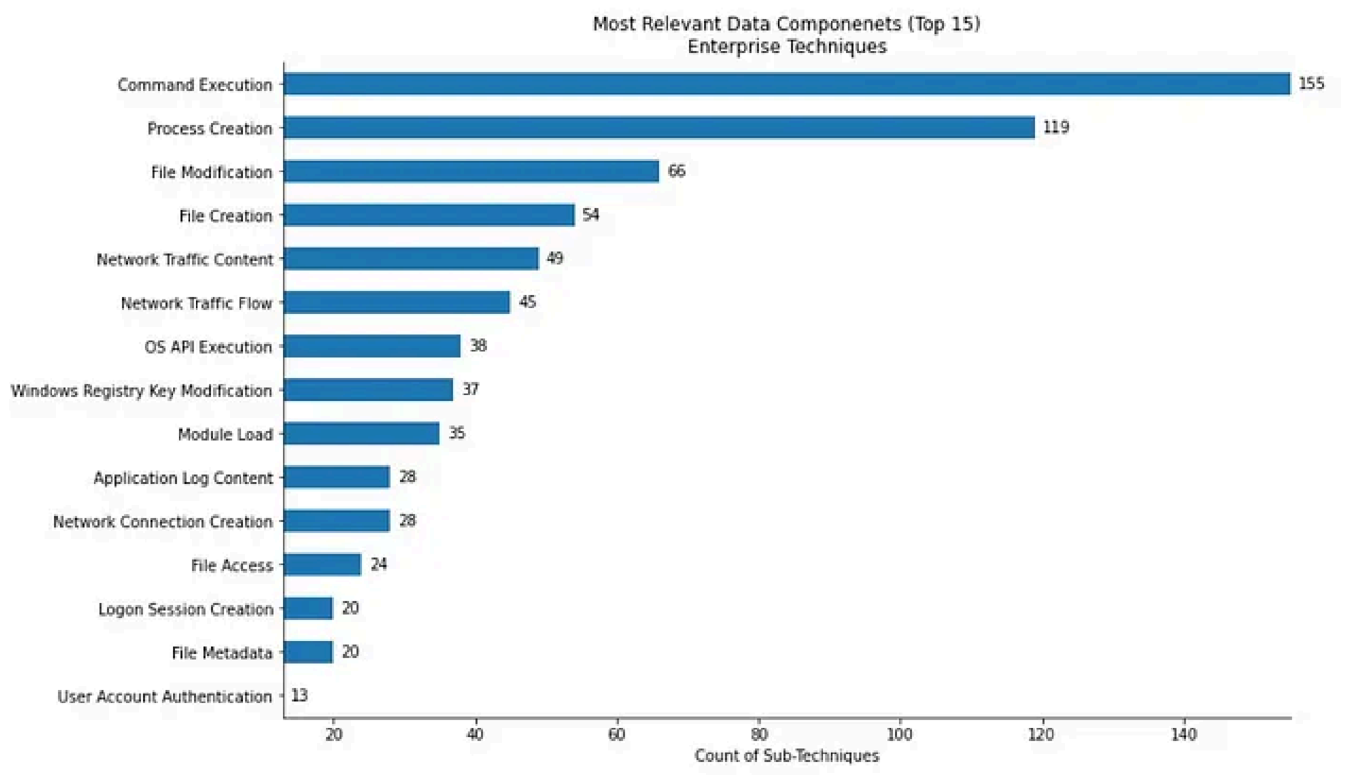
These are all the configuration files I'd recommend modifying. The audit.conf file you can have a look into but normally, I wouldn't change too much there. Depending upon your environment and if you want the auditd daemon to deal with, for example, what to do when the log file is full, the audit.conf file may be worth looking into. Normally, the default value have worked for me.

. . .

What to record?

The Linux audit subsystem allows us to record a lot of information. The question I look to provide some clarity to is — what's worth recording and what isn't?

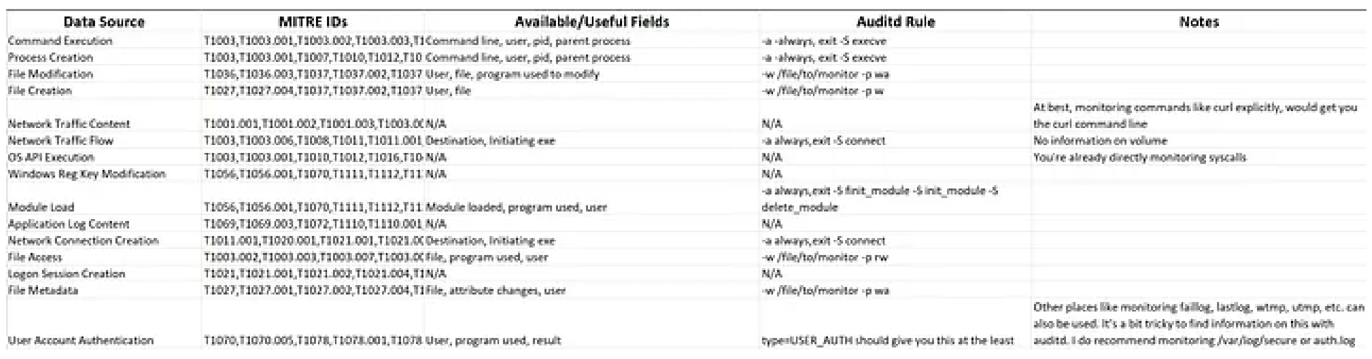
First, let's have a look at the most relevant data sources from MITRE.



Source: https://github.com/mitre-attack/attack-datasources

The above graph is based on Mitre's ATT&CK Enterprise framework and the data source that is associated by different sub-techniques in the framework. As we can see, majority of the techniques are detectable by command execution and process creation events. Reason being, these normally contain the command line executed.

Coming back to auditd, majority of these data sources can be monitored by auditd (at a price of course). Here's my shot at correlating the above list with auditd record types that give us this information.



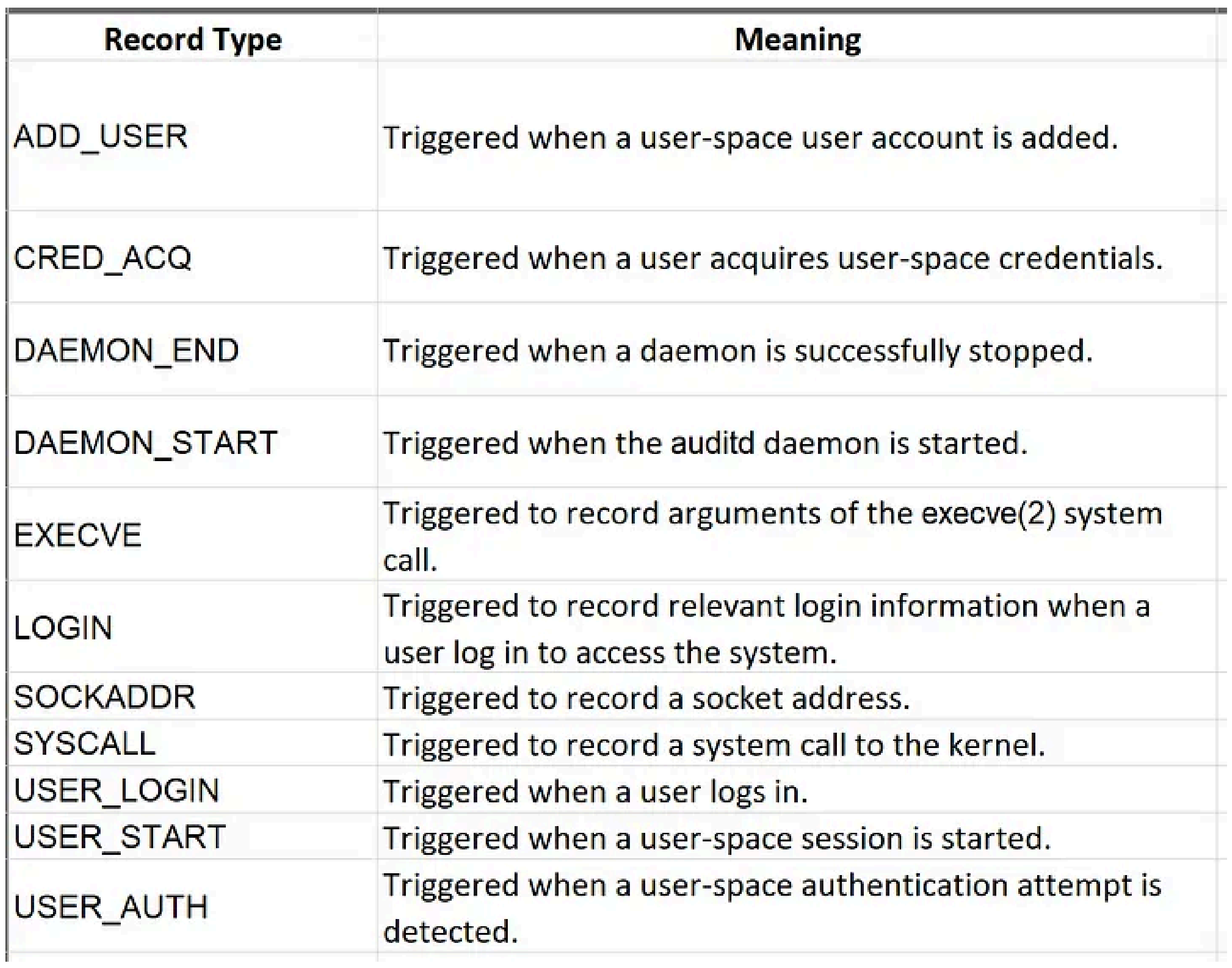
Data Source	MITRE IDs	Available/Useful Fields	Auditd Rule	Notes
Command Execution	T1003,T1003.001,T1003.002,T1003.003,T1	Command line, user, pid, parent process	-a -always, exit -S execve	
Process Creation	T1003,T1003.001,T1007,T1010,T1012,T10	Command line, user, pid, parent process	-a -always, exit -S execve	
File Modification	T1036,T1036.003,T1037,T1037.002,T1037	User, file, program used to modify	-w /file/to/monitor -p wa	
File Creation	T1027,T1027.004,T1037,T1037.002,T1037	User, file	-w /file/to/monitor -p w	
Network Traffic Content	T1001.001,T1001.002,T1001.003,T1003.0(	N/A	N/A	At best, monitoring commands like curl explicitly, would get you the curl command line
Network Traffic Flow	T1003,T1003.006,T1008,T1011,T1011.001	Destination, Initiating exe	-a always,exit -S connect	No information on volume
OS API Execution	T1003,T1003.001,T1010,T1012,T1016,T10	N/A	N/A	You're already directly monitoring syscalls
Windows Reg Key Modification	T1056,T1056.001,T1070,T1111,T1112,T11	N/A	N/A	
Module Load	T1056,T1056.001,T1070,T1111,T1112,T11	Module loaded, program used, user	-a always,exit -S finit_module -S init_module -S delete_module	
Application Log Content	T1069,T1069.003,T1072,T1110,T1110.001	N/A	N/A	
Network Connection Creation	T1011.001,T1020.001,T1021.001,T1021.0(	Destination, Initiating exe	-a always,exit -S connect	
File Access	T1003.002,T1003.003,T1003.007,T1003.0(	file, program used, user	-w /file/to/monitor -p rw	
Logon Session Creation	T1021,T1021.001,T1021.002,T1021.004,T1	N/A	N/A	
File Metadata	T1027,T1027.001,T1027.002,T1027.004,T1	File, attribute changes, user	-w /file/to/monitor -p wa	
User Account Authentication	T1070,T1070.005,T1078,T1078.001,T1078	User, program used, result	type=USER_AUTH should give you this at the least	Other places like monitoring faillog, lastlog, wtmp, utmp, etc. can also be used. It's a bit tricky to find information on this with auditd. I do recommend monitoring /var/log/secure or auth.log

Data Source to Audit Rule Mapping

Because Medium is a bitch with formatting, the above is an image. I've also uploaded the same here: https://github.com/izysec/linux-audit/blob/main/DS-to-audit.MD

The above list is just to help you prioritize your rule writing better. Auditd in general is going to be noisy and while exact impacts depends on your environment, if you had to keep a ballpark figure in mind, I would start with a 10–15% increase in terms of CPU utilization when running a security-centric auditd configuration on "production" systems. We can go into the nuances of the % and what "production" means but it's only a ballpark figure to help you set expectations right. Have no doubts, auditd is heavy, and depending on the nature of the endpoint you deploy on, you can experience "heavy" in different flavors. For example, there was once a production database server. The servers job was just to store data for a large public facing web application with several users. This meant millions of database write operations in a day. The auditd daemon was eating CPU and specifically the rules that monitored file-action related syscalls (open, for example). Since the nature of the database was to constantly perform several file operations, not only was the audit.log being filled up with minimally useful entries but everytime a file operation occurred (essentially always), auditd was monitoring and writing it into the logs. It's lucky that size of the audit.log is capped else, we would've had space issues as well. Eventually, we decided the best course of action was to write exclusions for certain folders that we know were constantly being written to and we placed this rule on top of the one causing issues so it would match first. We saw similar impacts with servers of a different business nature with regard to network-related syscalls (connect). While I can't tell you exactly how auditd would impact your production environment, I can share stories of how it impacted mine in hopes it will evoke premature thoughts about yours. I would say the major performance consideration with auditd comes in the form of CPU utilization and memory consumption but it may be hard to determine until you actually see it in effect. My advice here is work with your operations team, explain to them the potential issues auditd could cause and their knowledge of the environment may help you arrive at a near-reality expectation of production-behavior. Configure acceptable auditd log file sizes, consider capping auditd resource utilization (check nice, cpulimit and cgroups), roll out in batches and finally learn and adapt the configuration based on the learning.

Because auditd has so many different record types, monitoring all of them may not feasible. I've found that, aside from being strict with the rules you write, excluding message types can also be a nice way to trim down noise. Here are some message types I think are worth monitoring from a security perspective



Record Type	Meaning
ADD_USER	Triggered when a user-space user account is added.
CRED_ACQ	Triggered when a user acquires user-space credentials.
DAEMON_END	Triggered when a daemon is successfully stopped.
DAEMON_START	Triggered when the auditd daemon is started.
EXECVE	Triggered to record arguments of the execve(2) system call.
LOGIN	Triggered to record relevant login information when a user log in to access the system.
SOCKADDR	Triggered to record a socket address.
SYSCALL	Triggered to record a system call to the kernel.
USER_LOGIN	Triggered when a user logs in.
USER_START	Triggered when a user-space session is started.
USER_AUTH	Triggered when a user-space authentication attempt is detected.

When writing your audit.rules, it makes sense to explicitly specify which record types are not to be recorded. We've found this as a good way to reduce noise from auditd. You can do so using rules like the following

-a never,exclude -F msgtype=ANOM_ACCESS_FS

Each line in the auditd rule file gets evaluated sequentially and while compounding rules are generally a good idea from a performance perspective, unfortunately you cannot do that with the above exclusions. So for each msgtype you'd like to exclude, you'd need to have it as a separate line. Thanks to sqall01 for pointing it out.

```
 -a never,exclude -F msgtype=ANOM_ABEND
 -a never,exclude -F msgtype=ANOM_ACCESS_FS
 -a never,exclude -F msgtype=ANOM_ADD_ACCT
 -a never,exclude -F msgtype=ANOM_AMTU_FAIL
 -a never,exclude -F msgtype=ANOM_CRYPTO_FAIL
 -a never,exclude -F msgtype=ANOM_DEL_ACCT
 -a never,exclude -F msgtype=ANOM_EXEC
 -a never,exclude -F msgtype=ANOM_LOGIN_ACCT
 -a never,exclude -F msgtype=ANOM_LOGIN_FAILURES
 -a never,exclude -F msgtype=ANOM_LOGIN_LOCATION
 -a never,exclude -F msgtype=ANOM_LOGIN_SESSIONS
 -a never,exclude -F msgtype=ANOM_LOGIN_TIME
 -a never,exclude -F msgtype=ANOM_MAX_DAC
 -a never,exclude -F msgtype=ANOM_MAX_MAC
 -a never,exclude -F msgtype=ANOM_MK_EXEC
 -a never,exclude -F msgtype=ANOM_PROMISCUOUS
 -a never,exclude -F msgtype=ANOM_MOD_ACCT
 -a never,exclude -F msgtype=ANOM_RBAC_FAIL
 -a never,exclude -F msgtype=ANOM_RBAC_INTEGRITY_FAIL
 -a never,exclude -F msgtype=ANOM_ROOT_TRANS
```

As for syscalls, the auditd engine intercepts each syscall that a program makes, attempting to match it against system call rules. These rules, when sent to the kernel, the syscall fields are all put into a mask so that one compare can determine if the syscall matches or not. So in that case, combining syscalls in one rule is efficient.

Additionally, you can find the entire list of record types here. The list of main and auxiliary record types can also be found directly in the code.

An event on a Linux system may trigger multiple auditd events. There is one primary event followed by auxiliary events of different record types. These auxiliary events have supporting information for the event. The number of auxiliary records an event may have depends upon the path a syscall takes through the kernel and where auditd is designed to hook into it. At the moment, there's no mapping table between syscall and record types generated that I know of so I started to make one. An example of an auxiliary record would be the record type CWD which you will see many times in the audit.log. CWD records gives you the current working directory from the main event took place. But you don't see it on the list above. Reason being, while it can be useful to know this in some cases, it's also not the most critical piece of information when you're trying to detect bad activity and you can make some good guesses based on the other data you have. We're trying to even out security vs. feasibility and in doing so, you can't have absolutely everything.

. . .

Closing Notes

I believe auditd can indeed be used as a means for detecting threats on Linux endpoints, even in large environments. Throwing in a pre-configured rule set from the internet will give you issues however. Knowing exactly what you want to monitor and eliminating noisy events using methods described in this post, will help you arrive at a configuration suitable for even large, complex environments.

The approach I would take is this — monitor your syscalls of interest (execve, connect) right on top followed by the files/directories of importance and then finally simpler rules to look for monitoring specific binaries like nmap, tshark, etc. if that's important to you. When monitoring files, don't forget to monitor your auditd configuration as well. You don't want an attacker to get away after tinkering with your logging configs (Kudos to Security Shenanigans for this point!)

I plan to write another blog dedicated to tips to better analyze audit logs as well as share sample log snippets of different activity types so you know what they look like in the logs. Not sure when I'll get around to doing this, depends on how useful this one is to begin with. Thanks for making it this far :)

Suggestions/feedback, please drop me a message on Twitter.

[22 October 2022] Yes, part 2 is in the making. Should be out in a few months :)

Link to Part 2.

Linux · Blue Team · Threat Hunting · Cybersecurity · Incident Response

Written by IzyKnows

189 followers · 3 following

Follow

I like to break things. @IzySec

Responses (3)

Write a response

What are your thoughts?