Linux auditd for Threat Detection [Final]

IzyKnows · Follow
8 min read · Mar 21, 2023

Part 1: Linux auditd for Threat Detection [Part 1]

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

This will be the last of the series and IMO, the most interesting one. I highly recommend reading part 1 & 2 before reading this.

The focus of this article will be to describe what behaviors allow for which events to be recorded by auditd. Additionally, you will see where auditd is not capable of recording certain events, despite verbose settings.

By the end of this article, you should have some answers to the following questions

- What is auditd capable of recording?
- What does auditd miss to record?
- How many record types does auditd generate for a given behavior?

Fair disclaimer, this is a content-heavy article. The intended use however, is to serve as reference when investigating logs or creating detection rule sets. A quick read should give you an idea of how to use this article, after which bookmarking it for later use is how I recommend using it.

Let's get started.

Recap & Introduction

Part 1 of this series covered an introduction to auditd, relevant events to record and some tips on scalability.

Part 2 talked about how to investigate auditd logs taking a type of process execution as an example.

If you recall from part 1,

> *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.*

Till date, I still haven't found such a table. We hunt adversaries based on several artifacts, one of them being event logs. We rely on these logs to tell us stories of what happened. Aside from understanding the semantics of a log event and what it's capable of telling us, knowing what log events generate for a given behavior and why is equally important. The idea is if you understand and what auditd is capable of telling you for a given behavior (and why), you can not only better decide what's worth monitoring for your organization but make sense out of logs you're thrown to investigate.

This topic in my opinion, is far more researched in the Windows OS than Linux. You have great articles that explore the chain of logs generated for a given operation in Windows. A project worth mentioning is EVTX-ATTACK-SAMPLES from @SBousseaden which is great when you're looking to answer the question "How would this attack look in the logs?". With auditd however, due to the nature of how the Linux audit subsystem records events, the key in knowing how the logs look for an operation is in understanding

- What syscall(s) will be triggered for the operation — relatively easy to determine based on nature of the operation (file creation, process creation, network connection, etc.)
- What path does that syscall take upon execution vs. where does auditd have hooks to record it? — not so easy to determine

In this article, we look to answer the above questions for some of the common behaviors of interest to us as defenders. We'll see a series of executions of these operations/behaviors as well as what auditd is capable of recording for them.

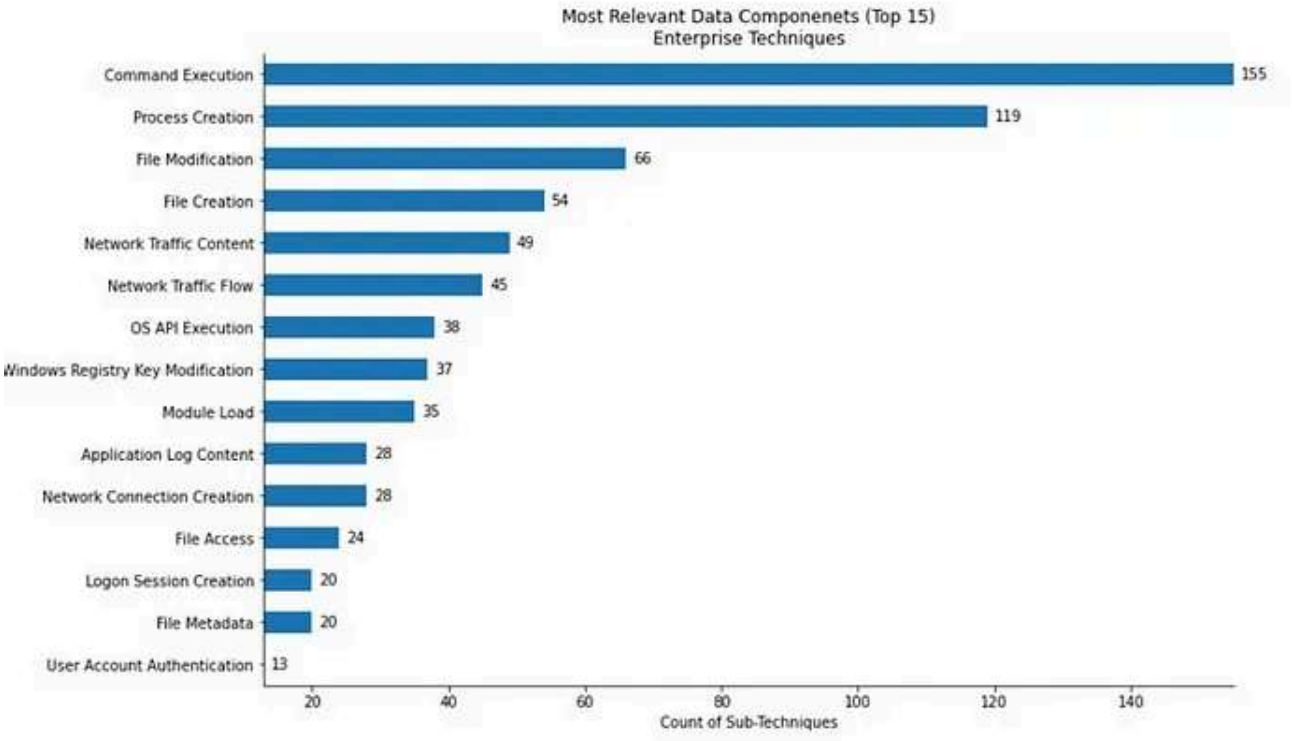
In this article you will find

- A matrix containing a variety of common behaviors along with the corresponding records generated by auditd and
- raw and enriched log samples of each via GitHub

I won't be explaining each entry as part 2 should give you that. Instead, I'll highlight noteworthy points in places I think need it. Please keep in mind, unlike the EVTX project mentioned above, I'm not working at a technique (or procedure) level. I focus on the **data source level** instead.

Data Source to Record Type

Posting the MITRE data sources once again for quick reference



Using the above as my basis, I've extended the simple data source <-> auditd rule table from part 1 to a data source <-> auditd rule <-> simulated behavior <->

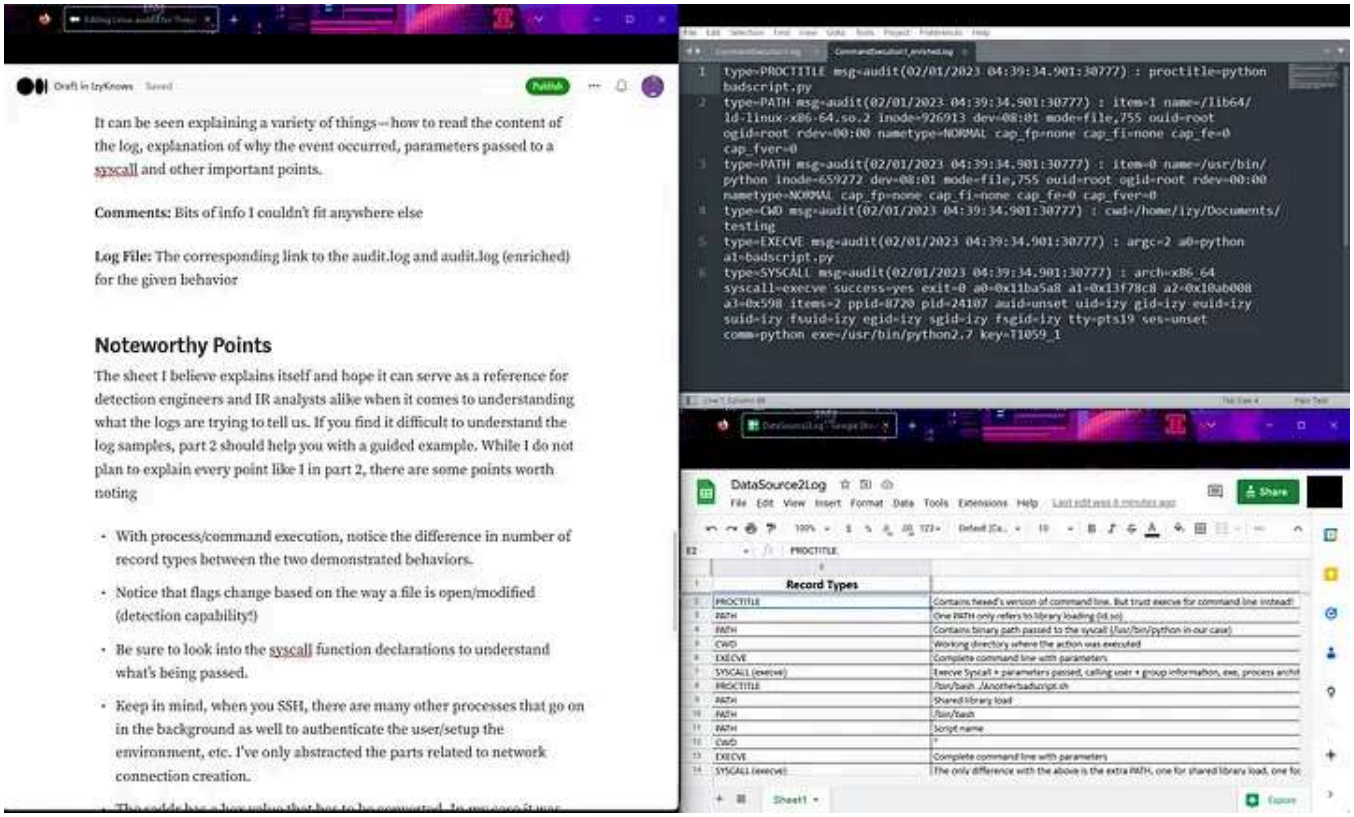
generated record types. I've updated some rules as well.

Link to Sheet: https://docs.google.com/spreadsheets/d/1OPX-RXl_OKhwsqbWUqyGJvREim2K6sAOVUYKMmlxMNM/edit?usp=sharing

Sneak Preview

At this point to understand the following sections better, I recommend displaying these 3 windows, preferably side-by-side

- this blog
- the above Google sheet
- the corresponding log file for each behavior (link in the Google sheet)



Understanding the Mapping Structure

Within the sheet you'll find each of the auditd-applicable data sources from MITRE's chart along with the following columns

Simulated Behavior: Commands executed to be captured by the corresponding data source

Ran as Root?: Depicting whether the behavior was run with root privileges or not. This should not make a difference to the logs unless explicitly mentioned

Auditd Rule: Holds the corresponding auditd rule(s) that were active when the given behavior was executed. Please note that other than the rule(s) depicted in the row, no additional filtering was performed in the audit.rules file

Record Types: A list of all record types as they appeared in the audit.log post execution of the simulated behavior commands

Useful information: Holds noteworthy information about the record type event generated. The explanation here can vary per behavior/record type. It can be seen explaining a variety of things — how to read the content of the log, explanation of why the event occurred, parameters passed to a syscall and other important points. This column will make the most sense when you're looking at it alongside the actual raw log event (from GitHub).

Comments: Bits of info I couldn't fit anywhere else

Log File: The corresponding link to the audit.log and audit.log (enriched) for the given behavior

Noteworthy Points

The sheet I believe explains itself and hope it can serve as a reference for detection engineers and IR analysts alike when it comes to understanding what the logs are trying to tell us. If you find it difficult to understand the log samples, part 2 should help you with a guided example. While I do not plan to explain every point like in part 2, there are some points worth noting

- Man pages of syscalls can help you understand arguments being passed
- With process/command execution, notice the difference in number of record types between the two demonstrated behaviors (UC001/UC002). This may change as you invoke execve() in different ways. The key is in knowing how your execution sequence happens under the hood. In the case of UC001, Python, which is found in the $PATH environment variable (not to be confused with PATH record type), is probably being invoked directly with the filename sent as a parameter. For the bash execution use case (UC002), because we specify a "/" in our command, we explicitly tell our shell where the file to execute is (in '.'), thereby bypassing a PATH check and executing via the current shell. It's not easy to have explanations for everything naturally, and you probably won't, but hopefully these two examples give you some potentially repeatable patterns with other execution methods
- Across the use cases that have to do with file handling operations (UC003-UC006), notice how the flags (`a1`) change based on the way a file is open/modified. The flag combinations can tell you for what intention the file was opened.
- Notice how despite the various methods of file execution, we cannot see the modified content, not even by monitoring execve. Should someone know a way to see modification content with auditd, I'd love to know
- UC003 is an example of how PROCTITLE cannot be trusted for complete command line. It doesn't do well with piped/redirect commands.

- With UC006, using vim causes a whole load of events when monitoring a folder since vim creates a bunch of temporary files as part of it's normal execution
- UC009-UC011 : I could not for the life of me get module loads to log despite executing different, very common instructions. If someone can trigger it, I'd be happy to update the sheet
- Keep in mind, when you SSH (UC012), there are many other processes that go on in the background as well to authenticate the user/setup the environment, etc. I've only abstracted the parts related to network connection creation.
- The saddr (UC012, UC013) has a hex value that has to be converted. In my case it was 0.0.0.0 as I did ssh in from the same host. It's got a strange encoding format, this Splunk query can help with decoding

```
| eval ipaddr = substr(saddr,9,30)
| rex field=ipaddr "(?i)(?<d1>[0-9A-F]{2})(?<d2>[0-9A-F]{2})(?<d3>[0-9A-F]{2})(
| eval ip=tostring(tonumber(d1,16))+"."+tostring(tonumber(d2,16))+"."+tostring(
```

- Additionally, network creation gives you the destination IP + what process and user initiated the connection (very useful!)
- For UC016, 'file metadata' as a behavior could mean a wide variety of things. MITRE's definition—

> *Contextual data about a file, which may include information such as name, the content (ex: signature, headers, or data/media), user/ower, permissions, etc."*

- Auditd isn't a polling solution like OSQuery, so we only get information about a file when a process interacts with it. In this case, I'm focusing on permission changes to the file. Looking through the MITRE techniques associated with this data source, I couldn't find any other use case that auditd would do justice to. We've already covered, in previous data sources, what the logs look like for other relevant file operations (writing, reading, execution, attribute changes). For permission-related changes, I'm choosing to monitor specific commands like chown/chmod. Note that it's worth while also monitoring syscalls related to

changing file attributes (fchmod, fchown, lsetxattr, etc.). The audit.rule references in Part 1 have a verbose list of the syscalls.

- UC017/018: I recommend in general using the secure.log or auth.log for monitoring authentication activities. Auditd can give it to you too, I've covered that too for comparison
- UC017: USER_AUTH seems to be more reliable than USER_LOGIN. I noticed some missing values in USER_LOGIN.
- Be careful, when you monitor failed ssh logins using audit.log, depending on what rules you have, you'll get a bunch of other events, with success results. For example, the hosts.allow, hosts.deny files are queried, socket connection (syscall connect) all show up as successful events. We're focused on whether the actual authentication event failed/succeeded, which is what you'll see in the attached log

Closing Notes

That is a lot of information. While verbose, I do hope it was comprehend-able nevertheless. Testing the above was time-staking but interesting nevertheless as many of the results surprised me as well. I tried to find a more reliable way of determining auxiliary events by looking through the source code but it was so scattered that I found this 'bruteforcing' as a more efficient option.

Like previously mentioned, the intention of this content is to serve as reference to IR analysts/detection engineers/threat hunters, anyone trying to understand auditd logs and trying to model them for finding threats. Should you like to add to the document, please reach out to me, I'd be happy to include it.

With that, we're at the end of this series and I must find a new muse to write about (Windows internals anyone?). I do hope you found the series useful and feedback is welcome, I'm not difficult to find :)

Threat Detection | Threat Hunting | Linux | Blue Team