



# Linux auditd for Threat Detection [Part 2]



IzyKnows

Follow

12 min read · Feb 10, 2023



104



2



Part 1: [Linux auditd for Threat Detection \[Part 1\]](#)

Part 3: [Linux auditd for Threat Detection \[Final\]](#)

Early 2022 I wrote part 1 of this “series” which received such positive response that I decided to do part 2.

Enjoy.

## Recap & Introduction

In part 1, we had an introduction to auditd and the basics of rule writing. I recommended some reading material and settings I've had success with in the past. We covered MITRE's most relevant data sources and which auditd rules would help record them. Lastly, we covered some noise reduction tips and how you can get auditd to work in large, diverse environments.

The most valuable takeaway from part 1 in my opinion was the rules <-> data source mapping and recommended exclusions. In this part, the goal is

- to deep dive into an auditd log event
- to look at what fields are particularly interesting to us from an adversary hunting perspective and
- some Splunk tips on log investigation

This post will be a light one and is really meant to be a preamble for part 3 which will be posted shortly after this one. The content is ready and was initially planned to be a part of this post until it became too long. Part 3 will be the most interesting in my opinion so stay tuned :) </end of advertising>

Throughout this article, I'll be sharing my experience along with fabricated scenarios within a monitored environment. It's a simple VM running Linux ubuntu 4.15.0-142-generic 64bit.

With that, let's get started.

. . .

## **Security in the Logs**

I'd like to pick a behavior, execute it and then go into the logs to talk about what different fields mean and which could potentially hold useful information for us.

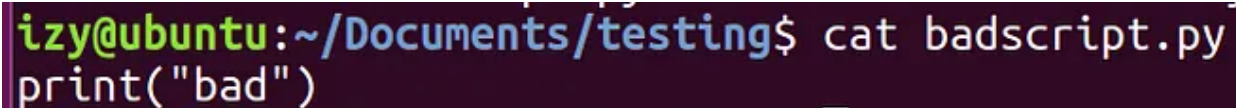
I will focus on simple execution of a script using Python. The exact process that is executed is not as important as understanding how the system logs the technique itself. Much of what you'll see below is transferable knowledge to other behaviors like file creation/modification/network events, etc. In this part, I will highlight only 1 technique (process/command execution) and explain the fields. In Part 3, I will show you tests I ran for several other behaviors.

## Command Execution

When a command is executed, `execve` is the syscall that executes the program. This is a critical one from an adversary detection perspective. Take execution of the following command

```
python badscript.py
```

The content of this script is irrelevant, it's just a print command.

A terminal window with a dark purple background. The prompt is 'izy@ubuntu:~/Documents/testing\$'. The command 'cat badscript.py' has been entered, and the output 'print("bad")' is displayed on the next line.

```
izy@ubuntu:~/Documents/testing$ cat badscript.py  
print("bad")
```

Before execution, we setup auditd with two rules to monitor `execve` calls. Notice how there's none of the filtering applied from we talked about in part 1.

```
-a always,exit -F arch=b64 -S execve
-a always,exit -F arch=b32 -S execve
```

Upon execution, we see within the logs we have 6 different entries for this activity

```
root@ubuntu:~# ausearch -a 30777
****
time->Wed Feb  1 04:39:34 2023
type=PROCTITLE msg=audit(1675255174.901:30777): proctitle=707974686F6E006261647363726970742E7079
type=PATH msg=audit(1675255174.901:30777): item=1 name="/lib64/ld-linux-x86-64.so.2" inode=926913 dev=08:01 mode=01007
55 ouid=0 ogid=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PATH msg=audit(1675255174.901:30777): item=0 name="/usr/bin/python" inode=659272 dev=08:01 mode=0100755 ouid=0 og
id=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=CWD msg=audit(1675255174.901:30777): cwd="/home/izy/Documents/testing"
type=EXECVE msg=audit(1675255174.901:30777): argc=2 a0="python" a1="badscript.py"
type=SYSCALL msg=audit(1675255174.901:30777): arch=c000003e syscall=59 success=yes exit=0 a0=11ba5a8 a1=13f78c8 a2=10a
b008 a3=598 items=2 ppid=8720 pid=24107 auid=4294967295 uid=1000 gid=1000 euid=1000 suid=1000 fsuid=1000 egid=1000 sgi
d=1000 fsgid=1000 tty=pts19 ses=4294967295 comm="python" exe="/usr/bin/python2.7" key="T1059_1"
```

The behavior is one, but the record types are multiple. We see that in the case of command execution, the path taken through the kernel triggers the record types PROCTITLE, PATH, PATH, CWD, EXECVE and SYSCALL. Each of them with different fields, some useful to us, some not. Knowing this is useful for hunting threats.

Let's narrow into the events

```
type=PROCTITLE msg=audit(1675255174.901:30777): proctitle=707974686F6E0062616473
```

**type=PROCTITLE:** The record type is said to give the complete command line that triggered the audit event. It is however hex-encoded. If you use the ausearch utility that comes with auditd with -i (for interpret), it will decode this for you. Splunk users can use something like this

```
| eval cmd = urldecode(replace(encoded_data,"([0-9A-F]{2})", "%\1"))
```

Before we go into the event, I'd like to highlight that while PROCTITLE is said to give complete command line, **it is not always reliable**. We'll cover more about where it cannot be relied on in part 3. There is a more reliable record type that we'll explore below.

The first field I want to cover is the 'msg' field. Every event within the audit.log should have a msg field.

```
audit(1659896884.775:15237)
```

The msg field follows the format `audit(time_stamp:ID)` . The time\_stamp is in epoch format. The ID however is interesting from an investigative perspective because all audit events that are recorded from one application's syscall have the same audit event ID. A second syscall made by the same application will have a different event ID. This way they are unique and you can use them for finding all activity related to that syscall. However, if you have custom tagging on auditd rules (-k), I recommend using ID + the tag to look for events as the ID field is not “truly” unique, i.e, I’ve seen them overlap over longer periods of time.

You can use the following Splunk query to split the two with the following SPL (the auditd TA won't do it for you).

```
| rex field=msg "audit\\(\\d+\\.\\d+:(?<id>[^\\)]+)"
| rex field=msg "audit\\((?<epoch>[^:]+)"
| eval log_time = strftime(epoch,"%Y-%m-%dT%H:%M:%S.%Q")
```

Adding that to a transform/macro for auditd events within your SIEM may be useful.

Moving on,

```
type=PATH msg=audit(1675255174.901:30777): item=1 name="/lib64/ld-linux-x86-64.s  
type=PATH msg=audit(1675255174.901:30777): item=0 name="/usr/bin/python" inode=6
```

**tpye=PATH:** This record type is said to contain the **path that is passed as a parameter to the syscall** (execve in this case).

Let's go down a quick rabbit hole here.

---

Get IzyKnows's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email



Subscribe



You'll often see PATH contain ld.so (i.e., /lib64/ld-linux-x86-64.so.2) like above. This often depends on the binary you execute. ld.so is an argument often passed to execve during execution of ELF files that require dynamically linked libraries. This means the program running requires shared libraries to be located and linked at run-time. Ld.so is a runtime linker and you should see which ELF binaries need dynamically linked libraries by looking at their ELF program header. Here's a nice read if you're interested in ELF binaries: <https://lwn.net/Articles/631631/>

Take /bin/chmod (although it's the same for /usr/bin/python). With readelf, you can find out if ld.so will be loaded at runtime like so

```

root@ubuntu:~# readelf -l /bin/chmod

Elf file type is EXEC (Executable file)
Entry point 0x402680
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz              MemSiz              Flags  Align
PHDR             0x0000000000000040 0x0000000000040040 0x0000000000040040
                 0x00000000000001f8 0x00000000000001f8  R E    8
INTERP           0x0000000000000238 0x00000000000400238 0x00000000000400238
                 0x000000000000001c 0x000000000000001c  R     1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD             0x0000000000000000 0x0000000000040000 0x0000000000040000
                 0x0000000000000c9bc 0x000000000000c9bc  R E    200000
LOAD             0x0000000000000ce10 0x0000000000060ce10 0x0000000000060ce10
                 0x00000000000004a4 0x0000000000000698  RW    200000
DYNAMIC          0x0000000000000ce28 0x0000000000060ce28 0x0000000000060ce28
                 0x00000000000001d0 0x00000000000001d0  RW     8
NOTE            0x0000000000000254 0x00000000000400254 0x00000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
GNU_EH_FRAME     0x0000000000000ad14 0x0000000000040ad14 0x0000000000040ad14
                 0x00000000000000474 0x00000000000000474  R      4
GNU_STACK        0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW    10
GNU_RELRO        0x0000000000000ce10 0x0000000000060ce10 0x0000000000060ce10
                 0x00000000000001f0 0x00000000000001f0  R      1

```

So, it is normal to see ld.so being passed to execve, depending on the ELF binary you're executing. Keep in mind most binaries today will dynamically load shared libraries so you'll see it more often than not.

Coming back, another useful bit of info for us in PATH is the mode because it contains permissions of the file it refers to. In this case, you will notice mode=0100755 which translates to 'file,755. 755' (rwxr-xr-x) being the permission of the object, in this case, lib64/ld-linux-x86-64.so.2 and /usr/bin/python. This could be important for PATH may contain a suspicious file that's attempting execution.

```
type=CWD msg=audit(1675255174.901:30777): cwd="/home/izy/Documents/testing"
```

**type=CWD:** This record type holds the **current working directory** (hence, CWD) from where the process which invoked the syscall (execve in our case) was executed. The process which invoked the syscall in our case is /usr/bin/python2.7 which was executed in the above mentioned directory. It's worth noting how PATH and CWD record different things and hence is capable of telling us different stories. In our example we see the two hold similar-ish values but this may not always be the case and we'll see some examples of that in part 3.

```
type=EXECVE msg=audit(1675255174.901:30777): argc=2 a0="python" a1="badscript.py"
```

**type=EXECVE:** This is the most interesting one because it contains the command line and one that **should be trusted over PROCTITLE**. I should however mention that there are *very few* cases where execve does not include the entire command line and PROCTITLE can prove to be more accurate. We'll look at a few of those examples in part 3 but for the most part, execve can be trusted for entire command line. Each argument in the command line will be a separate field starting from 'a0' to 'a\*'. 'a0' will always be the binary performing the execution, in this case, python. Every 'a' field thereafter will be command line parameters passed to a0.

If you're consuming these logs in a SIEM, the annoying part is you need to build up the command line yourself from the various arguments (the Splunk TA won't do this either). With Splunk, you can use a foreach statement to iterate through all a\* variables in an event and build the command line. The exact SPL is left as reader's exercise :)

```
type=SYSCALL msg=audit(1675255174.901:30777): arch=c000003e syscall=59 success=y
```

**type=SYSCALL:** This is a rich one too, with much meta-info to understand the event better. This event essentially tells us that the `execve` syscall was invoked successfully via 64-bit python process, the process and parent process ID and that the user with uid 1000 (izy) has performed this action under it's own user context. That's a lot of rich info, but how does this event tell us that?

**pid, ppid, comm, exe:** fairly straightforward fields, I won't go into them

**auid:** This refers to the audit user ID. This ID is assigned to a user upon login and is inherited by every process even when the user's identity changes. This field is useful to trace events when a user switches user contexts but you still want to know what they did. For example, if a user logs into a machine and thereafter escalates privileges to the root user (`sudo su`), the auid for the events will stay the same before and after escalation. It's worth noting that you may notice this value as '4294967295' in some cases. This is discussed in part 1 and is equivalent to 'unset'.

**success=yes:** The syscall returned a successful state, i.e. the process successfully executed

**arch=c000003e :** Hex representation for x86\_64 (64bit)

**syscall=59:** This is the ID of the syscall that was invoked. 59 is for “execve”. I’ve found [this table](#) helpful for such resolution. Additionally, you can use the `ausyscall --dump` utility to see the syscall. Splunkers could possibly add the above table to a lookup list for search-time enrichment.

All the \*id fields tell us about the user, which group they’re in, whether it was executed by the current session user or under the context of another user and also whether the suid bit was set for the exe in question. These are POSIX terms which I won’t go into explaining but it’s worth understanding them. If you want to understand them at depth, I’d look at the [man pages](#).

As a quick reference for the different audit field events, I can recommend this nice table: <https://access.redhat.com/articles/4409591>

The a\* fields within this record type are the invocation of the execve syscall itself (in hex), which has the syntax

```
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

It's worth noting the values of these arguments are hex-encoded although I did not dig deep enough to be able to meaningfully interpret these fields. If someone knows more about how one can easily interpret command line arguments passed to the syscall via this audit log event, I'd be happy to add it in.

Another useful trick, if you have endpoint access, and want to look at a wonderfully parsed version of an audit event, you can use the `ausearch -i` to do a lot of the parsing for you. Look at the difference `-i` makes. Everything (almost) we wish our Splunk TA would do

```
root@ubuntu:/home/izy/Documents/testing# ausearch -a 15237
****
time->Sun Aug  7 11:28:04 2022
type=CWD msg=audit(1659896884.775:15237): cwd="/home/izy/Documents/testing"
type=EXECVE msg=audit(1659896884.775:15237): argc=2 a0="python" a1="badscript.py"
type=SYSCALL msg=audit(1659896884.775:15237): arch=c000003e syscall=59 success=yes exit=0 a0=c38d08 a1=c38dc8 a2=
c1e008 a3=598 items=2 ppid=21182 pid=21195 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=
0 tty=pts4 ses=4294967295 comm="python" exe="/usr/bin/python2.7" key="T1548.001_1"
root@ubuntu:/home/izy/Documents/testing# ausearch -i -a 15237
****
type=CWD msg=audit(08/07/2022 11:28:04.775:15237) : cwd=/home/izy/Documents/testing
type=EXECVE msg=audit(08/07/2022 11:28:04.775:15237) : argc=2 a0=python a1=badscript.py
type=SYSCALL msg=audit(08/07/2022 11:28:04.775:15237) : arch=x86_64 syscall=execve success=yes exit=0 a0=0xc38d08
a1=0xc38dc8 a2=0xc1e008 a3=0x598 items=2 ppid=21182 pid=21195 auid=unset uid=root gid=root euid=root suid=root f
suid=root egid=root sgid=root fsgid=root tty=pts4 ses=unset comm=python exe=/usr/bin/python2.7 key=T1548.001_1
```

While on the topic of other ways to parse auditd logs, here's an interesting visualization tool from Steve Grubb: <https://github.com/stevegrubb/audit-explorer/>

. . .

### **Side-note: Scaling Security Agents**

I forgot to mention this point in the last part hence it's here as a quick side-note. I'll move this to part 1 thereafter since it's better suited there.

My focus with this article series is security at scale. So while there may be “better” ways of doing something, our definitions of “better” may vary. I'm looking into reasonable security that can scale.

We're security guys, not operations experts. In an ideal world, you'd partner with a central operation teams in your environment to collaboratively roll out agents in a responsible way. That was hardly the case in my scenario and had me fall into several operational bad practices during deployment that I hope to help you avoid.



I've seen auditd (and other agents) being used at large scales (Over 100,000 servers) and working fine. I won't lie, this doesn't come easy. Auditd, like every other security tool, needs care and constant tuning, there is no one configuration that will work in all environments. An added consideration is that with auditd, you're responsible for debugging issues and can't point to a vendor to fix. On the flip side, you have superior control of how the tech works, which in my opinion is better than a blackbox EDR vendor "fixing" the issue. Both approaches have pros and cons, you need to decide which works better for your goals and budget. With auditd, what's key is to know your environment and, hopefully with this series, know how auditd can give you the most.

When deploying auditd (or any security agent), It's often a good idea to deploy in phases/rings. If you build an auditd rule set that works perfectly in your lab and deploy it across a productive environment, prepare to fail almost instantly. There is no magic configuration that will work everywhere, make sure you set your stakeholder expectations right about that fact. You want to divide your environment into testing "rings" consisting of endpoints of preferably similar function. The level of granularity you want to go to define rings of "similar function" is based on the time/effort you can invest, but the underlying logic is the same. For example, your user endpoint devices in one ring, database servers in another, web servers in another, and

so on. You could further increase granularity with say, a ring for developer endpoints since you know they often run all kinds of binaries, compared to those not working so close to technology.

The logic is: endpoints of similar nature will probably portray *similar* kind of processes/activity and are your best shot at testing your rule set against diverse, but manageable representations of your environments. Let me give you an example. In an environment I worked in, I realized that while my auditd rule set worked quite well overall, there were a particular bunch of systems that were just overflowing with logs and was causing issues, both on the endpoint (performance) and SIEM side (space). On digging deeper, we found it was a set of web servers that were constantly opening and closing network connections. This was their normal behavior. As a result, the number of events being generated by monitoring the ‘connect’ syscall were through the roof. The good thing is that if you find the “offending” log event, you can use -F to tune out such events within the .rules file. Keep in mind the field types you’re allowed to use with -F. Remember, rules are matched in a top to down order, so ensure you put your highest confidence exclusions on top. Also know that you can have multiple .rules files within /etc/audit/rules.d/\*.rules. You can use this to your advantage and store environment specific exclusions in a separate rule file for better maintainability. This is discussed more in part 1.

. . .

## Conclusion

Record types like PROCTITLE, PATH, CWD, SYSCALL — they are common and you'll see them recurring throughout audit.log, regardless of the activity you're executing. So understanding what these record types tell us is the basis for understanding how to read auditd logs and transferable to other behaviors too. This helps with the decision of whether a record type is useful for you or not.

With the above, you should have the information you need to understand/hunt through other audit events as well. The intention of part 2 was to pick a particular behavior (process execution) and talk about what auditd is capable of telling you about it with no filtering.

With this knowledge, in part 3, we'll go into what the logs look like for simulations that cover all applicable MITRE data sources. I will not go into each in detail the way I did here but rather share the results and noteworthy observations where applicable. Additionally, I'll share raw & enriched log samples for each.

Lastly, a special thanks to those who reviewed this article and gave valuable feedback. It's a fairly long list but you know who you are :)

## Reading Material

Lastly, here are some related readings I enjoyed.

## 7.6. Understanding Audit Log Files Red Hat Enterprise Linux 7 | Red Hat Customer Portal

By default, the Audit system stores log entries in the `/var/log/audit/audit.log` file; if log rotation is enabled...

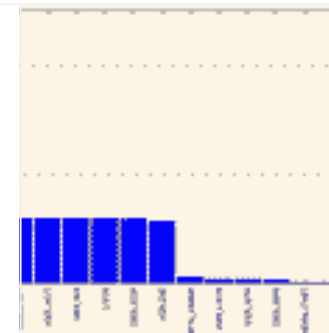
access.redhat.com



## Understanding Linux Audit | SLES 12 SP4

On this page Applies to SUSE Linux Enterprise Server 12 SP4 The Linux audit framework as shipped with this version of...

documentation.suse.com



## Building a SIEM: centralized logging of all Linux commands with ELK + auditd

Recently, working with the SOC department, we had to enable command logging for more than 10k instances. We also needed...

[illegible]