# Linux IR - Creating evidence of execution in Linux

Taz Wake | Halkyn Consulting Ltd | Originally published 2024-07-08

If you come from a Windows DFIR background, you will be very used to the wealth of data we have providing "evidence of execution." We routinely check prefetch, shimcache, amcache, event ID 4688 and so on, to determine if a particular file or process was executed.

If we do incident response on a Linux platform we don't really have that luxury. In fact, most out-of-the-box distro builds provide almost no way to prove process execution. This can generate significant issues during an investigation.

This is even true when a default auditd configuration is in use.

```
root@asgard:~# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.4 LTS
Release:        22.04
Codename:       jammy

root@asgard:~# systemctl status auditd | grep active
     Active: active (running) since Wed 2024-07-03 14:13:50 GMT; 16min ago

root@asgard:~# /root/example.sh

root@asgard:~# grep 'example\.sh' /var/log/* -r
root@asgard:~#
root@asgard:~#
```

Demonstration of a default Ubuntu 22.04 system with auditd installed and running, using the default settings. Here I was able to run the script "example.sh" without any evidence being stored in the log or journal files.

In very general terms, the best way we can get evidence of execution in any Linux distro today is by using AuditD. But, as the example above shows, the default configuration just isn't going to meet our needs as incident responders.

In RHEL family distros, AuditD **does** come installed by default but it also has a problem with the rules being insufficient for our needs.

This is the default setting for the audit.rules:

```
```

As a quick summary, what this means is:

- **-D**: Delete all existing audit rules. This ensures that the audit daemon starts with a clean slate before applying new rules specified in the audit.rules file.

- **-b 8192**: Set the size of the audit buffer to be able to store 8192 records before it starts to discard events. This is a good way to manage situations where there is a burst of activity and you don't want to lose data before it gets written to disk.
- **-f 1**: This sets the failure mode for the system. The 1 here means the kernel should "panic" (halt) if it is unable to write audit events.
- **--backlog_wait_time 60000**: This sets the amount of time (in milliseconds) that the audit daemon will wait for a backlog to clear before dropping audit events.

These are all really good things to have in place, but the default settings don't actually configure auditd to log activity. It will run in the background, it will generate *some* event data but it isn't really doing what we want it to.

Now, to be clear, the audit daemon will log some commands but we don't have control over it and it will generally be commands related to system activity - we could get a service starting but not an attacker running their C2 implant or exfiltrating data.

Going back to our system where example.sh has run, you can see that the audit log does contain data, including process execution - but none of it relates to that script.

```
root@asgard:~# aureport

Summary Report
======================
Range of time in logs: 20/02/23 12:56:50.306 - 03/07/24 14:42:12.463
Selected time for report: 20/02/23 12:56:50 - 03/07/24 14:42:12.463
Number of changes in configuration: 2396
Number of changes to accounts, groups, or roles: 28
Number of logins: 2
Number of failed logins: 1
Number of authentications: 191
Number of failed authentications: 20
Number of users: 4
Number of terminals: 34
Number of host names: 7
Number of executables: 42
Number of commands: 22
Number of files: 0
Number of AVC's: 0
Number of MAC events: 0
Number of failed syscalls: 0
Number of anomaly events: 19
Number of responses to anomaly events: 0
Number of crypto events: 0
Number of integrity events: 0
Number of virt events: 0
Number of keys: 0
Number of process IDs: 1596
Number of events: 24933
```

Aureport output - in the 18 months of data, there are only 22 commands logged.

Clearly, we need to improve this - and it is a critical part of the preparation phase of your IR cycle that you check to see how your environments are audited, If you don't have good auditing turned on, take action now!

One of the best resources for learning how to get the most out of auditd is a series of Medium posts by Rohit N - you can find these at https://for577.com/auditdguide. I can't recommend these three articles strongly enough.

## Creating evidence of execution

One of the ways, and possibly the simplest, that we can use to create an evidence of execution artifact in Linux is to ensure the audit daemon is configured to log the execve() system call. This is the system call used to execute programs and works by replacing the current process image with a new one specified by the given file.

When the auditd service is configured to monitor execve system calls, it generates log entries every time a process executes a new program. These log entries contain detailed information about the executed command, its arguments, environment variables, and other relevant metadata.

This is definitely what we want to see. It is not perfect, but it really is good enough. This **will** miss things which are "built-in" shell commands (for example, echo or cd). However, here we are really concerned with evidence that a program, script etc ran. This **will** be captured.

### Setting up the auditing

This is the easy part, add the following lines to your audit.rules file (normally at /etc/audit/rules.d/audit.rules)

```

```

This will ensure auditing is enabled for any execve events with 32 or 64-bit architectures.

Next, ensure the rules are loaded and/or restart the audit daemon.

```

```

or

```

```

You can validate the in-use rules with

```

```

### Where is the data?

When you have auditd running and well-configured rules (you probably need more lines than just the two above), the data is logged to two locations in most Linux distros.

1. The traditional audit log is stored in /var/log/audit/audit.log by default, although this can be changed by administrators.

2. A copy of the event data is also sent to the Systemd Journal and stored in a binary file at /var/log/journal/machine-id/*.journal. On some more recent versions of Linux distros, the traditional logging has been disabled and it might be that the journal is the only location for the data.
3. You can manually configure your system to send this data to a log centralisation tool or SIEM and this is **strongly** recommended.

### What does it look like?

This is an example audit log entry, recorded when the user ran "nano evil.sh".

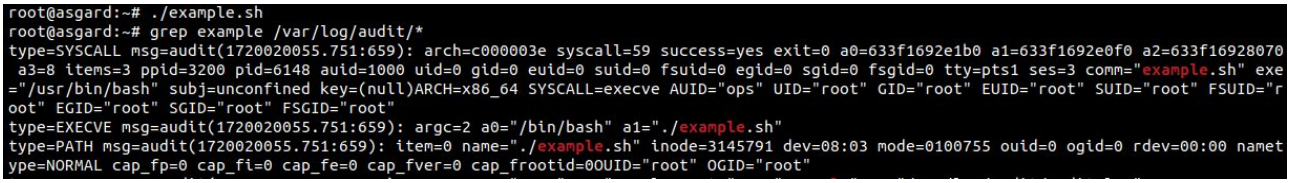
This breaks down as:

- **type=EXECVE** - this indicates the type of log entry.
- **msg=audit(1720021127.311:698):** - This field includes a timestamp (1720021127.311) in the Unix epoch and then an event identifier code. The event identifier can be used to track multiple log entries for a single event.
- **argc=2** - this is the number of arguments being passed to the command.
- **a0="nano"** - this is the first argument, often the command itself. In this example, it is the nano editor.
- **a1="evil.sh"** - this is the second argument and, in this example, it is the argument passed to the first argument.

### Does this work?

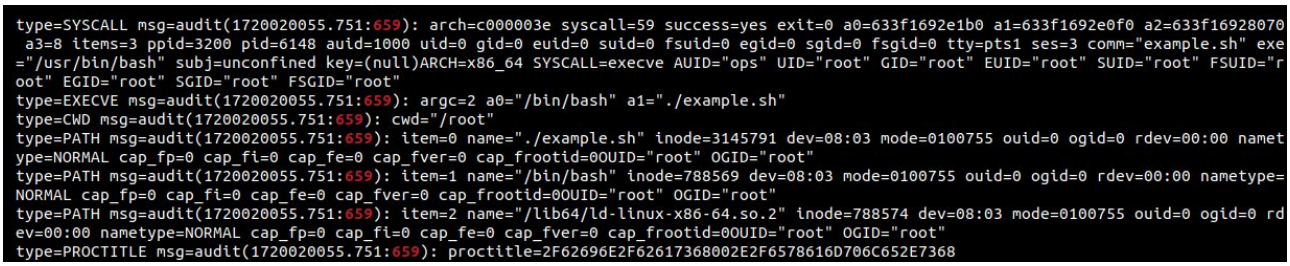
Short answer? Yes. Yes, it does.

With this auditing enabled, we get significantly enhanced visibility into process creation and have very strong evidence of execution.




Updating the audit rules captures the script execution now.

It is also interesting to note, that turning on this auditing does generate more entries than just the single EXECVE line. In the example above, we can see that the new logging now captures a SYSCALL entry (recording the execve() syscall), the EXECVE entry related to the script and also a PATH entry logging the path provided during the command execution. We can even use the event identifier to find other entries related to this event - including CWD and PROCTITLE records.




Searching for all events with the same event identifier

### What are the exceptions?

While the execve syscall will catch almost everything that happens, as mentioned before, it will miss shell built-in commands.

These are commands and instructions that are part of a specific shell, rather than calling out to binaries on the filesystem. For example, if you run cd in a bash shell, this is carried out by the shell (/bin/bash) rather than calling out to a binary like /bin/cd. This can be confusing because there are some built-in commands which are the same as binaries on the filesystem. The important thing to remember here is that the shell built-in commands have higher precedence in the "path", and will always execute before a binary on the filesystem.

If you absolutely want to use the filesystem version, you would need to type the full path to the command.

For example:

```
```

Will use the shell built-in version.

```
```

Will use the file system version, and trigger an EXECVE auditing event.

It is possible to get a list of built-in commands for different shells with one of the following commands:

- (bash): Shows a man page-like listing of built-in commands.
- (bash): Lists all built-in commands.
- (zsh): Lists all built-in commands.
- (zsh): Lists built-in commands and their definitions.
- (ksh): Lists built-in commands along with their paths if they are also external commands.

## Conclusion

The default settings for most (if not all) Linux distros is not great at supporting incident response and there generally isn't a way to prove evidence of execution unless you take measures to generate this data.

However, generating it is pretty easy - you need to ensure the audit daemon is installed, ensure it is running and add a couple of lines to the audit.rules file.

When you do this, you get an incredible enhancement with regards to the visibility of process creation events and it will be much easier if you need to investigate an intrusion.

## About Halkyn Consulting

Halkyn Consulting Ltd is a UK-based security and risk management consultancy providing 24/7 incident response, digital forensics, threat hunting, virtual security leadership (vCISO) and security compliance support to organisations worldwide. We deliver clear, practical, vendor-independent advice based on real-world threats — no hype, no upsell.

More free security resources are available at www.halkynconsulting.co.uk/security-resources.

**Web:** www.halkynconsulting.co.uk | **Email:** info@halkynconsulting.co.uk | **Tel:** +44 1244 940858