



## **Technical Operations and User Manual**

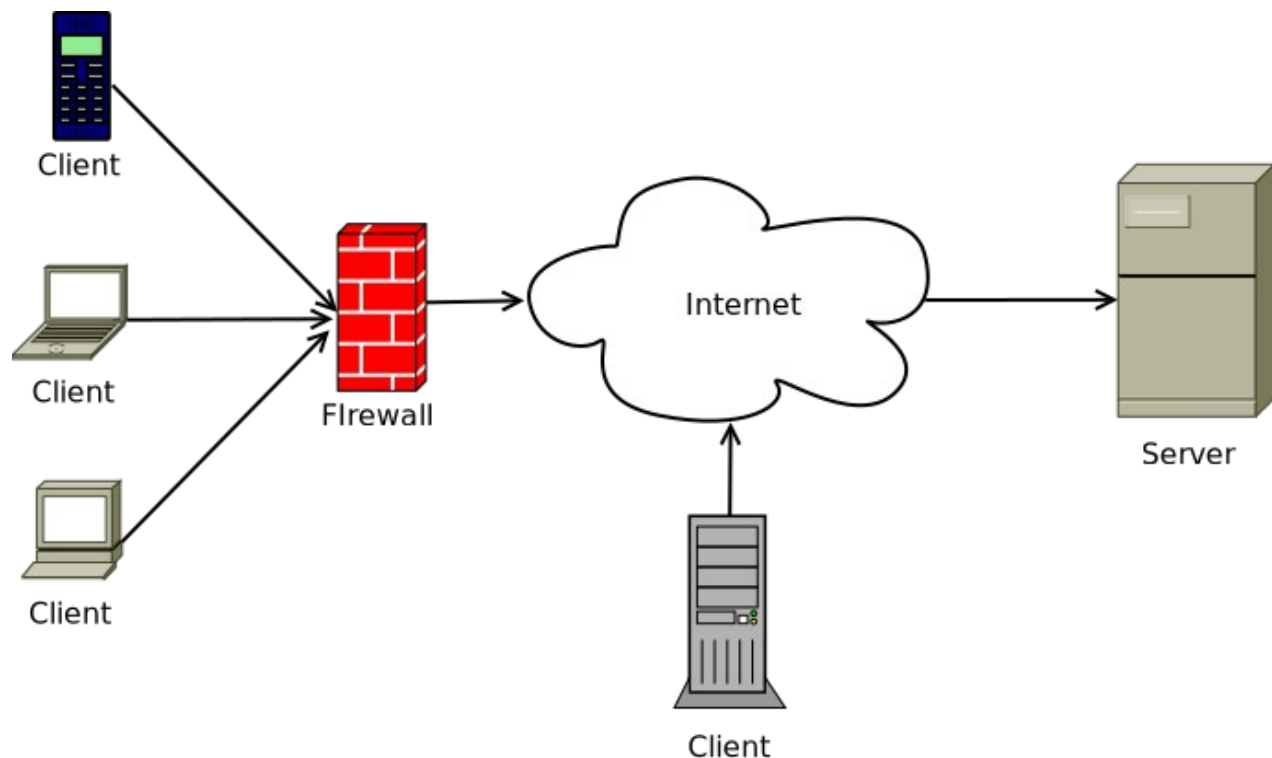


**Version 1.2.3: April 2023**

## A)Introduction and scope

The Penguin OS Flight Recorder collects process execution, file access and network/socket endpoint data from the Linux Operating System. Like an aircraft flight recorded (or black box), its main purpose is to reliably reproduce/replay OS level events that concern process execution, file access and network endpoint creation from each of the monitored Linux clients. IT experts (security analysts, system administrators, devops engineers can then use the collected information to:

- Examine/interrelate in great detail processes, file access and network endpoint events in a monitored system.
- Detect misbehaving computer accounts, applications.
- Issues with DevOps frameworks at Operating System level
- Conduct post-mortem evidence after security compromises with the evidence stowed away from the monitored system.
- Conduct incident threat response exercises and see their effect on Linux systems.
- Comply to security/audit requirements.
- Study the collected network endpoint and process execution traffic over time to obtain OSINT related data (honeypots).



**Figure 1: The POFR client server architecture**

There are other open source projects that perform similar but not identical functionality to POFR. Similar alternatives are:

- the [osquery project](#) which can provide a detailed picture of the state of infrastructures, including some of the information mentioned in the above bullet points.
- The [Microsoft Windows System Monitor \(sysmon\) package](#) which monitors and logs system activity to the Windows event log. The comparison should include its [linux version](#).

In direct contrast to osquery, POFR is better suited to collect information in a time series fashion, accumulating data over a period of time, using less computational overhead and complexity. In comparison to sysmon, POFR does not rely on device drivers (windows version), complex [eBPF](#) mechanisms (linux version). POFR stores instead carefully curated procfs snapshots into a relational database layer, always away from the monitored host.

POFR uses a client/server architecture (Figure 1). Clients are the systems to monitor and place the data on the server. The server parses, stores and acts as the gateway to analyze the data, by using an RDBMS layer. POFR was designed with simplicity and reliability in mind, following the design principles listed below:

- **Minimal installation and network footprint requirements:**
  - No usage of complex kernel modules that can impact the operational reliability of client systems. No interference/reliance with Linux distribution specific software modules is required on the client systems. Everything is provided by POFR repository.
  - No software agents exposing open network ports to obtain the data should run on client systems. Clients push all the data to the server using SHA message digests for data integrity in an encrypted manner by using the SSH protocol. Port 22 can traverse easily firewall/NAT schemes and the system should easily run on TCP/IP networks.
- **Minimal computational overhead on the client systems:** Clients only collect, compress and push data via SSH. All processing correlation is offloaded to the server.
- **Data correlation/analysis is build around the relational layer:** While there are more modern time series and data ingestion database mechanisms, POFR uses an SQL schema to structure and correlate data. The schema is simple enough (no complex index or foreign key dependencies) so that the data can be easily ingested and correlated. This means that the primary means of analysis/query is

the relational layer. However, this should not exclude ways to further forward the collected/correlated info to noSQL databases or other logging/visualization/search frameworks (Elasticsearch, Kibana). POFR provides only a minimal structure analysis approach on the relational layer (do one thing and do it well). Section F4 discusses the relational schema and the way ingested information is organized by POFR.

## **B)Compatibility**

### **B1) POFR clients**

POFR clients have been tested with the following Linux distributions:

- CentOS 7
- RHEL 7,8 and 9
- AlmaLinux/Rocky Linux 8 and 9
- Fedora: 35/36/37
- Ubuntu:18.04 LTS/20.04 LTS/22.04.1 LTS

### **B2) POFR servers**

For a POFR server, we recommend a Fedora 35/36 or a RHEL9 Linux Distribution.

## **C)Dependencies and preparation requirements**

A compatible Linux distribution (see section B ‘Compatibility’). Everything needed by the client and server components is provided, including an up-to-date PERL distribution with all the modules installed. No need to install system wide PERL modules or many packages for the software to function properly. However, you need to do some planning:

- The POFR clients can reach port 22 (SSH) of the POFR server (directly or via NAT). No special hardware requirements, most of the computational resource intensive parts are performed on the POFR server (see below). Even on busy systems, no more than a single core and 400 Megs of RAM are required to run the client software.
- The IP address, FQDN and SSH keys of the POFR server need to remain the same throughout the monitoring session.
- For the POFR server, one needs to ensure adequate hardware requirements: POFR is resource hungry on the server, so the more resources you have, the better. A 32

core, 24 Gb RAM and a few Tb of disk space on /home and /var filesystems should be sufficient to monitor 8-10 busy devices. You need a separate server instance to simultaneously monitor more devices at an adequate speed, so you need to partition the load to more servers/VM instances. For a single server instance use the following figures as a resource allocation guide:

- Disk space: 180-250 Mbytes per hour of continuous monitoring
- RAM: 3-4 Gigs per client
- (v)CPU cores: 4-8 cores per client

We strongly recommend to run both client and software with **SELinux enabled**. The software has been engineered to function with or without SELinux enabled, but you should certainly not encounter problems if SELinux is enabled.

## **D)POFR Client Installation**

To install the POFR client, follow the steps outlined below as the **root user**.

First ensure that your system has the legacy support library for NIS. On the Fedora/RHEL/Centos ecosystem, you could install those with:

**yum -y install libnsl\*** (RHEL7/CentOS7)

OR

**dnf -y install libnsl\*** (RHEL8/RHEL9/CentOS 8/Fedora)

You will also need to have the ‘psmisc’ package installed:

**yum -y install psmisc** (RHEL7/CentOS7)

OR

**dnf -y install psmisc** (RHEL8/RHEL9/CentOS 8/Fedora)

OR

**apt install psmisc** (Ubuntu)

As an optional component to tune the POFR client sampling delay (see Section F5), you can also install the ‘perf’ tool:

**yum -y install perf**

OR

**dnf install perf**

Choose a directory not accessible by ordinary system users and clone the git repo:

**git clone** <https://github.com/gmagklaras/POFR.git>

Change into the cloned git repo directory and unpack the proper POFR PERL distribution based on what sort of distribution you run. For example, if you run a Fedora Linux client, type the following:

**cd POFR**

**tar xvfz pofrperlfedorax86\_64.tar.gz**

When the untaring of the compressed tarball completes, you should have a ‘pofrperl’ directory. This is the directory that contains the specially made PERL distribution that the client runs with all the necessary modules pre-installed. As an option, if you would like to save space, you can delete the rest of the compressed pofrperl tarballs. That’s all, the software for the POFR client is now installed.

## **D1)POFR Client Registration**

POFR client registration is a process that can be outlined in three steps:

- **Step 1:** On the system to monitor: Run the **pofrcreg.pl** script. The scripy will send the registration request to the specific POFR server. Once this is done, the client will wait for the server to complete the registration process.
- **Step 2:** Run the **pofrsreg.pl** script on the server side to register the new client registration.
- **Step 3:** Go back to the POFR client and fetch the credentials from the server to complete the registration.

This section will walks you through that process and explains it step by step.

**Step 1:** If you know the IP address and the registration password for your POFR server, change to the ‘client’ directory and invoke the ‘pofrcreg.pl’ client registration script. For example, if your POFR server is called *mypofrserver.domain.com*

**cd client**

**./pofrcreg.pl --server mypofrserver.domain.com**

The registration client will send a request to the POFR server via the SSH protocol. All requests are sent to a designated userid of the POFR server (by default pofrsreg). The client registration should respond by informing you of the server it will try to send the request to and it will ask you permission to proceed:

**Hostname is : mypofrserver.domain.com**

**Client IP is : 192.168.18.24**

**33373436-3933-5a43-3332-303941394a371728960233**

**pofrcreg: OK. Connecting to the specified POFR server: mypofrserver.domain.com to send our registration request.**

**scp -p ./request33373436-3933-5a43-3332-303941394a371728960233.pofr**

**pofrsreg@mypofrserver.domain.com:~/**

**Proceed [y/N]:**

If you are certain that this is the POFR server you wish to connect to, respond with a 'y' to the proceed prompt.

If this is the very first time you register this client to the server, the SSH protocol will ask you to verify that the SSH key fingerprint of the POFR server is the proper one:

**The authenticity of host 'mypofrserver.domain.com (192.168.18.24)' can't be established.**

**ECDSA key fingerprint is**

**SHA256:EMjv4RMWrk6+vartverX6d8SuiJElKi8IXMl4tqIX+rCs.**

**ECDSA key fingerprint is MD5:e3:03:e8:ab:37:37:2f:29:48:e9:b6:9c:b9:43:67:eb.**

**Are you sure you want to continue connecting (yes/no)?**

Again, if you are certain that this is the proper server, type 'yes'. The very next thing is a prompt to type the registration password:

**pofrsreg@mypofrserver.domain.com's password:**

Type the password for the default registration user 'pofrsreg' and once you hit 'Enter', if you entered the password correctly, you should get a verification by the client:

**pofrcreg: OK. Request sent successfully to server mypofrserver.domain.com**

**.pofrcreg.pl: Waiting for the POFR server mypofrserver.domain.com to respond on our request...**

At this stage, the POFR client has successfully sent a **registration request** to the server. The sent request has to be acted upon on the server side.

**Step 2:** If you have control of the POFR server, you need to switch to your server SSH session and execute the **server/pofrsreg.pl** script (refer to the POFR Server Installation

and Administration Section). If you do not have control of the POFR server, you need to follow the instructions of your POFR server admin:

**cd server**

**./pofrsreg.pl**

**dbusername is: root, dbname is: lhlt, on hostname: localhost**

**Changing password for user 4f37aa1565f806b673d0d46cb1a7a2fd.**

**passwd: all authentication tokens updated successfully.**

**pofrsreg.pl Status: Created database name is:**

**4c4c4544004a52108057c4c04f5634328449907 and cid is: 4c4c4544-004a-5210-8057-c4c04f5634328449907**

The above message verifies that the POFR server has performed the new client registration

**Step 3:** Navigate back to the POFR client SSH/console session. The POFR client will wait a bit and then ask you for the registration password again, this time to download the **registration response** credentials from the server:

**scp -pr pofrsreg@mypofrserver.domain.com:~/response4c4c4544-004a-5210-8057-c4c04f5634328449907.reg ./**

Instruct the registration process to proceed in the process of downloading the server registration request:

**Proceed [y/N]:y**

**pofrsreg@mypofrserver.domain.com's password:**

If all goes well, you should observe two things:

- The response from the client that all completed well:

```
#####  
#pofrcreg:STATUS:OK.Client 4c4c4544-004a-52108057c4c04f5634328449907  
#was registered at the POFR server: mypofrserver.domain.com. #  
#####
```

- Under the 'client' directory, you should see four files amongst the client code:
  - **.lcaf.dat** : File containing authentication credentials for the server
  - **response[NUMERICCLIENTID].reg**: A unique client ID response returned from the server containing registration status of the server.



- pofr.rsa: A file containing the private RSA key of the POFR client. This is NOT the root/system wide RSA key but keys for the POFR data transmission process.
- pofr-rsa.pub: A file containing the public RSA keys of the POFR client. Again, this is NOT the root/system wide public RSA key but keys for the POFR data transmission process.

If you reached this stage, congratulations, you have registered your client and you should be ready to begin data transmission. If not, it is best to remove the client registration (refer to Section D2) and start the registration process again.

## **D2)Deregistering a POFR client**

If you wish to remove a client from the POFR server, this can be achieved in two steps:

- **Step 1:** At the POFR client, you need to execute the **pofrclientderegister.pl** script. This script will stop all data collection processes and remove the state files obtained from the server. It will not remove the data from the server (see “Step 2”)
- **Step 2:** At the POFR server, you need to execute the **pofrcleanreg.pl** script, which removes the POFR database entries, all client processed data in the relational database, as well as any client remaining unprocessed data.

This section will walk you through that process and explain it step by step.

**Step 1:** At the POFR client, cd to the client directory and execute the pofrclientderegister.pl script:

**cd client**

**./pofrclientderegister.pl**

**pofrclientderegister.pl STATUS: WARNING: YOU ARE ABOUT TO REMOVE THIS CLIENT REGISTRATION STATE FILES.**

**pofrclientderegister.pl STATUS: Are you sure you wish to remove them from the system? (y/n)y**

The script will ask you to verify with a ‘y’ or ‘n’ if you indeed need to unregister the POFR client. If you type ‘y’, the necessary actions will be taken:

**pofrclientderegister.pl STATUS: Operator chose to proceed with the process of removing the client registration files.**

**pofrclientderegister.pl STATUS: Inside the getusername subroutine function: Detected fs username 4873d4c04d7780e5f0ea91bf27bc6676**

```
pofrclientderegister.pl      STATUS:      Detected      fs      username
4873d4c04d7780e5f0ea91bf27bc6676
pofrclientderegister.pl STATUS: Inside the unregisterclient subroutine: Stopping
all POFR client ACTIVE processes.
/home/georgios/pubPOFR/POFR/client
stopclient.pl Info: No active POFR client processes to stop. Exiting.
pofrclientderegister.pl STATUS: Removing all relevant files now
/home/georgios/pubPOFR/POFR/client
pofrclientderegister.pl STATUS: All done. This removed all POFR client
registration files and stopped active processes.
pofrclientderegister.pl STATUS: Don't forget to run the pofrcleanreg.pl on the
*POFR server* to fully complete the process of deregistering the client.
pofrclientderegister.pl STATUS: The exact command you need to type on the
POFR server is:
pofrclientderegister.pl      STATUS:      ./pofrcleanreg.pl      --usertoremove
4873d4c04d7780e5f0ea91bf27bc6676
```

**Step 2:** Switch to the server. As suggested by the client side unregistration script, you need to do the following:

```
cd server
./pofrcleanreg.pl --usertoremove 4873d4c04d7780e5f0ea91bf27bc6676
pofrcleanreg.pl STATUS: Detected user to remove
4873d4c04d7780e5f0ea91bf27bc6676 in the database. WARNING: YOU ARE
ABOUT TO REMOVE THAT CLIENT FROM THE POFR DATABASE.
pofrcleanreg.pl STATUS: ALL DATA will be lost! Are you sure you wish to remove
use 4873d4c04d7780e5f0ea91bf27bc6676 from the system? (y/n)
```

As with the client unregistration part (in Step 1), you are warned again before you confirm proceeding (y/n):

```
pofrcleanreg.pl STATUS: Operator chose to proceed with the removal of user
4873d4c04d7780e5f0ea91bf27bc6676.
pofrcleanreg.pl STATUS: Detected user to remove
4873d4c04d7780e5f0ea91bf27bc6676 in the database. Proceeding with the removal.
pofrcleanreg.pl STATUS: Removed entry for user
4873d4c04d7780e5f0ea91bf27bc6676 from the lhlt.lhlttable.
pofrcleanreg.pl STATUS: Removed database 4c4c4544-0050-3210-8056-
b8c04f5a593269537931 for user 4873d4c04d7780e5f0ea91bf27bc6676 .
```

```
pofrcleanreg.pl STATUS: Found the filesystem directory for user to remove
4873d4c04d7780e5f0ea91bf27bc6676 ...
unlink /home/4873d4c04d7780e5f0ea91bf27bc6676/.ssh/authorized_keys
rmdir .ssh
unlink /home/4873d4c04d7780e5f0ea91bf27bc6676/.bash_profile
unlink /home/4873d4c04d7780e5f0ea91bf27bc6676/.bash_logout
rmdir extensions
rmdir plugins
rmdir .mozilla
unlink /home/4873d4c04d7780e5f0ea91bf27bc6676/.bashrc
rmdir /home/4873d4c04d7780e5f0ea91bf27bc6676
pofrcleanreg.pl STATUS: Filesystem directory for user
4873d4c04d7780e5f0ea91bf27bc6676 removed. All good. Bye, bye!
```

This completes a POFR client removal/deregistration process.

It should be noted that both the **pofrclientderegister.pl** client script as well as its server counterpart **pofrcleanreg.pl** can be used in batch non interactive mode by means of the **-batch** switch. For example to achieve the previously mentioned operations, one could type:

```
./pofrclientderegister.pl
```

and

```
./pofrcleanreg.pl --usertoremove 4873d4c04d7780e5f0ea91bf27bc6676
```

to achieve the same results without the y/n confirmation, a feature useful for automation/deployment provisioning mechanisms (ansible, terraform, kickstart).

## E)POFR Server installation

A server installation is a little bit more complex than that of a client. We recommend a Fedora 35/36 or RHEL9 and derivatives distro for best results.

As the POFR server runs performance intensive workloads, it is recommended to run it on a dedicated server, physical or virtual (see Section ‘*Dependencies and preparation requirements*’ of this document).

**Step 1:** First ensure that your system has the legacy support library for NIS. As the root user, please install:

**dnf -y install libnsl\***

You will also need to install the ‘scponly’ package for the POFR server to create secure non login accounts

**dnf -y install scponly**

***Note for RHEL 9 server installations only:*** As RHEL 9 has deprecated OpenSSH SCP, there is no ‘scponly’ package easily available. For that reason, we have taken the hardened patched config of Fedora 36 and produced the a RHEL9 rpm that is accessible as part of the POFR software distribution, under the extensions folder. Thus, for RHEL 9 server installs, instead of the previous ‘dnf -y install’ command, you should instead:

i) Verify that the produced package matches the one listed in the file ‘scponly-4.8-29.el9.x86\_64.rpm.sha256sum’ of the repo:

**sha256sum POFR/extensions/scponly-4.8-29.el9.x86\_64.rpm**

ii) If and only if the produced sha256sum matches the one on the file, proceed by installing the package:

**dnf -y localinstall POFR/extensions/scponly-4.8-29.el9.x86\_64.rpm**

Start as the root user and choose a directory not reachable by ordinary system users and clone the git repo:

**git clone <https://github.com/gmagklaras/POFR.git>**

**Step 2:** Change into the cloned git repo directory and unpack the proper POFR PERL for a Fedora distro as shown below:

**cd POFR**

**tar xvfz pofrperlfedorax86\_64.tar.gz**

**Step 3:** Install ‘scponly’ and a MariaDB RDBMS server. POFR has been tested with the default MariaDB version of an actively maintained Fedora distro (version 10.5.x for Fedora 34):

**dnf install -y sponly mariadb mariadb-server**

To ensure that the MariaDB server runs in a secure and scalable manner, please enter the following in the [mysqld]/[mariadb] section of the **/etc/my.cnf** file:

```
[mariadb]  
bind-address=127.0.0.1  
max_connections=1000
```

You can now enable and start the MariaDB server:

```
systemctl enable mariadb.service  
systemctl start mariadb.service
```

Check that the MariaDB server has started properly with the following command:

```
systemctl status mariadb.service
```

If all is well, you should get a response like the one below:

```
Loaded: loaded (/usr/lib/systemd/system/mariadb.service; enabled; vendor preset: disabled)  
Active: active (running) since Fri 2021-06-04 16:08:58 CEST; 36s ago  
....  
Status: "Taking your SQL requests now..."  
Tasks: 17 (limit: 4664)  
Memory: 69.5M  
CPU: 453ms  
CGroup: /system.slice/mariadb.service  
└─17349 /usr/libexec/mariabdb --basedir=/usr
```

In addition it is also worth checking and that the server is indeed listening on the localhost (no remote client connections are allowed/needed for POFR):

```
lsof -i | grep mariadb
```

should provide a a response like the one below:

```
mariabdb 1479 mysql 20u IPv4 18114 0t0 TCP localhost:mysql (LISTEN)
```

**Step 4:** Ensure that an sshd server is installed, is running and is not blocked by the local firewall:

A Fedora 33/34 installation normally includes by default an SSH server. If, for for some reason it does not, as user root, you can install it by typing:

**dnf -y install openssh-server**

You will then need to ensure that the server is systemd enabled and started properly:

**systemctl enable sshd.service**

**systemctl start sshd.service**

Check that the sshd service is running by confirming its status:

**systemctl status sshd.service**

The above command should give you a status similar to the one below, if sshd is installed, enabled and started properly:

● **sshd.service - OpenSSH server daemon**

**Loaded:** loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor preset: disabled)

**Active:** active (running) since Sun 2021-06-20 17:55:47 CEST; 32min ago

**Docs:** man:sshd(8)

man:sshd\_config(5)

**Main PID:** 883 (sshd)

**Tasks:** 1 (limit: 9495)

**Memory:** 7.6M

**CPU:** 1.922s

**CGroup:** /system.slice/sshd.service

└─883 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups

...

Finally, the last check is to ensure that the sshd service is not blocked by the local firewall and thus ensure your POFR clients will be able to contact the server. The default Fedora distro firewall config is controlled by firewalld-service which should be up and running (check with **systemctl status firewalld.service**). To check that the sshd.service is not blocked by firewalld.service, type:

**firewall-cmd --list-all**

A non blocking configuration would list 'ssh' in the 'services:' line:

**FedoraWorkstation (active)**

**target:** default

```
icmp-block-inversion: no
interfaces: enp1s0
sources:
services: dhcpv6-client mdns samba-client ssh
ports: 1025-65535/udp 1025-65535/tcp
protocols:
...
```

If you do not see the ssh service listed in the services line, you can enable it by typing:

```
firewall-cmd --add-service=ssh --permanent
```

The above command should normally indicate ‘success’ if it is executed properly and at that point, you should re-run the command:

```
firewall-cmd --list-all
```

and verify that ‘ssh’ is included in the ‘services:’ line.

**Step 5:** Provisionally install your preferred terminal multiplexing utility (screen/tmux) and the ‘htop’ utility. These tools will come in handy to start/stop the server and check performance utilization on your multi-core server:

```
dnf -y install screen tmux htop
```

**Step 6:** Continuing using the root account, secure your fresh MariaDB installation by running the following script:

```
mysql_secure_installation
```

The script will ask your input on a number of choices, outlined below. The current password on a fresh installation is blank (press Enter):

```
Enter current password for root (enter for none):  
OK, successfully used password, moving on...
```

You should switch to unix\_socket authentication:

```
Switch to unix_socket authentication [Y/n] Y  
Enabled successfully!  
Reloading privilege tables..
```

**... Success!**

Choose a secure root password for the MariaDB server and set this password up by entering it twice:

**Change the root password? [Y/n] Y**

**New password:**

**Re-enter new password:**

**Password updated successfully!**

**Reloading privilege tables..**

**... Success!**

Remove the anonymous users from your fresh installation:

**Remove anonymous users? [Y/n] Y**

**... Success!**

Disallow remote root login to maintain the security of your installation:

**Disallow root login remotely? [Y/n] Y**

**... Success!**

Remove the test database and access to it:

**Remove test database and access to it? [Y/n] Y**

**- Dropping test database...**

**... Success!**

**- Removing privileges on test database...**

**... Success!**

Finally, please reload the MariaDB privilege tables to ensure that the security changes you made are applied to the running instance of the MariaDB server:

**Reload privilege tables now? [Y/n] Y**

**... Success!**

**Cleaning up...**

**Step 7:** At this point, check that you can login to the RDBMS with your newly chosen MariaDB root password:



**mysql -u root -p**

If you have changed properly, you should have no problem getting into the MariaDB SQL prompt:

**Welcome to the MariaDB monitor. Commands end with ; or \g.**

**Your MariaDB connection id is 13**

**Server version: 10.5.10-MariaDB MariaDB Server**

**Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.**

**Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.**

**MariaDB [(none)]>**

While you are there, see if you have permission to create a database by typing the following at the prompt:

**MariaDB [(none)]> create database lhlt;**

If you get a:

**Query OK, 1 row affected (0.001 sec)**

you have proved that the basic permissions are set for MariaDB. Exit the MariaDB prompt by typing 'quit'.

**Step 8:** Now navigate to the 'server' directory of the POFR repo

**cd server**

and create the schema for the host lookup table (type your chosen MariaDB root password at the prompt and press Enter):

**cat LHLT.sql | mysql -u root lhlt -p**

If you do not get any error/warning messages, you have probably created the lookup table properly. You can verify that from the SQL prompt with this:

**MariaDB [(none)]> describe lhlt.lhlttable;**

Field	Type	Null	Key	Default	Extra
hostid	bigint(20)	NO	PRI	NULL	auto_increment
uuid	varchar(36)	NO		NULL	
cid	varchar(59)	NO		NULL	

ciduser	varchar(32)	NO	NULL
lastip	varchar(35)	YES	NULL
ryear	smallint(6)	NO	NULL
rmonth	tinyint(4)	NO	NULL
rday	tinyint(4)	NO	NULL
rmin	tinyint(4)	NO	NULL
rhour	tinyint(4)	NO	NULL
rsec	tinyint(4)	NO	NULL

11 rows in set (0.004 sec)

If you get something like the above output, you are all set. Type ‘quit’ to exit back to the OS shell.

**Step 9:** Create the authentication database file (.adb.dat). You should be under the server folder and the easiest way to do this is the following:

```
echo "root,lhlt,MY_MARIA_DB_ROOT_PASS,localhost" > .adb.dat
```

**Important:** Replace the string **MY\_MARIA\_DB\_ROOT\_PASS** with whatever root password you chose when you secured your MariaDB instance (Step 5).

**Step 10:** An equally important next task is to create the POFR registration user ‘pofrsreg’. This user will be used to register POFR clients in the server. You should create this user locally on the POFR server with the following command (as user root):

```
useradd -d /home/pofrsreg -s /bin/scponly pofrsreg
```

Create a secure password for that user by typing and verifying it after issuing this command:

```
passwd pofrsreg
```

Take good care of this password. You will be quoting/using it for every client registration (refer to Section ‘**POFR Client registration**’ of this document).

This concludes the installation of the POFR server.

## **F)POFR usage**

After completing the client and server installation procedures (Sections D and F of this manual), using POFR involves starting and stopping the client, as well as logging into the server to browse/search the collected data. This section will also demonstrate basic troubleshooting steps to ensure proper operation of the POFR engine.

### **F1)Starting and stopping the POFR client**

POFR chooses a simple/minimal way to start the client outside the systemd framework. The systemd framework is excellent for starting and stopping services, the decision for not using it for POFR is a design choice. A growing number of Linux malware attacks target the systems via systemd. In addition, if one wishes to forensically audit aspects of systemd's operation, it is best to not insert code/dependencies that can affect its performance/state. For these reasons and while it is feasible to make a system service file, it is recommended not to use systemd for the purposes of starting/stopping the POFR client. Use instead the scripts and methods outlined below.

As the 'root' user, fire up your favourite terminal multiplexer (screen/tmux/other) and execute the startup script. For example:

#### **screen -R client**

and then in the screen session, change to the POFR/client directory location and issue something like the following:

```
./startclient.pl >> /var/log/pofrclient.log 2>&1
```

At that point and if you have SSH connectivity between the client and the server, all the processes to collect and periodically send the data to the server. You could then detach from the 'screen' session. You could also inspect the log file you specified in the startup commands (/var/log/pofrclient) to see the status messages for the client.

If at any time, you wish to stop the client, you could navigate to the POFR/client directory and call 'stopclient.pl' script:

```
./stopclient.pl
```

The above command will stop all the necessary processes.

Should you wish to restart the client process you could re-attach to the screen session (screen -r client) and reissue the 'startclient.pl' script command before you detach again.

The above method is useful for interactive use on the client for testing our troubleshooting purposes. Once you are done with testing and troubleshooting, a more automated method that does not involve a terminal multiplexer, you could just use a cronjob as the root account by adding to the root account's crontab the following lines (**crontab -e**):

**HOME=[INSTPATH]/POFR/client**

**\*/5 \* \* \* \* [INSTPATH]/POFR/client/startclient.pl >> /var/log/pofrclient.log 2>&1**

where **[INSTPATH]**=the PATH of the directory under which you have installed POFR

For instance, if you have installed the git repo under /root, Figure 2 shows what you can type on the client's root crontab (**crontab -e**):



```
GNU nano 5.8
*/5 * * * * cd /root/POFR/client; ./startclient.pl >> /var/log/pofrclient.log 2>&1
```

**Figure 2: Automating the POFR client startup**

This would ensure that the services start automatically without terminal multiplexers and remain up for as long as the client OS is running AND has connectivity with the POFR server.

**Note:** It is good to ensure that the log file (**/var/log/pofrclient**) is logrotated to avoid filling up your disk/partition/volume. Consult the logrotate configuration of your linux distribution and adjust the log retention/rotation in terms of your requirements.

## **F2)Checking data reception on the POFR server**

After completing one or more POFR client registrations (Section D1) and starting the monitoring process (Section F1), you can check that data are received properly on the server side. Starting on the client side (POFR/client directory), take a look at the contents of the **response[ID].reg** file which was created on the client as a result of the client registration process. For example, a POFR client might have a file looking like this:

**responseab6fcbe3-3d1b-460d-96aa-11cde64b246a15610790.reg**

If you take a look at the contents of the file, you see a single status line looking like the one below:

**Status:GRANTED#71cc46a1b55e4a3f6b64af63d4cc5fc4#a7f1cb7c72f3bbcaea02105b358dc497e3e882519edb2c712ed704f43a54f613**

The ‘GRANTED’ string indicates that the registration was granted by the POFR server. The second string “71cc46a1b55e4a3f6b64af63d4cc5fc4” is the userid of the account that the data is sent via SSH on the POFR server. The third is a unique registration ID for the POFR client employed by the POFR server to detect/avoid duplicate registrations

You can ssh now on the POFR server, switch to the root user and navigate to the `/home/71cc46a1b55e4a3f6b64af63d4cc5fc4` directory. If the client is pushing data to the server successfully, you should be able to do an `ls -ltah` and see an output like the one below:

```
-rw-r--r--. 1 71cc46a1b55e4a3f6b64af63d4cc5fc4 71cc46a1b55e4a3f6b64af63d4cc5fc4 1.5M Jun
20 19:24
1624209875762201#+0200#dcf647ebdf3b2018c745743f3c3d300c4eb08f4f501dcededd9850cb408ff4
c7.tar
-rw-r--r--. 1 71cc46a1b55e4a3f6b64af63d4cc5fc4 71cc46a1b55e4a3f6b64af63d4cc5fc4 1.5M Jun
20 19:23
1624209830214172#+0200#6a2dd9da9be0cd93d9532e083a2a60c0b1405119a2bdc26b2aeaa611972
caf06.tar
-rw-r--r--. 1 71cc46a1b55e4a3f6b64af63d4cc5fc4 71cc46a1b55e4a3f6b64af63d4cc5fc4 1.5M Jun
20 19:23
1624209784671618#+0200#fcb7968ab73de5c23b69c73a65f2cd2d032450d06469ab542f6dc7b2bb58
4021.tar
-rw-r--r--. 1 71cc46a1b55e4a3f6b64af63d4cc5fc4 71cc46a1b55e4a3f6b64af63d4cc5fc4 1.5M Jun
20 19:22
1624209739033015#+0200#5687b186918b71cb932a2d1d0530ee9c0cc37f5986fcc456a9df26a502bc3
f65.tar
-rw-r--r--. 1 71cc46a1b55e4a3f6b64af63d4cc5fc4 71cc46a1b55e4a3f6b64af63d4cc5fc4 1.5M Jun
20 19:21
1624209693446555#+0200#82416a1edb42f15dd4130bf694613d5ccedbfff04fb443335c8c1d7ec365bd
c34.tar
...
```

Each one of these tar files is a signed with a SHA message digest and time-stamped snapshot of the client data (look at the time stamps to see the frequency of reception) that was uploaded through the SSH channel. This indicates proper communication

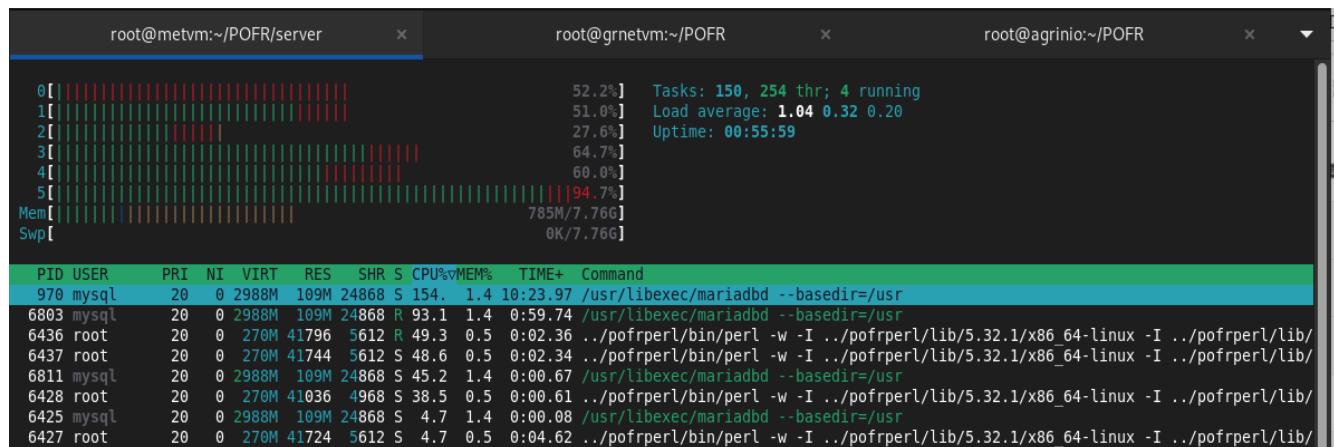
between the POFR client and server entities. If, after several minutes of starting the POFR client, you have no files on the POFR server, please refer to the troubleshooting section.

### F3)Starting and automating the server data parsing

Once you have confirmed proper data reception on the server (Section F2), you will need to automate the process of parsing the received data. Prior doing that and if you have installed the server for the first time, it's worth starting the data parsing process manually, to check that everything is working. It's a good idea to grab a cup of your favorite beverage and let the client(s) send enough data to the server (say you leave the server collect data from active clients for a couple of hours). Then, as user 'root', change to the git cloned repo directory and in particular the 'server' subdirectory and then issue the following command:

```
./newdeltaparseproc.pl > run01.txt 2>&1
```

The above command will call the main data parsing script and will redirect standard and error output to the file 'run01.txt'. This file will contain various debugging messages that can inform you of what the various processing threads are doing. Once you issue the above command, a good test is to run '**top**' (or even better an '**htop**' that can also give you threaded process information).



**Figure 3: An active POFR server**

If you see various processes POFR Perl interpreter processes and a busy MariaDB RDBMS process (Figure 3), things should be progressing well. You could also like to verify that from the command line (again user 'root') with the command:

**ps auxwww | grep pofr**

If all is well, you should get multiple instances of the POFR Perl client, as shown below:

```
root      6427  2.5  0.5 276624 41744 pts/0    S   18:47   0:08 ../pofrperl/bin/perl -w -I
../pofrperl/lib/5.32.1/x86_64-linux -I ../pofrperl/lib/5.32.1 ./newdeltaparseproc.pl
root      6439  1.9  0.5 276896 41772 pts/0    S   18:47   0:06 ../pofrperl/bin/perl -w -I
../pofrperl/lib/5.32.1/x86_64-linux -I ../pofrperl/lib/5.32.1 ./newdeltaparseproc.pl
root      6440  0.0  0.4 275528 39428 pts/0    S   18:47   0:00 ../pofrperl/bin/perl -w -I
../pofrperl/lib/5.32.1/x86_64-linux -I ../pofrperl/lib/5.32.1 ./newdeltaparseproc.pl
```

The 'run01.txt' file should contain output similar to the one below:

```
parseproc.pl Error: Could not detect /dev/shm/luarmserver/net dir...Fresh boot? Creating it...
user 71cc46a1b55e4a3f6b64af63d4cc5fc4:size of myprocfiles is 3284 and of of threadflags is 0
Processing POFR server: metvm on IP: 192.168.122.244
Running on 6 server cores and having 1 active users.
Users are: 71cc46a1b55e4a3f6b64af63d4cc5fc4
Processing user 71cc46a1b55e4a3f6b64af63d4cc5fc4
...
```

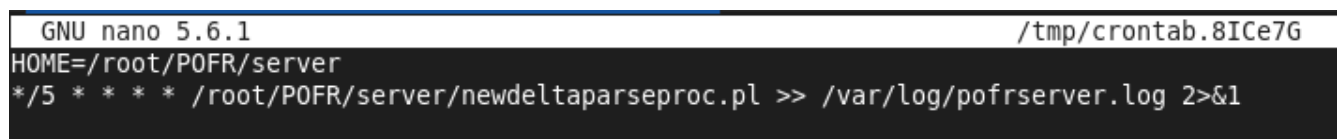
This indicates that the newdeltaparseproc.pl script is creating processing threads to parse the collected client data in parallel. If you do not see active processes as described above, then the output file should contain an error message to help locate the source of the problem. In such a case, you might need to consult again Section E of this manual.

If you have success in seeing active parsers, you can now proceed with the process of automating the execution of the newdeltaparseproc.pl script by means of creating (user root) a crontab entry with the following content:

```
HOME=[INSTPATH]/POFR/server
*/5 * * * * [INSTPATH]/POFR/server/newdeltaparseproc.pl >>
/var/log/pofrserver.log 2>&1
```

where **[INSTPATH]**=the PATH of the directory under which you have installed POFR

For example, if you installed the git repository under *root*, Figure 4 displays exactly what you can write/add in the root crontab (**crontab -e**):



```
GNU nano 5.6.1 /tmp/crontab.8ICe7G
HOME=/root/POFR/server
*/5 * * * * /root/POFR/server/newdeltaparseproc.pl >> /var/log/pofrserver.log 2>&1
```

**Figure 4: Automating the execution of the POFR server script**

## F4) Browsing and understanding collected data

```
CREATE TABLE `psinfo` (  
  `psentity` bigint(20) NOT NULL AUTO_INCREMENT,  
  `shanorm` char(40) NOT NULL,  
  `shafull` char(40) NOT NULL,  
  `uid` mediumint NOT NULL,  
  `pid` mediumint NOT NULL,  
  `ppid` mediumint NOT NULL,  
  `command` text NOT NULL,  
  `arguments` mediumtext,  
  `tzzone` char(6) NOT NULL,  
  `cyear` smallint(6) NOT NULL,  
  `cmonth` tinyint(4) NOT NULL,  
  `cday` tinyint(4) NOT NULL,  
  `cmin` tinyint(4) NOT NULL,  
  `chour` tinyint(4) NOT NULL,  
  `csec` tinyint(4) NOT NULL,  
  `cmsec` mediumint(6) NOT NULL,  
  `dyear` smallint(6) DEFAULT NULL,  
  `dmonth` tinyint(4) DEFAULT NULL,  
  `dday` tinyint(4) DEFAULT NULL,  
  `dhour` tinyint(4) DEFAULT NULL,  
  `dmin` tinyint(4) DEFAULT NULL,  
  `dsec` tinyint(4) DEFAULT NULL,  
  `dmsec` mediumint(6) DEFAULT NULL,  
  PRIMARY KEY (`psentity`)  
) ENGINE=MyISAM AUTO_INCREMENT=19470 DEFAULT CHARSET=latin1 COMMENT='This holds per host process execution data';  
  
CREATE TABLE `fileinfo` (  
  `fileaccessid` bigint NOT NULL AUTO_INCREMENT,  
  `shasum` char(40) NOT NULL,  
  `filename` varchar(300) NOT NULL,  
  `uid` mediumint NOT NULL,  
  `command` text NOT NULL,  
  `pid` mediumint NOT NULL,  
  `ppid` mediumint NOT NULL,  
  `tzzone` char(6) NOT NULL,  
  `cyear` smallint(6) NOT NULL,  
  `cmonth` tinyint(4) NOT NULL,  
  `cday` tinyint(4) NOT NULL,  
  `cmin` tinyint(4) NOT NULL,  
  `chour` tinyint(4) NOT NULL,  
  `csec` tinyint(4) NOT NULL,  
  `cmsec` mediumint(6) NOT NULL,  
  `dyear` smallint(6) DEFAULT NULL,  
  `dmonth` tinyint(4) DEFAULT NULL,  
  `dday` tinyint(4) DEFAULT NULL,  
  `dhour` tinyint(4) DEFAULT NULL,  
  `dmin` tinyint(4) DEFAULT NULL,  
  `dmsec` mediumint(6) DEFAULT NULL,  
  PRIMARY KEY (`fileaccessid`)  
) ENGINE=MyISAM AUTO_INCREMENT=246450 DEFAULT CHARSET=ucs2;  
  
CREATE TABLE `netinfo` (  
  `endpointinfo` bigint(20) NOT NULL AUTO_INCREMENT,  
  `cyear` smallint(6) NOT NULL,  
  `cmonth` tinyint(4) NOT NULL,  
  `cday` tinyint(4) NOT NULL,  
  `chour` tinyint(4) NOT NULL,  
  `cmin` tinyint(4) NOT NULL,  
  `csec` tinyint(4) NOT NULL,  
  `cmsec` mediumint(6) NOT NULL,  
  `tzzone` char(6) NOT NULL,  
  `transport` tinytext NOT NULL,  
  `sourceip` tinytext NOT NULL,  
  `sourcefqdn` tinytext,  
  `sourceport` smallint(6) unsigned NOT NULL,  
  `destip` tinytext NOT NULL,  
  `destfqdn` tinytext,  
  `destport` smallint(6) unsigned NOT NULL,  
  `ipversion` tinyint(4) NOT NULL,  
  `pid` mediumint NOT NULL,  
  `uid` mediumint NOT NULL,  
  `inode` int unsigned NOT NULL,  
  `dyear` smallint(6) DEFAULT NULL,  
  `dmonth` tinyint(4) DEFAULT NULL,  
  `dday` tinyint(4) DEFAULT NULL,  
  `dhour` tinyint(4) DEFAULT NULL,  
  `dmin` tinyint(4) DEFAULT NULL,  
  `dsec` tinyint(4) DEFAULT NULL,  
  `dmsec` mediumint(6) DEFAULT NULL,  
  `shasum` char(40) NOT NULL,  
  PRIMARY KEY (`endpointinfo`)  
) ENGINE=MyISAM AUTO_INCREMENT=2075 DEFAULT CHARSET=latin1 COMMENT='This table contains the endpoint info';
```

Figure 5: The SQL schema of the POFR psinfo, fileinfo and netinfo tables



After completing the previous section (F3), you should be able to have a basic setup up and running. This section will demonstrate how to use the relational database client to view the collected data.

Prior getting into practical details, it is good to have a conceptual view of how information is structured at the relational layer. This will help you comprehend how POFR organizes the data and which database tables to query, depending on what kind of information you are looking for.

If you are not familiar with POFR, the first thing you should examine is the basic structure of the relational schema by inspecting the [itpsqlschema.sql file](#) of the source code, part of which is shown in Figure 5. Information is structured in three basic relational tables:

- **psinfo**: It collects information about process creation. Every process recorded by the system is stored together with associating information such as its user and group identifiers, the actual command and its arguments.
- **fileinfo**: It collects information about file access activity by all processes that are accessing files. This includes special files (devices, pipes, sockets). Apart from the file name, correlating information such as the associated process, user and parent process identifiers.
- **netinfo**: This table collects information about the creation of network endpoints (sockets). Apart from the usual information (source and destination IP and port), other associative information is logged, including the process and user identifiers of the process that created the endpoint.

The rest of the tables are intended for assistance/management information, but the vital bits about the recorded events are only recorded in the previously mentioned tables and these are the tables that will receive investigative queries. This relational schema should enable an investigator to obtain answers after issuing queries of varying degree of complexity and specificity about a system. Examples include:

- Give me all the files the web server process “httpd” accessed under the “/usr” partition.
- Did user ‘toma’ executed the application nmap with the arguments “-P0 192.168.18.34”?
- Display all the network endpoints created/accessed by the ‘google-chrome’ web browser instances of user ‘nathanp’ and see if a particular website’s IP address was accessed.

- Has the monitored system been portmapped for ports 137 and 22 by IP addresses originating from Vietnam? If yes, collect the list of IP addresses for the last 24 hours, especially between the hours of 01:00-04:00 on the local timezone of the client system.

The previous examples demonstrate the type of information correlation you could achieve on a POFR setup. An equally important concept concerns the temporal aspects of how the information is logged and organized.

POFR clients push snapshots of system activity to the server and use **three levels of temporal assembly** to achieve event capture accuracy. The first level concerns the snapshots themselves, as they are dumped by each logging thread. These snapshots contain a lot of redundant information to ensure event capture accuracy. The second level archives the info from all the threads and removes some of the record redundancy. Finally, the third level is the one you should be mostly using to view the records and contains curated records of all events with the redundancy removed.

```
MariaDB [3337343639335a433332303941394a373290862469]> select dbname,ciduser,lastip from lhlt.lhlttable;
```

dbname	ciduser	lastip
4c4c4544003446108047b6c04f5337333309265	9087c4dc372e542555311c3d35f9eb67	192.168.8.8

**Figure 6: The POFR host to database lookup table**

These concepts are best demonstrated if we use the relational database client and start looking into the collected information. At the POFR server ensure that you are logged in as the RDBMS root user. One of the first things you have to do is to lookup the database name of your client, so that you know which database to query. You can do that by querying the host database lookup table (*lhlt.lhlttable*) as shown in Figure 6. The ‘*lastip*’ column records that monitored system IP address. The ‘*dbname*’ column gives you the database name. The ‘*ciduser*’ column gives you the user identifier on the POFR server for that specific system. So, if you navigate (as root) under the ‘*/home/ciduservalue*’ directory, you should be able to see the compressed snapshot tarballs that the client sends.

Let’s choose to browse data from a test client system that has a dbname value name of ‘63406’, as shown above. To do this, we switch the database with a value of 63406, as shown in Figure 7. The SQL statements above select the database and show its tables in their initial state (once the client has been registered and before the parsing of the data)

of the first level of temporal analysis. The 'psinfo', 'fileinfo' and 'netinfo' tables should be empty.

```
MariaDB [(none)]> use 63406;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [63406]> show tables;
+-----+
| Tables_in_63406 |
+-----+
| fileinfo         |
| groupinfo        |
| hostinfo         |
| hwinfo           |
| netinfo          |
| netint           |
| netroute         |
| psinfo           |
| urlinfo          |
| userinfo         |
+-----+
10 rows in set (0.001 sec)
```

**Figure 7: Selecting an empty host database in POFR**

Once the parsing of data for that client has started (Section F3), the output of the 'show tables' SQL statement will be very different. For each of the 32 processing threads that will be launched in parallel for that client to perform the data ingestion and parsing, there will be a triad of time-stamped instances of tables such as the one below:

***fileinfo16350802815025041635080420587772***  
***netinfo16350802815025041635080420587772***  
***psinfo16350802815025041635080420587772***

In total you should get  $32 \times 3 = 96$  UNIX epoch-ed time-stamped instances of these tables during the first processing cycle of the data ingestion and snapshot assembly process. At the end of this cycle, when every single one of the 32 processing threads has completed, you should examine the contents of the psinfo, fileinfo and netinfo non time-stamped tables, as shown below. They should contain the merged view of the data of each of the previous 32 threads

```

MariaDB [63406]> select cday,cmonth,year,chour,cmin,csec,command,arguments,pid,ppid,uid from psinfo LIMIT 10 ;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| cday | cmonth | year | chour | cmin | csec | command | arguments | pid | ppid | uid |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /init ro | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /init ro | 6 | 1 | 11 |
| 24 | 10 | 2021 | 5 | 59 | 25 | bash | -bash | 7 | 6 | 4 |
| 24 | 10 | 2021 | 5 | 59 | 25 | sudo | sudo -i | 25 | 7 | 5 |
| 24 | 10 | 2021 | 5 | 59 | 25 | bash | -bash | 26 | 25 | 4 |
| 24 | 10 | 2021 | 6 | 3 | 26 | init | /init ro | 1 | 0 | 10 |
| 24 | 10 | 2021 | 6 | 3 | 26 | init | /init ro | 6 | 1 | 11 |
| 24 | 10 | 2021 | 6 | 3 | 26 | bash | -bash | 7 | 6 | 4 |
| 24 | 10 | 2021 | 6 | 3 | 26 | sudo | sudo -i | 25 | 7 | 5 |
| 24 | 10 | 2021 | 6 | 3 | 26 | bash | -bash | 26 | 25 | 4 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
10 rows in set (0.001 sec)

MariaDB [63406]> select cday,cmonth,year,chour,cmin,csec,command,filename,pid,ppid,uid from fileinfo LIMIT 10 ;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| cday | cmonth | year | chour | cmin | csec | command | filename | pid | ppid | uid |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /dev/null | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /dev/null | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /dev/lxss | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /unknown | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | / | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | socket:[8] | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | anon_inode:[eventpoll] | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | anon_inode:[eventpoll] | 1 | 0 | 10 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /dev/null | 6 | 1 | 11 |
| 24 | 10 | 2021 | 5 | 59 | 25 | init | /dev/kmsg | 6 | 1 | 11 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
10 rows in set (0.006 sec)

```

**Figure 8: Browsing freshly ingested data in POFR**

In the next processing cycle, a succeeding fresh set of 32 processing threads will create an additional set of 96 epoch-ed time-stamped tables. Once again, at the end of the cycle, information will be merged in the respective psinfo, fileinfo and netinfo tables. The cycles continue in this **first level of temporal assembly** and analysis continues up to a certain point, when the POFR system will detect that certain processing limits have been reached. These limits are concerned with computational performance issues.

To counteract the undesirable effects of record redundancy and ensure that the server is able to keep up with the amount of data, the system will create archived versions of these merged tables with the redundant records removed, once it senses either that:

- we have more than a certain number of first level tables
- the merged view contains more than a certain number of file access records.

All this is regulated by the server [mergetables.pl script](#). This script will produce **second level of temporal assembly** and is called automatically by the system. If you leave the client to log data for several hours/days, you should eventually see triads of archived tables named like the ones below:

**archfileinfo20211024191757to20211025071848**

**archnetinfo20211024191757to20211025071848**

**archpsinfo20211024191757to20211025071848**

coexisting with the previous first-level tables. These archive tables are named using the following convention:

- **archfileinfoyyyymmddhhmmssstoyyyyymmddhhmmss**
- **archnetinfoyyyymmddhhmmssstoyyyyymmddhhmmss**
- **archpsinfoyyyymmddhhmmssstoyyyyymmddhhmmss**

where:

*yyyy* → four digit year representation  
*mm* → two digit month representation  
*dd* → two digit day representation

For example, the previous triad of tables represent snapshot data from the 24<sup>th</sup> of October 2021 and local client time 19:17:42 to 25<sup>th</sup> of October 2021 and local client time 07:18:48.

Figure 9 displays part of the SQL listing of POFR tables on a client that has ingested data for a little bit more than a day. If you look carefully at the time stamp of the arch\* tables, you will see that the collected data start on November 7<sup>th</sup> 2021 at 10:49:26 and end on November 8<sup>th</sup> at 17:54:37. The first level temporal analysis tables show also in the listing, containing the latest snapshot data (data ingested after November 8, 17:54:37 that have not been archived yet).

If you browse the contents of the second level archive tables with suitable SQL statements, you will discover the same schema structure as the one encountered in the first level of temporal assembly tables (Figures 7 and 8). The difference on this second level of archived tables is that there is a lesser amount of redundant records and that the duration is greater, in comparison to the first level tables that contain shorter and more recent snapshots.

```

MariaDB [EA4AFEE2FBE4266484CD00224D37027413572428086]> show tables;
+-----+
| Tables_in_EA4AFEE2FBE4266484CD00224D37027413572428086 |
+-----+
| archfileinfo20211107104926to20211107210853 |
| archfileinfo20211107211211to20211108073152 |
| archfileinfo20211108073507to20211108175437 |
| archnetinfo20211107104926to20211107210853 |
| archnetinfo20211107211211to20211108073152 |
| archnetinfo20211108073507to20211108175437 |
| archpsinfo20211107104926to20211107210853 |
| archpsinfo20211107211211to20211108073152 |
| archpsinfo20211108073507to20211108175437 |
| fileinfo |
| fileinfo16363943210400101636394466933534 |
| fileinfo16363945155597341636394661431123 |
| fileinfo16363947101798941636394855921874 |
| fileinfo16363949046404331636395050470116 |
| fileinfo16363950990872001636395244706307 |
| fileinfo16363952933281291636395439041165 |
| fileinfo16363954876685631636395633495907 |
| fileinfo16363956820325691636395828960530 |
| fileinfo16363958775993411636396023539587 |
| fileinfo16363960721797211636396218016687 |
| fileinfo16363962666470321636396412296492 |
| fileinfo16363964609290261636396606698010 |
| fileinfo16363966552145761636396801258927 |
| fileinfo16363968499024131636396995772404 |
| fileinfo16363970443018291636397190253947 |
| fileinfo16363972392255281636397387342140 |
| fileinfo16363974359891361636397581914269 |
| fileinfo16363976306016761636397776594332 |
| fileinfo16363978251895681636397971047223 |

```

*Figure 9: First and second level tables in POFR*

The question that rises now is whether the duration of each archive table is enough to audit something useful. For the client system example of Figure 9, it seems that each archive table triplet has a duration of approx. 10 hours. Each time the archive slot is finished and the archive triplet is produced, the audit process starts from the beginning and some information will be repeated in the next 10 hour slot. What if we want to arrange the data in a more continuous fashion, without any redundancy? This would be the **third level of temporal analysis** of POFR tables. It can be produced by the [mergearchive.pl](#) script.

Note that unlike the second-level temporal assembly script (mergetables.pl), the mergearchive.pl is not called by the system automatically. You will need to do that, interactively or as part of cron/batch jobs. If you consider necessary to merge all the archived tables into one contiguous third level table triad, all you have to do is to lookup the ciduser for that database and pass it as an argument to the script, as shown in Figures 10 and 11. Figure 10 shows the SQL statement that obtains the ciduser. You can then (on another terminal/shell) navigate to the server POFR subdirectory and call the 'mergearchive.pl' script with a single argument (Figure 11). The program will parse the data and produce the following three tables:

```
periodfile20211107104926to20211108175437 |
periodnet20211107104926to20211108175437 |
periodprocess20211107104926to20211108175437
```

that should contain the non redundant version of all ingested data for the archived tables.

```
MariaDB [EA4AFEE2FBE4266484CD00224D37027413572428086]> select ciduser from lhlt.lhlttable where
cid='EA4AFEE2-FBE4-2664-84CD-00224D37027413572428086';
+-----+
| ciduser |
+-----+
| eaec7d766e7fb326c5c3fb965824f3d4 |
+-----+
1 row in set (0.001 sec)
```

**Figure 10: Looking up the ciduser from a client database id (cid)**

```
[root@anaximander server]# ./mergearchive.pl eaec7d766e7fb326c5c3fb965824f3d4
mergearchive.pl status: Detected user eaec7d766e7fb326c5c3fb965824f3d4 in the database...
mergearchive.pl status: Found the filesystem directory for user eaec7d766e7fb326c5c3fb965824f3d4 ...
mergearchive.pl status: User eaec7d766e7fb326c5c3fb965824f3d4 clear of active merge or archive merge threads, c
ntinuing...
mergearchive.pl status: This is producearchive(eaec7d766e7fb326c5c3fb965824f3d4,EA4AFEE2FBE4266484CD00224D37027
413572428086) starting work...
mergearchive.pl status: Starting up, detected /dev/shm/luarmserver/eaec7d766e7fb326c5c3fb965824f3d4/temp dir...
mergearchive.pl status: Detected SELinux in Enforcing mode, good! Thus ensuring that the temp dir has the right
target context...
ValueError: File context for /dev/shm/luarmserver/eaec7d766e7fb326c5c3fb965824f3d4/temp already defined
mergearchive.pl status: The producearchive sub is about to make the periodprocess20211107104926to20211108175437
, periodfile20211107104926to20211108175437 and periodnet20211107104926to20211108175437 period tables.
mergearchive.pl:Cleaning up archive tables for user eaec7d766e7fb326c5c3fb965824f3d4 ...
mergearchive.pl:cleaned up archived process tables: `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archpsinfo20
1107104926to20211107210853` `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archpsinfo20211107211211to202111080
8152` `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archpsinfo20211108073507to20211108175437`
mergearchive.pl:cleaned up archived file tables: `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archfileinfo20
1107104926to20211107210853` `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archfileinfo20211107211211to20211108
73152` `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archfileinfo20211108073507to20211108175437`
mergearchive.pl:cleaned up archived network tables: `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archnetinfo2
21107104926to20211107210853` `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archnetinfo20211107211211to2021110
873152` `EA4AFEE2FBE4266484CD00224D37027413572428086`.`archnetinfo20211108073507to20211108175437`
mergearchive.pl: User eaec7d766e7fb326c5c3fb965824f3d4 process archived tables are: `EA4AFEE2FBE4266484CD00224D
37027413572428086`.`archpsinfo20211107104926to20211107210853`
year:2021, pmonth:11, pday:07, phour:10, pmin:49, psec:26, pmsec:436642
lyear:2021, lmonth:11, lday:08, lhour:17, lmin:54, lsec:37, lmsec:110518
```

**Figure 11: Producing the third level POFR temporal analysis tables**

If you just call the mergearchive.pl script with only the ciduser to merge as an argument, it will merge all the previously archived second level tables into one table triad. If you have logged weeks/months worth of data, the merge process can take a while. If you need to produce tables for a specific/shorter time period and not the entire archive, you can instruct the script to produce them for you using the **--tspec** command line option. The general command line syntax for this is the following:

**mergearchive.pl USER\_TO\_MERGE --tspec=y|n --fromday=dd --frommonth=mm --fromyear=yyyy --fromhour=hh --frommin=mm --fromsec=ss --today=dd --tomonth=mm --toyear=yyyy --tohour=hh --tomin=mm --tosec=ss**

As an example, to produce a contiguous third level assembly archive of all stored events from user bef5f0350b4a3395896f14d2926abcf5 of the 24<sup>th</sup> of June 2022 between 06:00 and 23:00 hours, we would type the following on the POFR server:

**./mergearchive.pl bef5f0350b4a3395896f14d2926abcf5 --tspec=y --fromday=24 --frommonth=06 --fromyear=2022 --fromhour=06 --frommin=00 --fromsec=00 --today=24 --tomonth=06 --toyear=2022 --tohour=23 --tomin=00 --tosec=00**

The script will look at the archive, check if info is available/stored for the specified time period and then, after some state info messages, if the information is found and assembled, it will return output like the one below:

**mergearchive.pl STATUS: Inside the producearchive subroutine: User bef5f0350b4a3395896f14d2926abcf5 process archived tables are: `3337343639335a433332303941394a373618843800`.` archpsinfo20220624043919to20220624130940`  
pyear:2022, pmonth:06, pday:24, phour:04, pmin:39, psec:19, pmsec:787935  
lyear:2022, lmonth:06, lday:24, lhour:23, lmin:09, lsec:42, lmsec:267303**

The output would confirm the time stamps of the produce third level temporal assembly table triad and would produce the following tables for you:

**periodprocess20220624043919to20220624230942  
periodfile20220624043919to20220624230942  
periodnet20220624043919to20220624230942**



## F5) Event representation accuracy and computational load tuning

POFR uses a data reduction technique based on data differencing of the collected procfs snapshots. Details of this technique are provided in [a research paper preprint](#). The practical implication of this technique is that the number of the captured events depends on the procfs sampling frequency. This is particularly true for busy systems with many short lived processes.

The greater the sampling frequency, the greater the completeness of the collected snapshot datasets. A sampling frequency that is too small will inevitably miss events that have extremely short durations. For instance, if a program execution takes 0.1 seconds to complete and the procfs is parsed 3 times per second, it is likely that the program execution won't be recorded by a snapshot. To ensure that the snapshots would intercept such a program, we would need a sampling frequency of at least 10 times per second.

POFR uses a microsecond based sampling delay timer. The scanproc.pl utility has an adjustable delay variable that counts the number of microseconds:

```
my $sdelay=300000;
```

By default, a value of 300000 microseconds (roughly 3 times per second) is chosen. Forensics analysts and DevOps engineers can adjust this frequency on the basis of the event characteristics they try to intercept.

<b>sdelay (microseconds)</b>	<b>Sampling frequency Hz</b>	<b>% single core util</b>	<b>24 h snapshot data size (Gb)</b>	<b>24 h reduced data size (Mb)</b>
300000	3.3	20	8	71
150000	6.6	42	15	72
50000	20	68	31	77

**Table F51: POFR sampling frequency, CPU utilization and data sizes**

However, increasing the sampling frequency has limitations. Reducing the sampling delay to achieve a more frequent procfs sampling increases the computational overhead (CPU) and the amount of data generated by the snapshots. Table F51 relates the sampling frequency to the average single core computational overhead, the produced snapshot data size and the amount of data reduction achieved after the analysis of the data at the server of a busy 12 core development workstation, for a period of 24 hours.

The question of determining the maximum usable sampling frequency depends on how long a single snapshot cycle takes on a specific system. You can determine this by running the performance analysis (perf) tool against the tuning script (tuneperl.pl) when you have a usual workload for the system. All that this script does is to create a single pair of snapshot files under */dev/shm*. This is shown below:

```
[root@test client]# perf stat ./tuneperf.pl
```

**Performance counter stats for './tuneperf.pl':**

<b>86.17 msec task-clock</b>	<b># 0.994 CPUs utilized</b>
<b>6 context-switches</b>	<b># 69.631 /sec</b>
<b>0 cpu-migrations</b>	<b># 0.000 /sec</b>
<b>1,590 page-faults</b>	<b># 18.452 K/sec</b>
<b>306,052,681 cycles</b>	<b># 3.552 GHz</b>
<b>421,528,927 instructions</b>	<b># 1.38 insn per cycle</b>
<b>90,451,302 branches</b>	<b># 1.050 G/sec</b>
<b>1,909,032 branch-misses</b>	<b># 2.11% of all branches</b>

**0.086647744 seconds time elapsed**

**0.048546000 seconds user**

**0.037586000 seconds sys**

Based on the perf tool results, a single snapshot on an Intel(R) Core(TM) i7-10850H CPU system that has a total of 500 running processes takes 86.17msecs. Thus, with reference to table F51, while it is possible to specify a delay of 50000 microseconds, it would not make sense to specify anything lower than 87000 microseconds, if a single sampling cycle takes about the same amount of time to complete.

## **G) Ethical considerations for POFR usage**

POFR is a systems level forensic and monitoring utility that stores and processes information that is potentially sensitive to user privacy. Information such as:

- User commands and program execution
- File and directory names that a user accesses over a period of time
- IP addresses associated with programs that a particular user executes
- Country information of remote IP address endpoints that the system contacts

are potentially sensitive data that the tool stores and processes. POFR does **not** store or processes in any shape or form:

- User file content
- User URLs (browsing history)
- Content/payload of network connection/traffic

As a result, the authors recommend that prior deploying POFR to IT infrastructures, relevant permissions need to be obtained from your organisation's GDPR/Data Protection officers to ensure that the use of the utility adheres to all applicable legal requirements related to the storage and processing of personal information.