ELF Malware Analysis 101: Part 3 - Advanced Analysis

February 17, 2021

Written by Avigayil Mechtinger - 17 February 2021

[Get Free Account](#)

Join Now

Top Blogs

Getting Caught Up to Speed

So far in this series we have profiled the ELF threat landscape and covered the most common intrusion vectors seen in Linux systems. We also pursued initial ELF analysis with an emphasis on static analysis. We learned about the different artifacts and components that are relevant for initial analysis and how they can help us gather immediate insights about a file. **In Part 3, we will take the next step and dynamically analyze ELF malware**.

While static analysis is performed without execution, dynamic analysis gives us an understanding of how a file interacts with the operating system at runtime. Insights about file behavior help us better assess the potential impact of the malware and gather additional indicators that can help us paint the bigger picture (C&C for instance). Dynamic analysis also allows us to collect further Tactics, Techniques, and Procedures (TTPs) that can be attributed to specific malicious tools and threat actors.

Static and dynamic analysis are complementary. The information gathered during initial analysis will accelerate the dynamic analysis process.

Agenda

After reading this article you will be able to understand which insights can be extracted during ELF runtime and what tools can help you do so.

The following subjects will be covered in this article:

1. Linux Processes
2. ELF Syscalls
3. Persistence Methods
4. Network Sniffing
5. Sandboxes

After covering our dynamic analysis toolset, we will put them to use by practicing on a real sample found in the wild.

Analysis Environment Preparations

Before we get started, let's prep your Linux VM (virtual machine). If you don't have a Linux VM, follow this guide to install one.

Needless to say, running malware should only be done in isolated environments such as VMs. Even when using a VM the malware can cause harm, not only to your system but also to other machines over the internet. For example, running a worm can spread it over the network (see New Golang Worm Drops XMRig Miner on Servers). Running ransomware or stealers can encrypt and/or collect information from your VM, including files located on the shared folders with your host.

Make sure that before you run a malware your virtual environment meets the following criteria:

1. **Your machine host name and user name are generic**. Run **uname -n** to see the host name of your machine, and run **getent passwd {1000..60000}** to list all human usernames on the machine.

2. **Your IP is not easily trackable**. Blurring your public IP can be done in several ways:
   1. **Basic**: Use a <u>VPN service</u> on your host machine during dynamic analysis. The traffic from your VM should be tunneled through your host. Verify your public IP from within the VM by running a command such as: curl 'https://api.ipify.org'. There are various free VPN programs you can use such as <u>OpenVPN</u>.
   2. **Advanced:** Use a second VM as a router that tunnels traffic, via Tor for instance. Establishing that, you can route the entire network from your malware analysis VM through the router VM.

   Note that you can always disconnect your machine from the internet and start the dynamic analysis with no network connection as a first step.

1. **Your machine is clean from private information.** Make sure that you don't have any passwords, API keys, etc. written on the machine.
2. **Disable shared folder and shared clipboard before running the malware.**

Setting Up SSH Connection

Use SSH to transfer files from your machine (host) to your VM (guest). Follow these steps to establish connection:

1. Install <u>OpenSSH</u> on your VM:
   sudo apt-get update
   sudo apt-get install -y openssh-server
2. Shut down your VM.
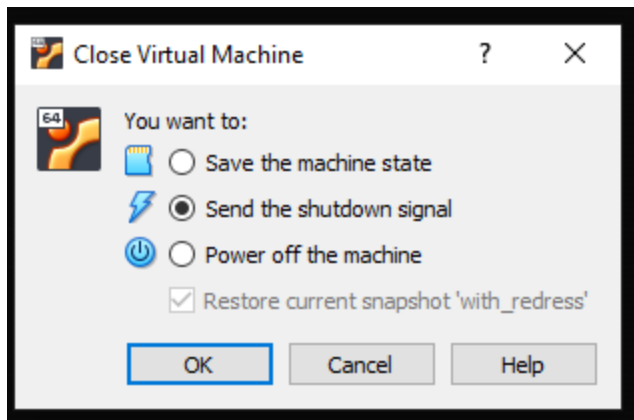
1.
   1. Go to Settings > Network.
      1. Adapter 1 should be set to NAT:
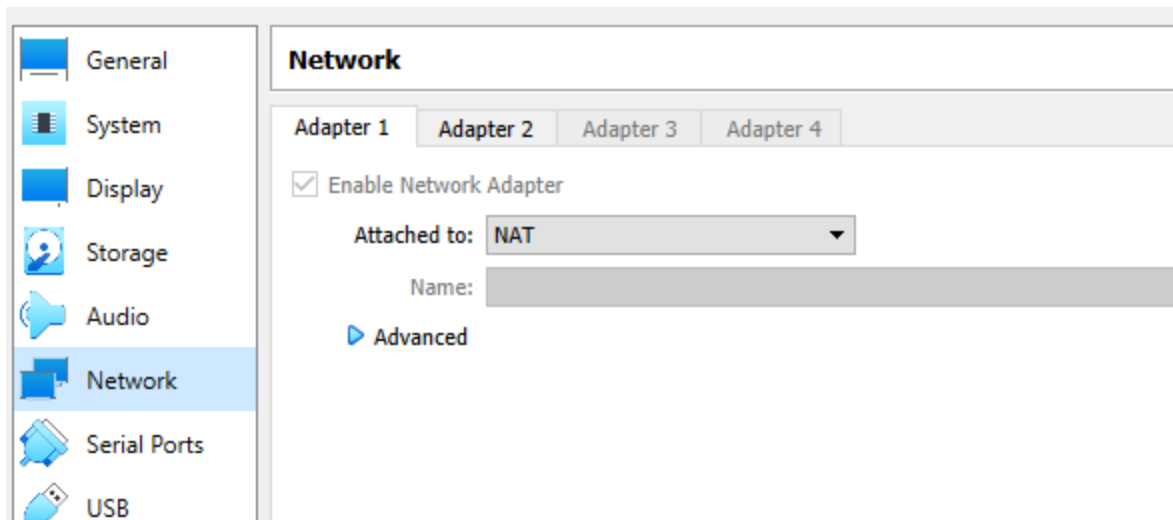
1.
    1. Add a second adapter: Choose "Host-only Adapter" and apply changes.
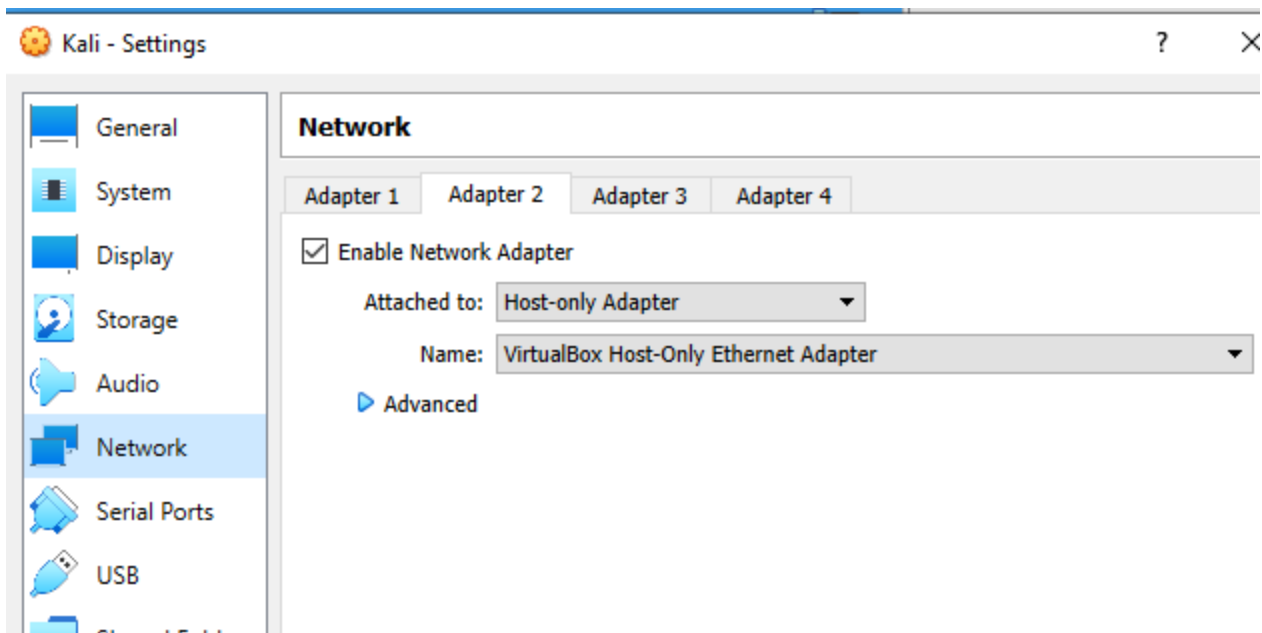
1. Go to Settings > System > Motherboard and make sure these entries are enabled:
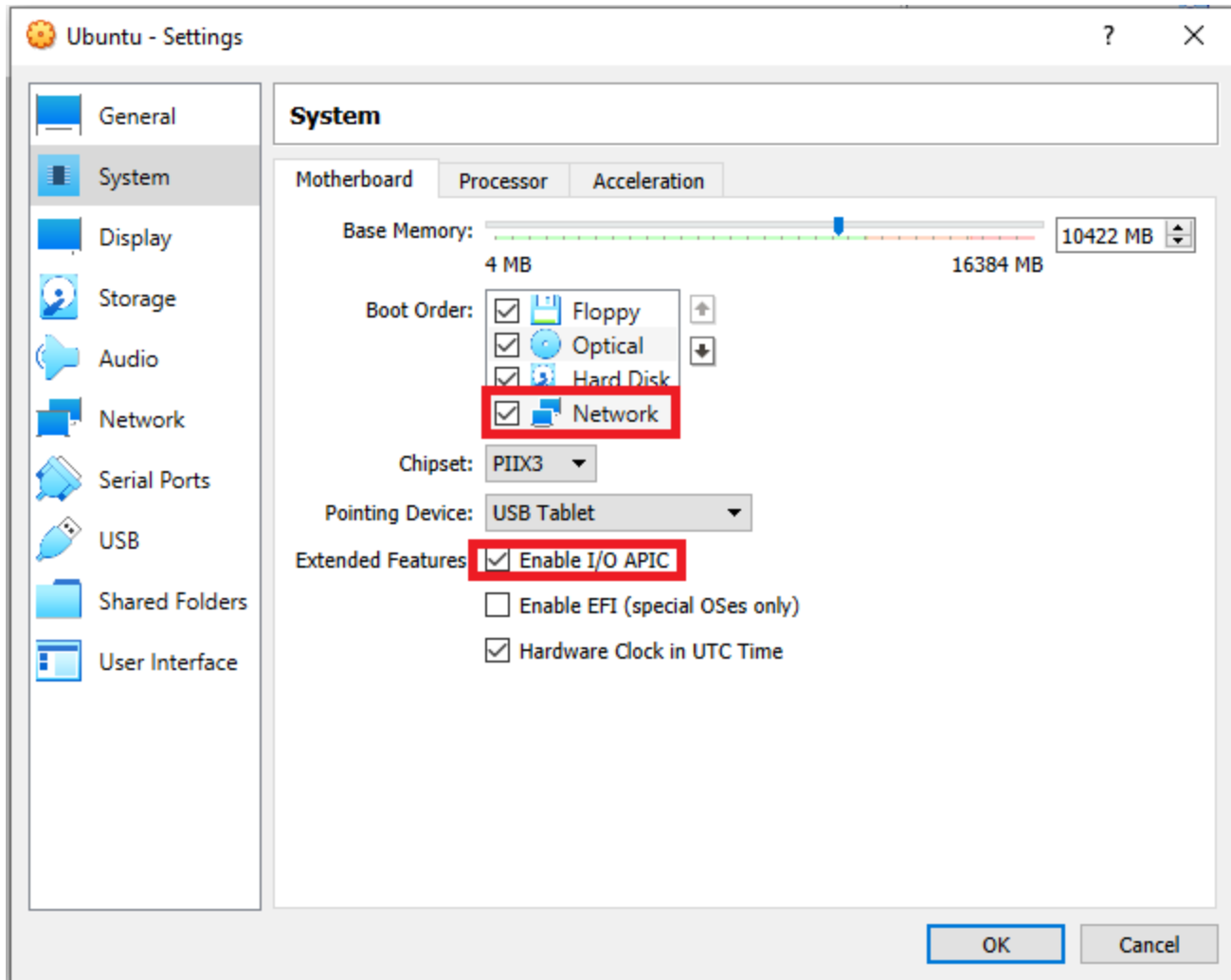
1. Start your VM and get the LAN IP address of the VM instance. Run **ip addr show**.


```
root@ubuntu:~# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defau
lt qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000
    link/ether 08:00:27:6f:33:c1 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 85579sec preferred_lft 85579sec
    inet6 fe80::4484:a015:46f1:35f6/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000
    link/ether 08:00:27:33:74:56 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.106/24 brd 192.168.56.255 scope global dynamic noprefixrout
e enp0s8
       valid_lft 357sec preferred_lft 357sec
    inet6 fe80::8f24:510:9541:d025/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

1.
   1. Make sure the OpenSSH service is active on the VM by running: **sudo service ssh status**.

1. You should now be able to connect from your host to the guest machine via SSH. Using **scp** command you can copy files and directories from your host machine to the VM. Run on your host:
   scp -r \my\host\path VM-username@/path/to/whereyouwant/thefile

Linux Processes

Every instance of a running program on the system is a process. Each process has its unique process ID. You can see all of your processes by running **ps aux.** The ps command displays information about the process that ran at the exact time you ran the ps command. To see an ongoing repetitive output use top.

```
root@ubuntu:~# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 160164  9516 ?        Ss   16:13   0:03 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    16:13   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   16:13   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   16:13   0:00 [rcu_par_gp]
root           6  0.0  0.0      0     0 ?        I<   16:13   0:00 [kworker/0:0H-kb]
root           9  0.0  0.0      0     0 ?        I<   16:13   0:00 [mm_percpu_wq]
root          10  0.0  0.0      0     0 ?        S    16:13   0:00 [ksoftirqd/0]
root          11  0.0  0.0      0     0 ?        I    16:13   0:00 [rcu_sched]
root          12  0.0  0.0      0     0 ?        S    16:13   0:00 [migration/0]
root          13  0.0  0.0      0     0 ?        S    16:13   0:00 [idle_inject/0]
root          14  0.0  0.0      0     0 ?        S    16:13   0:00 [cpuhp/0]
root          15  0.0  0.0      0     0 ?        S    16:13   0:00 [kdevtmpfs]
root          16  0.0  0.0      0     0 ?        I<   16:13   0:00 [netns]
root          17  0.0  0.0      0     0 ?        S    16:13   0:00 [rcu_tasks_kthre]
root          18  0.0  0.0      0     0 ?        S    16:13   0:00 [kauditd]
root          19  0.0  0.0      0     0 ?        S    16:13   0:00 [khungtaskd]
root          20  0.0  0.0      0     0 ?        S    16:13   0:00 [oom_reaper]
root          21  0.0  0.0      0     0 ?        I<   16:13   0:00 [writeback]
root          22  0.0  0.0      0     0 ?        S    16:13   0:00 [kcompactd0]
root          23  0.0  0.0      0     0 ?        SN   16:13   0:00 [ksmd]
root          24  0.0  0.0      0     0 ?        SN   16:13   0:00 [khugepaged]
root         116  0.0  0.0      0     0 ?        I<   16:13   0:00 [kintegrityd]
root         117  0.0  0.0      0     0 ?        I<   16:13   0:00 [kblockd]
root         118  0.0  0.0      0     0 ?        I<   16:13   0:00 [blkcg_punt_bio]
root         119  0.0  0.0      0     0 ?        I<   16:13   0:00 [tpm_dev_wq]
```

Figure 1: **ps aux** output

The proc Filesystem

The "proc" filesystem is a pseudo-filesystem provided by the Linux kernel. It provides an interface to kernel data structures which includes information about all currently running processes. It will commonly be mounted under the /proc directory. Each process has its own

directory under /proc and the directory name is the process ID.

To get a better understanding of the proc directory, open two terminals on your Linux VM and run **ping 8.8.8.8** on one of them. On the second terminal, run **pidof ping** to retrieve ping's process ID.

Let's review the process directory by browsing the /proc/PID directory and running **ls**. You can see that it has different directories and files. All process directories share the same structure, filenames and directories. Figure 2 describes the flow above.

```
root@ubntu:~# pidof ping
14568
root@ubntu:~# cd /proc/14568
root@ubntu:/proc/14568# ls
arch_status      environ      mountinfo       personality    statm
attr             exe          mounts          projid_map     status
autogroup        fd           mountstats      root           syscall
auxv             fdinfo       net             sched          task
cgroup           gid_map      ns              schedstat      timers
clear_refs       io           numa_maps       sessionid      timerslack_ns
cmdline          limits       oom_adj         setgroups      uid_map
comm             loginuid     oom_score       smaps          wchan
coredump_filter  map_files    oom_score_adj   smaps_rollup
cpuset           maps         pagemap         stack
cwd              mem          patch_state     stat
```

Figure 2: Retrieving ping's process ID and the process directory content

The following are some interesting files that will be present under every process directory:

cmdline – command line arguments that ran the file.

status – process status in human readable form.

maps – memory maps regions and their access permissions.

exe – symbolic link containing the pathname of the executed command. Attempting to open it will open the executable. Try running **sha256sum exe** and sha256sum on the ping file: **sha256sum $(which ping)**. You will see they match.

After you kill the ping process, you will see the /proc/PID directory no longer exists.

Browse here for more information about the proc file system.

Process Tree

The process tree structure can give you insights about what is running on a machine before diving into specific processes.

A single executable can create more than one process on the machine. Let's emphasize that by using the **pstree** command to view the running processes as a tree. The following are some examples of what the process tree will look like for each scenario:

1. **Other process creation**: A process can call other processes. Let's compile a simple program that runs a ping command via bash and name it **ping-google-dns**:

*Figure 3: **ping-google-dns** source code*

Let's run the program and on another terminal run **pstree | grep ping-google-dns**:

*Figure 4: Process tree created by **ping-google-dns***

We ran the **ping-google-dns** program from a terminal (**bash** process) which called **sh** binary (a command language interpreter) that called the **ping** binary.

1. **Forks**: fork() creates a new, duplicate process of the called process. The new process is called a child process and it will have a different process ID than its parent. Let's re-compile the **ping-google-dns** source code, this time with the addition of a fork() call.

```
#include <unistd.h>
#include <stdlib.h>

void main() {
    fork();
    system("ping 8.8.8.8");
}
```

*Figure 5: **ping-google-dns** source code*

Let's run the program and on another terminal run ***pstree | grep ping-google-dns***.

Figure 6 shows what the process tree looks like with one fork call. You will see in the screenshot below that the same program has two process IDs.

```
root@ubntu:~# pstree | grep ping-google-dns
        |           |               |-bash---ping-google-dns-+-ping-google-dns---sh---ping
root@ubntu:~# pidof ping-google-dns
15223 15222
```

*Figure 6: Process tree created by **ping-google-dns***

1. **Threads**: Threads provide multiple executions within the program. A process thread will not create a new process ID. Let's compile this code that runs three threads as **print-something** (the compiled file can be downloaded from <u>here</u>).

```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
pthread_t tid[3];


void* print_something(void *arg){
printf("Threading..\n");
sleep(120);
return NULL;
}

void main()
{
int i=0;
int err;
 while(i<3)
{
  pthread_create(&(tid[i]), NULL,&print_something,NULL);
  i++;
}
sleep(120);
return;
}
```

*Figure 7: **print-something** source code*

Now let's run the program and run ***pstree | grep print-something*** on another terminal. Figure 8 emphasizes what the process tree of a program that runs with three threads will look like.

```
root@ubntu:~# pstree | grep print-something
        |       |                     |-bash---sudo---su---bash---print-something---3*[{print-something}]
root@ubntu:~# pidof print-something
15176
```

*Figure 8: Process tree created by **print-something***

Process Monitoring Tools

Sysinternals Suite provides convenient GUIs for monitoring all processes that are running on a Windows machine (among other things). Microsoft's ProcMon-for-Linux based on ProcMon and Intezer's Linux Expl0rer inspired by ProcExp, are intuitive solutions for Linux

process tracking.

System Calls

Only the kernel can perform changes outside of the process's own memory space. The process must ask the kernel to perform tasks such as creating files or writing output. This is where system calls come into play.

Syscalls (system calls) are the interface used by the application to request services from the kernel. Syscalls are usually invoked via glibc wrappers and not directly to the kernel because of portability. The low level syscalls differ between architectures which is why glibc handles these differences instead of the developer.

Syscalls are an interface that the malware must pass through in order to cause actual harm to the system. Analyzing syscalls can help us understand how the file interacts with the system and how it operates behind the scenes.

strace is a powerful tool to trace a file's system calls. Run ***strace whoami*** on your Linux VM and take a look at the output. Each row in the strace output is a syscall, and the first syscall will be ***execve*** which stands for execute program. Each system call has a return value that varies between calls. It can be a file descriptor (integer) or 0 on success, -1 on error and more.

These are some interesting syscalls we will look for by analyzing the strace output:

open/openat – open and possibly create a file.

read – read from a file descriptor.

access – check user's permissions for a file.

write – write to a file descriptor.

mkdir/mkdirat – make directories.

connect – initiate a connection on a socket.

socket – create an endpoint for communication.

execve – execute program.

Let's try it ourselves. Run the trace-me file on your VM using strace. Use the -o flag to save the command output: ***strace -o out.txt ./trace-me***:

```
root@ubuntu:~# strace -o out.txt ./trace-me
What just happened??
```

*Figure 9: **trace-me** output*

Now, let's read the strace output to see what happened on the system. Run ***cat out.txt***:


```
mkdir("/tmp/.tomato", 0700)                = 0
brk(NULL)                                  = 0x55eb8155e000
brk(0x55eb8157f000)                        = 0x55eb8157f000
openat(AT_FDCWD, "/tmp/.tomato/answer.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
write(3, "I Was created!!!!", 17)          = 17
close(3)                                   = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "What just happened??\n", 21What just happened??
)    = 21
exit_group(0)                              = ?
+++ exited with 0 +++
```

*Figure 10: **strace** output*

You will see the file creates the **.tomato** directory under **tmp** directory, and creates a file **answer.txt** inside this directory. If you look carefully on the flow, you can see the return value of **openat** for the anser.txt file is 3, which is the file descriptor. Then, **write** syscall**, "I was created!!"** uses the file descriptor 3 as an input. This means that this text is written to the answer.txt.

Tips:

1. In most cases the syscalls output will be much bigger than the last example. It is recommended to always save the strace output to a text (***strace -o out.txt ./file)***. Make sure that you have a convenient text editor such as Sublime Text that will help you analyze the large text file.
2. To gain full visibility on all syscalls of the processes created by the file, including forks, you can run **strace** with the fork flag -f (***strace -f ./file).***
3. The default strings length is 32. Use the -v (verbose) and -s (string size) flags to see more characters (**strace -v -s 150 ./file**).
4. To reduce noise, you can choose which syscalls will be presented on strace output using the -e flag. For example, if you are searching for network-related calls together with read and write calls, you can run **strace -e network,read,write ./file**.

Learn more about syscalls

Persistence

Once a malware finds its way into a compromised system, it will often attempt to achieve persistence in order to survive reboot. Another reason why malware developers add persistence capabilities is to harden removal efforts, which is also attempted by other malware families in order to gain a foothold on already compromised systems. This is more common among CoinMiners, which will search for other known Miners on a compromised machine and attempt to kill them, to be the only CoinMiner running on the system and ultimately win the machine's resources. See <u>War of Linux Cryptocurrency Miners: A Battle for Resources</u>.

Detecting persistence methods is important for understanding how to respond to malware on a compromised machine. Killing the process is not always enough to mitigate the threat, the persistence methods should be cleared as well.
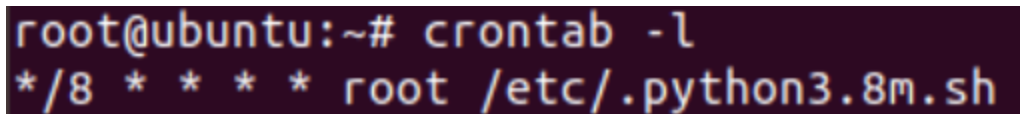
These are the most common persistence methods used by Linux malware:

1. <u>**Cron**</u> – Malware will create scheduled tasks to run periodically on a system using cron jobs. <u>crontab</u> and <u>anacrontab</u> are the configuration files used for <u>cron</u> and <u>anacron</u> services which are in charge of executing scheduled tasks. The malware will write registries to the configuration files which are located under: /etc/crontab, /var/spool/cron/, /etc/cron.d/, /etc/anacrontab, /var/spool/anacron/. The following <u>XMRig Miner dropper</u> uses crontab as one of its persistence methods. Figure 11 emphasizes what a crontab registry looks like:

```
root@ubuntu:~# crontab -l
*/8 * * * * root /etc/.python3.8m.sh
```

Figure 11: crontab registry written by a malware

Note: Running crontab -l will list the existing cron jobs per user. Using root privileges, you can either run ***crontab -l -u <user>*** or ***cat /var/spool/cron/crontabs/<user>*** for specific users. To view the cron jobs for all users, run ***cd /var/spool/cron/crontabs/ && grep . ****

1. <u>**Services**</u> – Linux has initialization scripts that are used to start services on system boot. The program in charge of starting the rest of the system will run as PID 1 (you can explore your /proc/1 directory). Malware will often attempt to gain persistence by creating a service which will run by the init program on boot. The init program varies between Linux distributions and versions, however, systemd is most common these days. rc.d and init.d are older init services which are still used in certain Linux distributions. <u>Learn more about why init.d was replaced by systemd</u>
The service will commonly be found under the following paths: /etc/systemd/system/, ~/.config/systemd, /etc/rc.d/, /etc/init.d/.

   <u>IPStorm</u> Linux version is an example of a malware that gained persistence by creating a service under /etc/systemd/system/storm.service. Figure 12 emphasizes the structure of a systemd service file.

```
[Unit]
Description=storm
Requires=
After=

[Service]
PIDFile=/var/run/storm.pid
ExecStartPre=/bin/rm -f /var/run/storm.pid
ExecStart=/usr/bin/storm
Restart=on-failure

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

Figure 12: storm.service content

1. **Event Triggered Executions** – Once a user logs in or a new shell session is created, Linux will automatically launch executables by using configuration files. These configuration files are bash scripts which can be edited to trigger a malware. Files locations: /etc/profile.d, /etc/profile, /etc/bash.bashrc, ~/.bashrc, ~/.bash_profile, ~/.bash_login, ~/.profile. Linux Rabbit is a malware that uses the .bashrc to set up persistence.

1. **Graphical Desktop Autostarts** – Similar to event triggered executions, once a user logs in via a graphical desktop environment such as GNOME or KDE, Linux will automatically search for desktop entries to execute applications on startup. A generic location for desktop entries is under ~/.config/autostart/. Other possible locations include: ~/.kde/Autostart,~/.kde/share/autostart, ~/.kde4/Autostart, /usr/share/autostart/, /etc/xdg/autostart/. The autostart service will have a .desktop suffix. This persistence method is relevant for malware that targets endpoints. ElectroRAT's Linux version gains persistence by creating a desktop entry under ~/.config/autostart/mdworker.desktop.

1. **Loadable Kernel Modules (LKM)** – The Linux kernel is designed to allow loading of external pieces of code called kernel modules. Kernel modules are automatically loaded on system boot. Because the Linux kernel runs as root, and the modules can be loaded dynamically (using modprobe or insmod) with no need to reboot the system, this method is commonly used for rootkits as well. Malware that attempts to gain persistence via this method will create malicious kernel modules that will be loaded to the kernel and run on boot. The location of the kernel modules is under: /lib/modules/$(uname -r). By running **lsmod** you can see what kernel modules are currently loaded.

1. **Hijack Execution Flow** – Dynamically linked binaries use shared libraries during runtime (in the previous article we explained the differences between dynamically and statically linked files). These libraries are loaded by the dynamic linker which searches for libraries on absolute paths and common directories. LD_PRELOAD is an optional environmental variable containing paths to shared libraries or objects. The dynamic linker will load the libraries in LD_PRELOAD before loading any other shared library (including libc). Malware developers can set LD_PRELOAD to point to malicious libraries. Once a dynamically linked binary is executed on a compromised machine, it will load the malicious library too. HiddenWasp is a malware that uses LD_PRELOAD for persistence.

To summarize this section, there are different startup locations which can be used as persistence methods for malware. However, the most commonly used methods are services and cron jobs.

Tip: A convenient way to check if a malware attempts to gain persistence is by analyzing the syscalls (strace output) and searching for known methods. For example, you can run ***cat strace_output.txt | grep cron*** to search for any interaction with the cron process.

Network Sniffing

So far we covered how to monitor a file's activity internally on the system. You can understand if a malware interacts with a C&C or external services by analyzing network-related syscalls such as socket and connect. However, syscalls are not the way to go for network monitoring. You should use a packet sniffing and network monitoring tool to analyze the traffic. The most popular tools for this purpose are tcpdump and Wireshark, both based on libpcap. tcpdump is a CLI tool that should be preinstalled on commonly used Linux distributions. Wireshark has similar functionalities to tcpdump and provides a convenient GUI.

To install Wireshark on your VM run the following commands:

sudo add-apt-repository ppa:wireshark-dev/stable
sudo apt install -y wireshark
sudo wireshark

Tip: Make sure to run Wireshark before running the file you are analyzing so that you won't miss any packets related to the malware.

Sandboxes

Sandboxes can accelerate analysis by providing context about the file's behavior on a machine without the hassle of opening a VM, running the malware and relevant tools. They will not always replace a hands-on, deep dive analysis of a VM, but sandboxes help you pinpoint interesting behaviors while saving you time.

Here are some relevant tools:

Real Life Example

Let's practice your dynamic analysis skills! We will begin with initial analysis to gather insights and then proceed to dynamic analysis. Our mission is 1. Understand how the file behaves and 2. See how dynamic analysis tools come in handy.

Our test sample? This ELF malware found in the wild.

A quick side note. Uploading this file to Intezer Analyze classifies it as Reekoobe, based on code reused from previous Rekoobe samples. We will analyze the file manually for practice anyway.
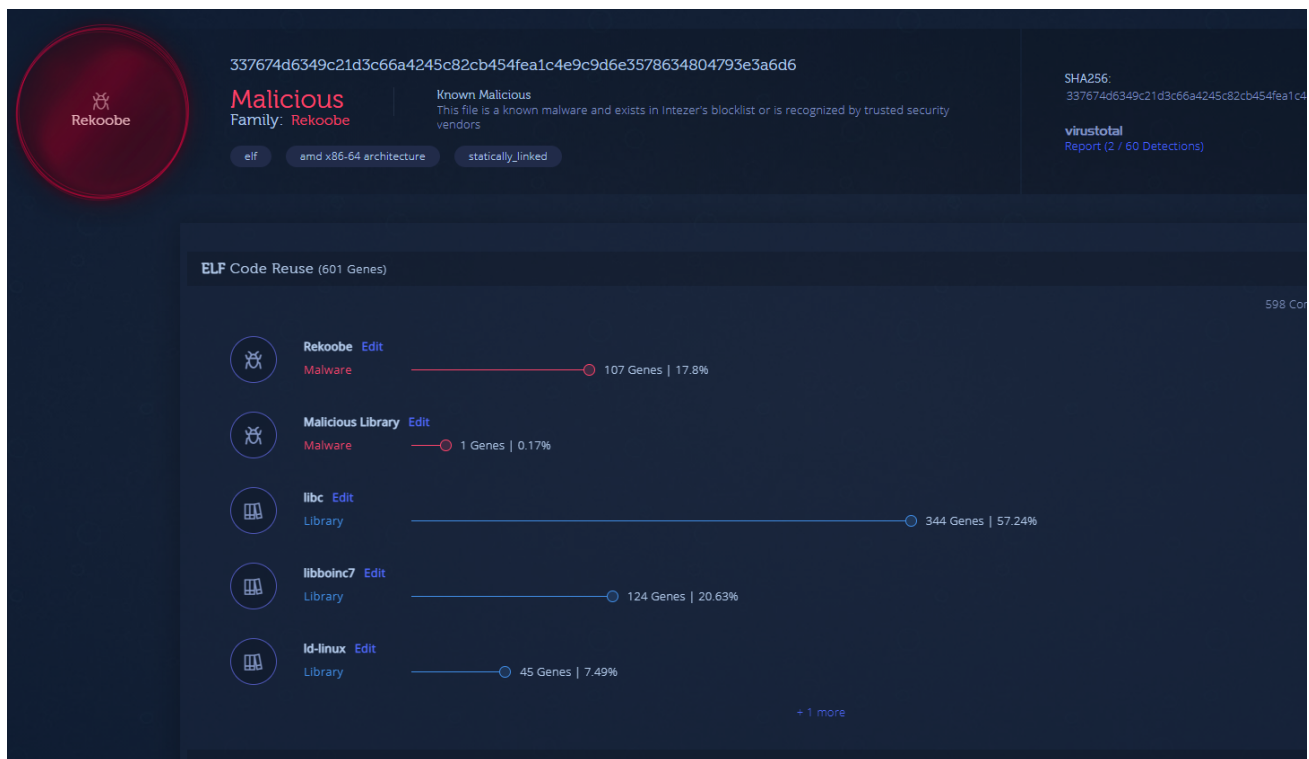
Step one, download/copy the <u>sample</u> to your VM. Make sure:

1. Your environment is prepared based on the Analysis Environment Preparation section.
2. You have a clean snapshot of your VM with the relevant tools. In this example we will be using Wireshark and Sublime Text editor. Other tools should be preinstalled on the machine.

Let's begin with static analysis and try to read the file's symbols to see if there are any human readable function names. Run *readelf -s sample.* There is no output for the command, which means the file has been stripped.

Let's read the file's program headers by running *readelf -l sample*. The following image shows the command's output:

```
Elf file type is EXEC (Executable file)
Entry point 0x4003c0
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000cb8e1 0x00000000000cb8e1  R E    0x200000
  LOAD           0x00000000000cb8e8 0x00000000006cb8e8 0x00000000006cb8e8
                 0x00000000000017e8 0x0000000000007000  RW     0x200000
  NOTE           0x0000000000000158 0x0000000000400158 0x0000000000400158
                 0x0000000000000044 0x0000000000000044  R      0x4
  TLS            0x00000000000cb8e8 0x00000000006cb8e8 0x00000000006cb8e8
                 0x0000000000000028 0x0000000000000060  R      0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x8

 Section to Segment mapping:
  Segment Sections...
   00      .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text __li
bc_subfreeres __libc_atexit __libc_thread_subfreeres .stapsdt.base .eh_fra
   01      .tdata .ctors .dtors .jcr .data.rel.ro .got .got.plt .data .bss
   02      .note.ABI-tag .note.gnu.build-id
   03      .tdata .tbss
   04
```

Figure 13: Sample's program headers

We can tell the file is statically linked because there is no dynamic symbols table or dynamic program header.

Let's run the strings command on the file. Because the file is statically linked you should expect to see a large number of strings related to libc. Figure 14 is a snippet of the strings commands output:

```
/etc/init.d
/proc/self/exe
/usr/bin/likemae
open error
#!/bin/bash
#chkconfig: 2345 80 90
#description:likemae
chmod +x /etc/init.d/likemae
chkconfig --add likemae
chkconfig likemae on
service likemae
systemdctl
/lib/systemd/system
[Unit]
[Service]
Restart=always
Type=forking
[Install]
WantedBy=multi-user.target
systemctl enable likemae
systemctl daemon-reload
systemctl start likemae
/usr/lib/systemd
```

*Figure 14: Snippet from **strings** output*

The strings snippet is a big indicator that the file will attempt to create a service to gain persistence. "systemctl enable likemae" and other strings can help us understand that the service name will be **likemae**. Another interesting string is c[.]linux-hosts[.]com which could be the C&C.

Now that we have gathered enough information in the initial analysis step, let's proceed to dynamic analysis.

First, prepare a running program of Wireshark and then run the sample with strace: **sudo strace -f -o out.txt ./sample**. We will analyze this malware as if it ran with a privileged user to see its full capabilities.

Open the out.txt with a convenient text editor such as Sublime Text. The following snippet shows the beginning of the strace output:

```
access("/lib/systemd/system", F_OK)        = 0
access("/lib/systemd/system/likemae.service", F_OK) = -1 ENOENT (No such file or directory)
readlink("/proc/self/exe", "/root/sample", 1024) = 12
rename("/root/sample", "/usr/bin/likemae") = 0
open("/lib/systemd/system/likemae.service", O_RDWR|O_CREAT|O_TRUNC, 0666) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2728ba5000
write(3, "[Unit]\nDescription=likemae Secur"..., 199) = 199
close(3)                                    = 0
```

Figure 15: Snippet from strace output

Analyzing syscalls from the snippet, it's clear that this malware checks for the existence of likemae.service with the **access** syscall. The return value is -1 because the service doesn't yet exist on the machine. The malware renames the file and locates it under **/usr/bin/likemae** using the **rename** syscall. Next, using the **open** syscall, the malware creates **likemae.service** and writes to it.

Let's view the service's full content:

```
root@ubuntu:~# cat /lib/systemd/system/likemae.service
[Unit]
Description=likemae Secure Shell check
After=network.target auditd.service

[Service]
Restart=always
Type=forking
ExecStart=/bin/bash -c /usr/bin/likemae

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

*Figure 16: **likemae.service** content*

You can also use **grep** to find interesting syscalls. Run ***cat out.txt | grep exe*** to see all executions created by the malware.

```
root@ubuntu:~# cat out.txt | grep exe
execve("./sample", ["./sample"], 0x7ffd2bd4c1f8 /* 26 vars */) = 0
readlink("/proc/self/exe", "/root/sample", 1024) = 12
[pid 15038] execve("/bin/sh", ["sh", "-c", "systemctl enable likemae"], 0x7ffc99129238 /* 26 vars */) = 0
[pid 15039] execve("/bin/systemctl", ["systemctl", "enable", "likemae"], 0x5610a9af7bc0 /* 27 vars */) = 0
[pid 15059] execve("/bin/sh", ["sh", "-c", "systemctl daemon-reload"], 0x7ffc99129238 /* 26 vars */) = 0
[pid 15060] execve("/bin/systemctl", ["systemctl", "daemon-reload"], 0x55aa65bccb88 /* 27 vars */) = 0
[pid 15079] execve("/bin/sh", ["sh", "-c", "systemctl start likemae"], 0x7ffc99129238 /* 26 vars */) = 0
[pid 15080] execve("/bin/systemctl", ["systemctl", "start", "likemae"], 0x557c61ccbbc0 /* 27 vars */) = 0
```

The malware made sure that the likemae service would run. Check that the service is indeed active on your machine by running *systemctl | grep likemae*.

*Figure 17: **likemae** service is active*

So far, you know the malware created persistence on the machine using the service creation method and copied itself to the **/user/bin/likmae**. You can also tell the process that we ran has exited after establishing persistence. Let's check if a process called likemae is currently running on our system. Run **pstree -p | grep likemae** (-p flag to retrieve the process ID).

*Figure 18: **likemae** process is running with PID 15083*

If we run *cat /proc/1503/cmdline*, we will see that is **/bin/bash**. This is because the process was triggered by the service.

Now that the service exists and the malware established persistence, you can expect the new malware process to perform different actions on the system. Attach strace to the likemaes's PID by running **strace -p 15083**.

Analyzing syscalls we can understand that the process attempts to connect to the c[.]linux-hosts[.]com host and then sleeps for 30 seconds (see nanosleep syscall). The following snippet shows the connection attempt loop. You can see the syscalls that are made in order to connect to the C&C.

```
socket(AF_INET, SOCK_DGRAM, IPPROTO_IP) = 28
openat(AT_FDCWD, "/etc/hosts", O_RDONLY|O_CLOEXEC) = 29
fstat(29, {st_mode=S_IFREG|0644, st_size=232, ...}) = 0
read(29, "127.0.0.1\tlocalhost\n127.0.1.1\tad"..., 4096) = 232
read(29, "", 4096)                       = 0
close(29)                                = 0
stat("/etc/resolv.conf", {st_mode=S_IFREG|0644, st_size=20, ...}) = 0
socket(AF_INET, SOCK_DGRAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_IP) = 29
connect(29, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("127.0.0.1")}, 16) = 0
gettimeofday({tv_sec=1612712822, tv_usec=32387}, NULL) = 0
poll([{fd=29, events=POLLOUT}], 1, 0)    = 1 ([{fd=29, revents=POLLOUT}])
sendto(29, "F\\\1\0\0\1\0\0\0\0\0\0\1c\vlinux-hosts\3com\0\0"..., 35, MSG_NOSIGNAL, NULL, 0) = 35
poll([{fd=29, events=POLLIN}], 1, 5000) = 0 (Timeout)
gettimeofday({tv_sec=1612712827, tv_usec=38220}, NULL) = 0
poll([{fd=29, events=POLLOUT}], 1, 0)    = 1 ([{fd=29, revents=POLLOUT}])
sendto(29, "F\\\1\0\0\1\0\0\0\0\0\0\1c\vlinux-hosts\3com\0\0"..., 35, MSG_NOSIGNAL, NULL, 0) = 35
poll([{fd=29, events=POLLIN}], 1, 5000) = 0 (Timeout)
close(29)                                = 0
rt_sigprocmask(SIG_BLOCK, [HUP USR1 USR2 PIPE ALRM CHLD TSTP URG VTALRM PROF WINCH IO], [], 8) = 0
uname({sysname="Linux", nodename="              ...}) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
write(2, "gethostbyname: Success\n", 23) = -1 EDESTADDRREQ (Destination address required)
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 29
openat(AT_FDCWD, "/etc/hosts", O_RDONLY|O_CLOEXEC) = 30
fstat(30, {st_mode=S_IFREG|0644, st_size=232, ...}) = 0
read(30, "127.0.0.1\tlocalhost\n127.0.1.1\tad"..., 4096) = 232
read(30, "", 4096)                       = 0
close(30)                                = 0
stat("/etc/resolv.conf", {st_mode=S_IFREG|0644, st_size=20, ...}) = 0
socket(AF_INET, SOCK_DGRAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_IP) = 30
connect(30, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("127.0.0.1")}, 16) = 0
gettimeofday({tv_sec=1612712832, tv_usec=46660}, NULL) = 0
poll([{fd=30, events=POLLOUT}], 1, 0)    = 1 ([{fd=30, revents=POLLOUT}])
sendto(30, "\260\16\1\0\0\1\0\0\0\0\0\0\1c\vlinux-hosts\3com\0\0"..., 35, MSG_NOSIGNAL, NULL, 0) = 35
poll([{fd=30, events=POLLIN}], 1, 5000) = 0 (Timeout)
gettimeofday({tv_sec=1612712837, tv_usec=52107}, NULL) = 0
poll([{fd=30, events=POLLOUT}], 1, 0)    = 1 ([{fd=30, revents=POLLOUT}])
sendto(30, "\260\16\1\0\0\1\0\0\0\0\0\0\1c\vlinux-hosts\3com\0\0"..., 35, MSG_NOSIGNAL, NULL, 0) = 35
poll([{fd=30, events=POLLIN}], 1, 5000) = 0 (Timeout)
close(30)                                = 0
rt_sigprocmask(SIG_BLOCK, [HUP USR1 USR2 PIPE ALRM CHLD TSTP URG VTALRM PROF WINCH IO], [], 8) = 0
uname({sysname="Linux", nodename="              ", ...}) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({tv_sec=30, tv_nsec=0}, strace: Process 21602 detached
```

Figure 19: Connection attempt loop

You can view the network connection attempts in Wireshark as well.

```
    8 34.047553118  10.0.2.15        192.168.0.1       DNS   88 Standard query 0xe3f0 A c.linux-hosts.com OPT
    9 34.136357171  192.168.0.1      10.0.2.15         DNS  104 Standard query response 0xe3f0 A c.linux-hosts.com A 128.199.171.76 OPT
   10 34.136801807  10.0.2.15        128.199.171.76    UDP  113 35780 → 10010 Len=71
   11 39.286418513  PcsCompu_6f:33:c1  RealtekU_12:35:02  ARP   42 Who has 10.0.2.2? Tell 10.0.2.15
   12 39.287558944  RealtekU_12:35:02  PcsCompu_6f:33:c1  ARP   60 10.0.2.2 is at 52:54:00:12:35:02
   13 39.287903095  10.0.2.15        128.199.171.76    TCP   74 55220 → 54479 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=352
   14 40.310262427  10.0.2.15        128.199.171.76    TCP   74 [TCP Retransmission] 55220 → 54479 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
   15 42.326154494  10.0.2.15        128.199.171.76    TCP   74 [TCP Retransmission] 55220 → 54479 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
   16 42.877108439  128.199.171.76   10.0.2.15         TCP   60 54479 → 55220 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

Figure 20: Network connection attempts in Wireshark

In Figure 20, you can see the host DNS resolution and the attempt to reach it via TCP. The TCP connection is not established, meaning the host doesn't reply with a [SYN,ACK] packet but rather a [RST,ACK] response is sent back. The port the malware attempts to connect to on the C&C is closed.

Because the C&C is not reachable as expected, we have essentially come to a "dead end" with the dynamic analysis step. You can still explore the domain for other open ports but we will not dive into it further in this article. Nonetheless, you gained crucial insights on how the malware operates once it runs, how to detect it on compromised systems, and how to clear the malware from your system.

Tip: Some malware behave differently when they run as a privileged vs. an unprivileged user. The malware we have analyzed will not gain persistence if it runs as an unprivileged user. It is recommended to run the file as both privileged and unprivileged users to understand the full spectrum of its capabilities.

Wrap-Up

We reviewed ELF dynamic analysis and detailed the different components and tools relevant for this step. You learned how to gather insights about a file's behavior using these tools. You also learned how insights gathered during initial analysis can help you focus on certain components during dynamic analysis.

There are certain cases where you will hit a dead end during the dynamic analysis process. The C&C could be down, the malware runs on a particular environment or time zone, and more. Remember, the goal in dynamic analysis is to gather additional insights on how the malware behaves. This will help you detect the malware on compromised systems, collect indicators related to the malware such as file names, C&C, persistent methods and more. This can lead you to connect the malware to other tools, campaigns and/or threat actors.

Dynamic analysis should be done responsibly. The safety of your system and network together with other machines should be taken into consideration when analyzing malware dynamically.

What's Next?

Next up you will learn how to get payloads from packers and loaders by extracting memory dumps.

Appendix

These tools and commands were used or mentioned in this article:

Avigayil Mechtinger

Avigayil is a product manager at Intezer, leading Intezer Analyze product lifecycle. Prior to this role, Avigayil was part of Intezer's research team and specialized in malware analysis and threat hunting. During her time at Intezer, she has uncovered and documented different malware targeting both Linux and Windows platforms.