

## Intrusion detection analysis

This analysis is performed on Kali Linux (version 2022.4) which is running as a virtual machine on Apple MacBook M1 Pro 2022 (macOS Ventura, version 13.1). The software used for virtualisation is UTM (version 4.0.9). The tool used for the analysis itself is Volatility Framework version 2.6.1. Analysed system is an Ubuntu machine.

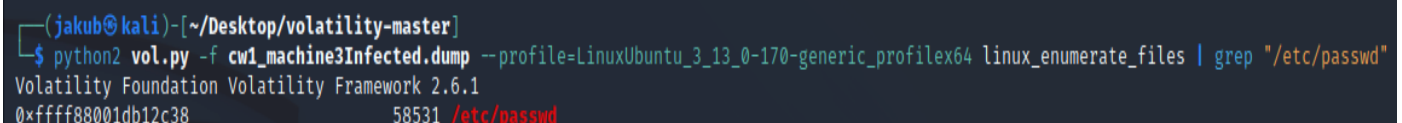
### 1. Users in the system

#### Infected memory dump analysis

The first step of this analysis is to find all users in the analysed system. There is no direct command that would print out all users that works with this profile, therefore another way must be found. In Linux operating systems, all users are listed in the file residing in `/etc/passwd`. To get the file from the memory dump, its exact memory address must be found first. The volatility plugin that is capable to do this is called `linux_enumerate_files` and it is used in combination with the `grep` command:

```
python2 -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_enumerate_files | grep "/etc/passwd"
```

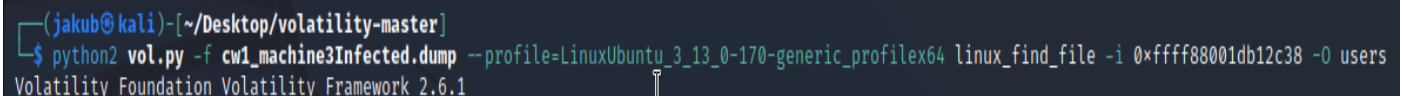
`linux_enumerate_files` will enumerate through files in the directory and output their exact memory address. The `grep` command is used to point to the file that is supposed to be enumerated. `grep` is used to find text data matching the expression given to the command.



```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_enumerate_files | grep "/etc/passwd"
Volatility Foundation Volatility Framework 2.6.1
0xffff88001db12c38 58531 /etc/passwd
```

Figure 1 "linux\_enumerate\_files | grep" used on the memory dump to find out memory address of the file containing all users in the system

Now that the memory address is known, the next step is to find the file and output its contents into a text file. To do this, the command `linux_find_file` is used with parameters `-i memory address` and `-O output file`.



```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_find_file -i 0xffff88001db12c38 -O users
Volatility Foundation Volatility Framework 2.6.1
```

Figure 2 `linux_find_file` plugin will make sure that all the contents of the file on memory address `0xffff88001db12c38` will be outputted into newly created "users" file.

Once the "users" file is created in filled, it can be examined by the Linux command `cat`.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ cat users
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
libuuid:x:100:101::/var/lib/libuuid:
syslog:x:101:104::/home/syslog:/bin/false
messagebus:x:102:106::/var/run/dbus:/bin/false
landscape:x:103:109::/var/lib/landscape:/bin/false
sshd:x:104:65534::/var/run/sshd:/usr/sbin/nologin
pollinate:x:105:1::/var/cache/pollinate:/bin/false
vagrant:x:1000:1000::/home/vagrant:/bin/bash
colord:x:106:112:colord colour management daemon,,:/var/lib/colord:/bin/false
statd:x:107:65534::/var/lib/nfs:/bin/false
puppet:x:108:114:Puppet configuration management daemon,,:/var/lib/puppet:/bin/false
ubuntu:x:1001:1001:Ubuntu:/home/ubuntu:/bin/bash
mysql:x:109:116:MySQL Server,,:/nonexistent:/bin/false
bot:x:1002:1002::,/home/bot:/bin/bash
rootkit:x:1003:1003::,/home/rootkit:/bin/bash
secrets:x:1004:1004::,/home/secrets:/bin/bash
```

Figure 3 File "users" outputted into terminal via command cat

File "users" contains many users, which could be chaotic at first glance and, these users are mostly not important for purposes of this analysis. To simplify the output and to make it show only information relevant for this analysis, the command `cat` needs to be modified. This modification can be done with the above-mentioned command `grep`, which will filter output. The parameter "bash" must be given to output only users in the system who can access the bash shell and therefore can make changes to the system.

`cat users | grep "bash"` is used for this purpose.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ cat users | grep "bash"
root:x:0:0:root:/root:/bin/bash
vagrant:x:1000:1000::/home/vagrant:/bin/bash
ubuntu:x:1001:1001:Ubuntu:/home/ubuntu:/bin/bash
bot:x:1002:1002::,/home/bot:/bin/bash
rootkit:x:1003:1003::,/home/rootkit:/bin/bash
secrets:x:1004:1004::,/home/secrets:/bin/bash
```

Figure 4 All the users in the system with access to bash are displayed in this output.

Users with access to bash shell
root
vagrant
ubuntu
bot
rootkit
secrets

Figure 5 table of system users with access to bash shell found in the analysis of the infected memory dump

## Clean memory dump analysis for verification

To verify the information above, an analysis of the clean memory dump is needed for comparison with the results from the infected memory dump. To get all users from the clean memory dump, the same process as above is performed on the clean memory dump.

The first step is to find the exact memory address of the file containing information about users in the system.

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_enumerate_files | grep "/etc/passwd"
Volatility Foundation Volatility Framework 2.6.1
0xffff88001d9aa860 58509 /etc/passwd
```

Figure 6 linux\_enumerate\_files is used to find memory address of file containing information about users in the system in the clean memory dump

The second step is to output the contents of the file on the found memory address into a newly created file named "cleanMemoryDumpUsers".

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_find_file -i 0xffff88001d9aa860 -o cleanMemoryDumpUsers
Volatility Foundation Volatility Framework 2.6.1
```

Figure 7 Content of the file is outputted into new file cleanMemoryDumpUsers

Last step is to check the contents of the file by using command cat.

```
(jakub@kali)~[~/Desktop/volatility-master]
$ cat cleanMemoryDumpUsers | grep "bash"
root:x:0:0:root:/root:/bin/bash
vagrant:x:1000:1000::/home/vagrant:/bin/bash
ubuntu:x:1001:1001:Ubuntu:/home/ubuntu:/bin/bash
```

Figure 8 Using command cat to see contents of the cleanMemoryDumpUsers file

Users with access to bash shell
root
vagrant
ubuntu

Figure 9 table of system users with access to bash shell found in the analysis of the clean memory dump

## Conclusion

Three suspicious system users were found in the infected memory dump. Names of some of these users indicate that they were created with malicious intent to install a rootkit and possibly prepare the machine to become a part of a botnet. To find out more about the reasons these users were created, a more thorough analysis of the system and circumstances is needed.

Users found in the clean memory dump	Users found in the infected memory dump
root	root
vagrant	vagrant
ubuntu	ubuntu
	bot
	rootkit
	secrets

Figure 10 Comparison of the users found in the clean and infected memory dump

## 2. Function hooking

### Definition of function hooking

Function hooking is a technique used to intercept and alter the behaviour of a function. This is typically done by intercepting and redirecting the execution flow of a program to a different function. This different function is known as a hook, and it can perform additional tasks before or after the original function is called. Function hooking is used for multiple purposes, such as debugging. It can also be used for malicious purposes. Best-known cases of such use are:

- Keylogging – function hooking can be used to intercept and record keystrokes. This could lead to the loss of login credentials, credit card information or other sensitive data.
- Adware – this type of malware uses function hooking to modify the behaviour of web browsers. If successful, it would display unwanted ads on the victim's computer, or redirect them to malicious websites.
- Rootkit – this is a type of malware that uses function hooking to hide its presence on a system. It can intercept system calls and hide files and processes to stay hidden. This one is the biggest security threat of them all because it can enable hackers to gain root access to a system [1].

### Detection of function hooking

The first step is to start this analysis on the clean dump. This is to obtain some image of what is going on in the system and to compare it to the infected dump right away. This will help to uncover suspicious activity faster.

Linux command tree is used to display all running processes in a tree-like structure. This structure helps to understand the relationship between processes – parent and child process. Parent processes create child processes by passing some information into them – for example another command to be executed.

```
python2 -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_pstree
```

Output of the command:

```
└─$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_pstree
Volatility Foundation Volatility Framework 2.6.1
Name      Pid      Uid
init      1
.upstart-udev-br 413
.systemd-udev 423
..systemd-udev 4060
.dhclient  528
.rpcbind   648
.rpc.statd  673      107
.upstart-socket- 676
.dbus-daemon 797      102
.rpc.idmapd 815
.systemd-logind 839
.rsyslogd   842      101
.upstart-file-br 881
.getty      955
.getty      958
.getty      962
.getty      963
.getty      965
.sshd       1004
.acpid      1006
.cron       1007
.atd        1008
.VBoxService 1057
.puppet     1085
.ruby       1115
```

Figure 11 Output of the pstree command on the clean memory dump. There are few more processes not shown in the picture.

```
python2 -f cw_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_pstree
```

Output of the command:

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_pstree
Volatility Foundation Volatility Framework 2.6.1
Name                Pid      Uid
init                 1
.upstart-udev-br     373
.systemd-udevd       378
.rpcbind             514
.rpc.statd           530      107
.upstart-socket-     534
.dhclient            656
.dbus-daemon         842      102
.systemd-logind      915
.rpc.idmapd          918
.rsyslogd            933      101
.upstart-file-br     973
.getty               1050
.getty               1053
.getty               1057
.getty               1058
.getty               1060
.acpid               1105
.cron                1106
.atd                 1107
.VBoxService         1165
.sshd                1241
.. sshd              2090
... sshd             2161      1000
....bash             2162      1000
.....su              8494      1000
.....bash            8495
.....bash            10296
.....wget            10478
.....wget            10479
```

Figure 12 Partial output of the linux\_pstree command. Output is too long to be included completely but there is noticeable difference in comparison to the output from the clean memory dump.

Visible difference in comparison to the clean dump is found in the output of pstree command - “sshd” process with multiple child processes.

```
.sshd                1241
.. sshd              2090
... sshd             2161      1000
....bash            2162      1000
.....su              8494      1000
.....bash            8495
.....bash            10296
.....wget            10478
.....wget            10479
.....wget            10485
.....wget            10486
.....wget            10487
.....wget            10491
.....wget            10492
```

Figure 13 Suspicious processes uncovered by linux\_pstree. Notice the sshd process and all the wgets under. There are more wgets than showed in the picture.

There is a running process named sshd – the Secure Shell Daemon application for ssh. This program is known for providing encrypted communication between two *untrusted* hosts. There are many child processes of the sshd process. As said above, this means that the child processes were executed from within their parent processes, ergo there are other commands executed from the sshd process and therefore from another host. Very suspicious under these circumstances is the execution of the bash - closer examination is needed.

For closer examination of the process, command

```
python2 -f cw_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_psen -p 2162
```

is used. `-p {number}` is a specification of the process id. The `psenv` plugin is used to print out environment variables for a specific process. These variables store information about the system and user but can store more information.

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_psenv -p 2162
Volatility Foundation Volatility Framework 2.6.1
Name Pid Environment
bash 2162 LANG=en_US.UTF-8 LC_TERMINAL=iTerm2 LC_TERMINAL_VERSION=3.4.10 USER=vagrant LOGNAME=vagrant HOME=/home/vagrant PATH=/usr/local/s
bin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games MAIL=/var/mail/vagrant SHELL=/bin/bash SSH_CLIENT=10.0.2.2 45078 22 SSH_CONN
ECTION=10.0.2.2 45078 10.0.2.15 22 SSH_TTY=/dev/pts/0 TERM=xterm-256color XDG_SESSION_ID=2 XDG_RUNTIME_DIR=/run/user/1000
```

Figure 14 Psenv used to show environment variables of the bash process

This command shows that the bash shell process was executed through an SSH connection. The process was executed under the user *vagrant*. To examine what exactly was done under this process, volatility module *bash* is used. The bash plugin allows for analysis of the bash shell history.

`python2 -f cw_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash -p 2162`

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash -p 2162
Volatility Foundation Volatility Framework 2.6.1
Pid Name Command Time Command
2162 bash 2021-10-27 16:52:08 UTC+0000 sudo apt-get install apache2
2162 bash 2021-10-27 16:52:26 UTC+0000 ls /home/
2162 bash 2021-10-27 16:52:32 UTC+0000 sudo apt-get -y install apache2
2162 bash 2021-10-27 16:52:44 UTC+0000 sudo apt-get -y install mysql-server php5 ircd-hybrid ii
2162 bash 2021-10-27 16:53:25 UTC+0000 ifconfig
2162 bash 2021-10-27 16:53:32 UTC+0000 sudo adduser bot
2162 bash 2021-10-27 16:53:48 UTC+0000 sudo adduser rootkit
2162 bash 2021-10-27 16:54:05 UTC+0000 sudo adduser secrets
2162 bash 2021-10-27 16:54:22 UTC+0000 cd /vagrant/
2162 bash 2021-10-27 16:54:23 UTC+0000 ls
2162 bash 2021-10-27 16:54:26 UTC+0000 cd Diamorphine/
2162 bash 2021-10-27 16:54:27 UTC+0000 ls
2162 bash 2021-10-27 16:54:34 UTC+0000 insmod diamorphine.ko
2162 bash 2021-10-27 16:54:39 UTC+0000 sudo insmod diamorphine.ko
2162 bash 2021-10-27 16:54:46 UTC+0000 ssh rootkit
2162 bash 2021-10-27 16:54:50 UTC+0000 su rootkit
```

Figure 15 Bash shell history. These bash commands were executed through the ssh connection

History of bash shell from figure 4 shows all the commands that were used via the ssh connection.

Analysis of the used commands:

- `sudo apt-get install apache2` – installs apache2, open-source HTTP server
- `ls /home/` - list all files and directories in the home directory
- `sudo apt-get -y install apache2` – again, installation of the apache2 web server, `-y` means “answer yes to all prompts”
- `sudo apt-get -y install mysql-server php5 ircd-hybrid ii` – installation of multiple tools, most noticeable `ircd-hybrid ii`
- `ifconfig` – this command prints configuration of network interfaces
- `sudo adduser bot` - new user “bot” added
- `sudo adduser rootkit` – new user “rootkit” added
- `sudo adduser secrets` – new user “secrets” added
- `cd /vagrant/` - navigates to user’s “vagrant” directory
- `ls` – lists all files and directories
- `cd Diamorphine` – navigates to the Diamorphine directory



- ls
- insmod diamorphine.ko – insmod is used to insert modules into kernel, but it needs to be executed with super user privileges. .ko file extension means it is a file that extends kernel of the Linux.
- sudo insmod diamorphine.ko – same command as before, but executed with the super user privileges
- ssh rootkit – ssh connection to the “rootkit” user, probably just a typo
- su rootkit – execute commands under user rootkit

The bash history inspected above is showing signs of malicious activity. Installation of the apache2 web server and ircd-hybrid are giving away a hint that the attacker is probably trying to add this computer to their botnet [2]. A botnet is a network of compromised computers that are controlled by a black hat hacker. Botnets are often used to perform distributed denial of service attacks but can be also used for spreading malware or a phishing campaign.

Ircd-Hybrid is an open-source implementation of an Internet Relay Chat. It is used for real-time communication amongst users over the internet. It can be misused and set up as a command-and-control server, to send commands to the compromised computers.

Another malicious command is used to insert diamorphine.ko into the kernel. Diamorphine is a known *loadable kernel module rootkit*. To load the kernel module, super user privileges are needed. Once loaded, these rootkits are very powerful and enable hackers to manipulate systems in many ways [3].

To double if the rootkit was loaded into the kernel, plugin *check\_modules* is used. This module will list all currently loaded kernel modules. First, it is performed on the clean dump:

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_check_modules
Volatility Foundation Volatility Framework 2.6.1
Module Address      Core Address      Init Address Module Name
```

Figure 16 Check\_modules plugin is used on the clean dump to list all currently loaded kernel modules.

The result is blank, which means there are not any loaded kernel modules in the system. To make sure this information is accurate and there are no hidden kernel modules, plugin *hidden\_modules* is also used on the clean dump.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_hidden_modules
Volatility Foundation Volatility Framework 2.6.1
Offset (V)      Name
```

Figure 17 Checking for hidden loaded kernel modules with hidden\_modules plugin

This also comes back blank, which means there are no hidden loaded kernel modules in the system. The same examination is performed on the infected memory dump.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_check_modules
Volatility Foundation Volatility Framework 2.6.1
Module Address      Core Address      Init Address Module Name
0xfffffffffa016a000 0xfffffffffa0168000 0x0 diamorphine
```

Figure 18 Checking for loaded kernel modules with check\_modules plugin

This confirms that diamorphine was successfully loaded into the kernel. To make sure there are no other modules, the hidden\_modules plugin is used.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_hidden_modules
Volatility Foundation Volatility Framework 2.6.1
Offset (V)      Name
0xfffffffffa016a000 diamorphine
```

Figure 19 hidden\_modules plugin only shows diamorphine, that means there are no other kernel modules

Now it is established that the diamorphine rootkit kernel module was successfully loaded into the kernel, it is needed to establish the extent of its influence on the system. It is needed to verify if there are any files opened by the kernel module. Kernel modules usually do not directly open any files; thus, it is highly unusual and suspicious if they do. To verify this kernel\_opened\_files module is used. This technique is also used to find hidden kernel modules in the system.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_kernel_opened_files
Volatility Foundation Volatility Framework 2.6.1
Offset (V)      Partial File Path
```

Figure 20 Verifying there are not any files opened by the kernel modules

The result is blank, which means there are not any files opened by the kernel modules.

The next step in this analysis is to look for the kernel hooks. The kernel modules can intercept system calls, modifying the behaviour of the system, as explained at the beginning of this chapter. Analysis of kernel hooks can be performed by the plugin check\_syscall. This plugin is used to find hooks in the kernel's system call table. This table contains pointers to the functions that handle system calls. If any of the addresses in the table were modified, this plugin will flag it as HOOKED.

First, this check is performed on the clean memory dump.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuClean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_check_syscall
Volatility Foundation Volatility Framework 2.6.1
Table Name Index System Call Handler Address Symbol
64bit 0 0xffffffff81c9a40 Sys_read
64bit 1 0xffffffff81c9ae0 sys_write
64bit 2 0xffffffff81c8530 sys_open
64bit 3 0xffffffff81c6550 Sys_close
64bit 4 0xffffffff81ce5a0 Sys_newstat
64bit 5 0xffffffff81ce5d0 Sys_newfstat
64bit 6 0xffffffff81ce5b0 sys_newlstat
64bit 7 0xffffffff81dea30 Sys_poll
64bit 8 0xffffffff81c8bb0 sys_lseek
64bit 9 0xffffffff81019820 Sys_mmap
64bit 10 0xffffffff8118c630 sys_mprotect
64bit 11 0xffffffff8118b0c0 sys_munmap
64bit 12 0xffffffff8118a520 Sys_brk
64bit 13 0xffffffff81082ea0 Sys_rt_sigaction
64bit 14 0xffffffff81081bb0 sys_rt_sigprocmask
64bit 15 0xffffffff8174dc10 stub_rt_sigreturn
64bit 16 0xffffffff811dc9e0 Sys_ioctl
64bit 17 0xffffffff811c9b80 Sys_pread64
64bit 18 0xffffffff811c9c30 sys_pwrite64
```

Figure 21 Output of the check\_syscall plugin. It is too long to be stored here in full length

Even on the clean memory dump, there has been found multiple hooked functions, as is visible in Figure 22. To make the output more readable, it is filtered with the use of `grep`.



```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_check_syscall | grep "HOOKED"
Volatility Foundation Volatility Framework 2.6.1
32bit 356 0xffffffff81d58fe2 HOOKED: UNKNOWN
32bit 357 0xffffffff81d58fe2 HOOKED: UNKNOWN
32bit 358 0xffffffff81d58fe2 HOOKED: UNKNOWN
32bit 359 0xffffffff81d58fe2 HOOKED: UNKNOWN
32bit 360 0xffffffff81d58ffd HOOKED: UNKNOWN
32bit 361 0xffffffff81d58ffd HOOKED: UNKNOWN
32bit 362 0xffffffff81d58fe2 HOOKED: UNKNOWN
32bit 363 0xffffffff81d58ff6 HOOKED: UNKNOWN
32bit 364 0xffffffff81d58fdb HOOKED: UNKNOWN
32bit 365 0xffffffff81d59004 HOOKED: UNKNOWN
32bit 366 0x5f7465735f696665 HOOKED: UNKNOWN
32bit 367 0x73736d6d5f637472 HOOKED: UNKNOWN
32bit 369 0xffffffff81069a70 HOOKED: UNKNOWN
32bit 370 0xffffffff81068d40 HOOKED: UNKNOWN
32bit 371 0xffffffff81068d68 HOOKED: UNKNOWN
32bit 372 0xffffffff81069870 HOOKED: UNKNOWN
32bit 373 0xffffffff81069850 HOOKED: UNKNOWN
32bit 374 0xffffffff810698a0 HOOKED: UNKNOWN
```

Figure 22 Hooked functions found in the clean memory dump by check\_syscall plugin. There are many more

To explain findings in the clean memory dump, it is important to understand that hooked functions are not always malicious. As mentioned in the explanation at the beginning of this chapter, function hooking can be, in most cases used to debug, monitor, or add new features to the system.

The same analysis is performed on the infected memory dump and the results are compared. The same plugin is used.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_check_syscall
Volatility Foundation Volatility Framework 2.6.1
Table Name Index System Call Handler Address Symbol
64bit 0 0xffffffff811c9a40 Sys_read
64bit 1 0xffffffff811c9ae0 sys_write
64bit 2 0xffffffff811c8530 sys_open
64bit 3 0xffffffff811c6550 Sys_close
64bit 4 0xffffffff811ce5a0 Sys_newstat
64bit 5 0xffffffff811ce5d0 Sys_newfstat
64bit 6 0xffffffff811ce5b0 sys_newlstat
64bit 7 0xffffffff811dea30 Sys_poll
64bit 8 0xffffffff811c8bb0 sys_lseek
64bit 9 0xffffffff81019820 Sys_mmap
64bit 10 0xffffffff8118c630 sys_mprotect
64bit 11 0xffffffff8118b0c0 sys_munmap
64bit 12 0xffffffff8118a520 Sys_brk
64bit 13 0xffffffff81082ea0 Sys_rt_sigaction
64bit 14 0xffffffff81081bb0 sys_rt_sigprocmask
64bit 15 0xffffffff8174dc10 stub_rt_sigreturn
64bit 16 0xffffffff811dc9e0 Sys_ioctl
64bit 17 0xffffffff811c9b80 Sys_pread64
64bit 18 0xffffffff811c9c30 sys_pwrite64
64bit 19 0xffffffff811ca0d0 sys_readv
```

Figure 23 Output of the check\_syscall plugin, investigating possible hooked functions. The output is too long to shown wholly

To filter out only the hooked function, *grep "HOOKED"* is used again.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_check_syscall | grep "HOOKED"
Volatility Foundation Volatility Framework 2.6.1
64bit 62 0xfffffffffa01684d0 HOOKED: diamorphine/hacked_kill
64bit 78 0xfffffffffa0168220 HOOKED: diamorphine/hacked_getdents
64bit 217 0xfffffffffa0168070 HOOKED: diamorphine/hacked_getdents64
32bit 356 0xffffffff81d58fe2 HOOKED: UNKNOWN
32bit 357 0xffffffff81d58fe2 HOOKED: UNKNOWN
```

Figure 24 Filtered output of the check\_syscall in the infected dump. Diamorphine comes up

The first three functions are hooked via Diamorphine. After a comparison of the clean and infected dump, hooked functions following the Diamorphine ones are the same as in the clean memory dump, meaning these hooked functions are not malicious.

To see which functions are hooked, it is needed to get back to the clean memory dump and filter out the functions according to their index number. In this case, the table name and index numbers of the hooked

functions by Diamorphine – 62,78, 217 – are used as parameters for filtering. To do this, *the awk* command is used.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-generic_profilex64 linux_check_syscall | awk '$1="64bit" 66 ($2 == 62 || $2 == 78 || $2 == 217)'
Volatility Foundation Volatility Framework 2.6.1
64bit      62      0xfffffffff81082510 sys_kill
64bit      78      0xfffffffff811dcef0 Sys_getdents
64bit     217      0xfffffffff811dd010 Sys_getdents64
```

Figure 25 Search result on the clean memory dump, showing functions which were hooked in the infected dump

The three functions which are hooked into the infected memory dump are:

- `sys_kill` – is a system call that is used in the Linux OS to send a signal into a process or group of processes. This process can have multiple purposes – to kill the process, interrupt it or other types of signals. This is beneficial for the attacker because it is possible to hide the presence of some processes or files in the system. It also allows the malware to ignore the kill command against it and helps to avoid detection [4]
- `Sys_getdents` – is a system call used to read the content of the directories, like the 'ls' command
- `Sys_getdents64` – same as the `Sys_getdents`, but the function is available on the 64-bit systems [5]

With the hooked function `getdents`, Diamorphine is most likely able to hide the presence of files and directories in the system, allowing the malware to evade detection of malicious activity. These files and directories can be hidden in a way that they are not even visible to the system administrator.

## Conclusion

In conclusion, during the analysis of the bash history in the infected memory dump, it became clear that there is a rootkit kernel module loaded into the system. Since the rootkit is a kernel module, this indicates the possibility of hooked kernel functions. This possibility was confirmed in the last part of the analysis, uncovering three hooked kernel functions in the infected memory dump, enabling attackers to manipulate system processes and hide malware's presence in the system.

### 3. Suspicious script

In the chapter 2 analysis, many hints about where to look were found in the processes and bash history. These gave away all the indicators needed to find the kernel hooks, but they were not inspected in full depth. Therefore, another examination of processes and bash history is needed, to continue beyond the malicious kernel module installation.

In this chapter, only the infected memory dump will be analysed until stated otherwise – all the needed comparisons of processes in the clean and infected memory dump have been performed in [chapter 2 analysis, starting with Figure 11](#).

Plugin *pstree* is used again to show all the running processes in the infected dump.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_pstree
Volatility Foundation Volatility Framework 2.6.1
Name      Pid      Uid
init      1
.upstart-udev-br 373
.systemd-udev  378
.rpcbind      514
.rpc.statd    530      107
.upstart-socket- 534
.dhclient     656
.dbus-daemon  842      102
```

Figure 26 *pstree* plugin shows all running processes in the infected memory dump. There are many, therefore they are not all shown in this picture.

It is needed to look again at the suspicious processes that were inspected in chapter 2. These are part of the output of the *pstree* plugin.

```
.sshd      1241
..sshd     2090
...sshd    2161      1000
....bash   2162      1000
.....su    8494      1000
.....bash  8495
.....bash  10296
.....wget  10478
.....wget  10479
.....wget  10485
.....wget  10486
.....wget  10487
.....wget  10491
.....wget  10492
```

Figure 27 continuation of the *pstree* plugin output, showing suspicious processes.

In [chapter 2](#), the comparison has been done to the clean memory dump and these suspicious processes (sshd and all its child processes) are not in the clean memory dump.

Bash process no. 2162 must be inspected again to find out what previous actions led to the execution of the following processes. To perform this inspection, bash history is revealed with the volatility plugin *linux\_bash*.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash -p 2162
Volatility Foundation Volatility Framework 2.6.1
```

Pid	Name	Command Time	Command
2162	bash	2021-10-27 16:52:08 UTC+0000	sudo apt-get install apache2
2162	bash	2021-10-27 16:52:26 UTC+0000	ls /home/
2162	bash	2021-10-27 16:52:32 UTC+0000	sudo apt-get -y install apache2
2162	bash	2021-10-27 16:52:44 UTC+0000	sudo apt-get -y install mysql-server php5 ircd-hybrid ii
2162	bash	2021-10-27 16:53:25 UTC+0000	ifconfig
2162	bash	2021-10-27 16:53:32 UTC+0000	sudo adduser bot
2162	bash	2021-10-27 16:53:48 UTC+0000	sudo adduser rootkit
2162	bash	2021-10-27 16:54:05 UTC+0000	sudo adduser secrets
2162	bash	2021-10-27 16:54:22 UTC+0000	cd /vagrant/
2162	bash	2021-10-27 16:54:23 UTC+0000	ls
2162	bash	2021-10-27 16:54:26 UTC+0000	cd Diamorphine/
2162	bash	2021-10-27 16:54:27 UTC+0000	ls
2162	bash	2021-10-27 16:54:34 UTC+0000	insmod diamorphine.ko
2162	bash	2021-10-27 16:54:39 UTC+0000	sudo insmod diamorphine.ko
2162	bash	2021-10-27 16:54:46 UTC+0000	ssh rootkit
2162	bash	2021-10-27 16:54:50 UTC+0000	su rootkit

Figure 28 Output of the plugin `linux_bash`, showing bash history on the bash process.

All these used commands were explained in detail in chapter 2 of this analysis. For this chapter, it is needed to focus on the last line of the output, command `us rootkit`. This command is used to execute commands under the user rootkit. As seen in Figure 27, another bash process was opened as a child process to the `su` process. To inspect what is happening there, the bash history must be revealed again with the use of the same `linux_bash` plugin on the process ID 8495. But first, verification that this bash process runs under the user rootkit, as expected. `linux_psenv` module is used to verify this.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_psenv -p 8495
Volatility Foundation Volatility Framework 2.6.1
```

Name	Pid	Environment
bash	8495	XDG_SESSION_ID=2 TERM=xterm-256color SHELL=/bin/bash SSH_CLIENT=10.0.2.2 45078 22 SSH_TTY=/dev/pts/0 <b>USER=rootkit</b> MAIL=/var/mail/r ootkit PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games PWD=/vagrant/Diamorphine LANG=en_US.UTF-8 SHLVL=1 HOME =/home/rootkit LC_TERMINAL_VERSION=3.4.10 LOGNAME=rootkit SSH_CONNECTION=10.0.2.2 45078 10.0.2.15 22 LESSOPEN= /usr/bin/lesspipe %s XDG_RUNTIME_DIR=/run/u ser/1000 LC_TERMINAL=iTerm2 LESSCLOSE=/usr/bin/lesspipe %s %s _=/bin/su OLDPWD=/vagrant

Figure 29 Verification of user under which is the bash process runned. It shows `USER=rootkit`

Now that it is confirmed that this bash process run under one of the users that were added to infected dump, it's time to look at the bash history.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash -p 8495
Volatility Foundation Volatility Framework 2.6.1
```

Pid	Name	Command Time	Command
8495	bash	2021-10-27 16:55:14 UTC+0000	kill -64 1
8495	bash	2021-10-27 16:55:20 UTC+0000	apt-get install make
8495	bash	2021-10-27 16:55:25 UTC+0000	apt-get install python3
8495	bash	2021-10-27 16:55:31 UTC+0000	apt-get install python2.7
8495	bash	2021-10-27 16:55:37 UTC+0000	apt-get install emacs
8495	bash	2021-10-27 17:02:36 UTC+0000	cd
8495	bash	2021-10-27 17:02:41 UTC+0000	cp /vagrant/script.bash .
8495	bash	2021-10-27 17:03:03 UTC+0000	tc qdisc add dev eth0 root tbf rate 28kbit latency 50ms burst 1540
8495	bash	2021-10-27 17:03:08 UTC+0000	nano script.bash
8495	bash	2021-10-27 17:03:20 UTC+0000	bash script.bash

Figure 30 Output of the `linux_bash` plugin. Bash history of the process id. 8495 is shown

Analysis of the commands used:

- `kill -64 1` – sends signal “64” into process id 1. Signal 64 is not a common signal; it is a custom defined signal in the Diamorphine rootkit. It is used to elevate user privileges to root privileges<sup>1</sup>. Process id 1 is known as the init process. This process is taking care of the overall functioning of the system, such as starting and terminating processes. Elevated privileges in this process give attacker root privileges - the ability to perform actions that would completely affect the system, such as start or stop system services, modifying and adding users, replacing processes and much more.
- `apt-get install makes` – installs the make package. Make package is used to build and compile software on Linux machines.

<sup>1</sup> <https://github.com/m0nad/Diamorphine/blob/master/README.md>

- apt-get-install python3 – installs python version 3 package (programming language interpreter)
- apt-get install python2.7 – installs python version 2.7 package (programming language interpreter)
- apt-get install emacs – install emacs, a powerful text editor suited for coding.
- cd – used to go back to the home directory of the current user
- cp /vagrant/script.bash, – cp is used for files or directories. script. bash was copied to the root folder
- tc qdisc add dev eth0 root tbf rate 28kbit latency 50ms burst 1540 – this command is used to configure network traffic control – an average rate of data, maximum delay, and burst. It also employs a Token Bucket Filter, which controls the traffic rate of network interface. This is particularly interesting because by setting a low-rate limit and high burst size, the system could be used for effective Denial of Service attacks.
- nano script.bash – opens script.bash in the nano text editor. This text editor is used for coding.
- bash script.bash – executes script.bash

The analysis conducted so far indicates that the machine is maybe becoming a part of a botnet, a network of compromised computers under the control of a hacker, that is getting set up to be used as a port for Distributed Denial of Service attacks. This is just a working theory based on the hints from the system so far. Further examination may uncover if this theory is true. The edited script found in the home directory of the rootkit user is a good following step to the previous analysis, as it is visible in the processes in Figure 27 and the bash history in Figure 30, another bash process was started with this script. First, an inspection of the process environment is performed, to get more detail. To perform this inspection, the plugin *linux\_psen* outputs the environment of the process is used.

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_psen -p 10296
Volatility Foundation Volatility Framework 2.6.1
Name Pid Environment
bash 10296 XDG_SESSION_ID=2 SHELL=/bin/bash TERM=xterm-256color SSH_CLIENT=10.0.2.2 45078 22 SSH_TTY=/dev/pts/0 USER=rootkit PATH=/usr/local/
sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games MAIL=/var/mail/rootkit PWD=/home/rootkit LANG=en_US.UTF-8 HOME=/home/rootkit
SHLVL=2 LC_TERMINAL_VERSION=3.4.10 LOGNAME=rootkit SSH_CONNECTION=10.0.2.2 45078 10.0.2.15 22 LESSOPEN= /usr/bin/lesspipe %s LC_TERMINAL=iTerm2 XDG_RUNTIM
E_DIR=/run/user/1000 LESSCLOSE=/usr/bin/lesspipe %s %s _=/bin/bash OLDPWD=/vagrant/Diamorphine
```

Figure 31 Process environment of the suspicious process

Multiple information is outputted by the *linux\_psen* plugin. The process is executed by the user rootkit (as seen within the USER environment variable). There is also an SSH environment variable, which shows us that there is an ongoing SSH connection in this process. The interesting environment variables are:

- PWD – shows the current working directory, which is /home/rootkit
- OLDPWD – shows the previous working directory, which, according to this process environment variable is /vagrant/Diamorphine

Rootkit such as Diamorphine is capable to manipulate the OLDPWD environment variable. This is done to hide malicious activity in the system because if the user wants to move to the previous working directory from within the directory when the process is located, thanks to this manipulation, the previous working directory would be different than it is. This could help the rootkit to hide its process better.

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash -p 10296
Volatility Foundation Volatility Framework 2.6.1
Pid Name Command Time Command
```

Figure 32 Attempt to inspect bash history with plugin *linux\_bash* returned blank

The output returned blank, but the fact that many wget processes are running under this process means that something is going on. To investigate this process, one way is to create a dump of the process and investigate it with various tools, such as strings or load it into the debugger. Plugin *procdump* is used for dump creation.



```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_procdump -D dump -p 10296
Volatility Foundation Volatility Framework 2.6.1
Offset      Name      Pid      Address      Output File
-----
0xffff880005213000 bash      10296      0x0000000000400000 dump/bash.10296.0x400000
```

Figure 33 Successfully performed procdump created dump of the process in the directory named dump

To inspect the dump, the `strings -a` command is used. Strings will find and print text strings in the process dump, which is an executable file.

```
(jakub@kali)~[~/Desktop/volatility-master/dump]
$ strings -a bash.10296.0x400000
/lib64/ld-linux-x86-64.so.2
$DJ
0'0
"0B1
B82
pD@kB
%) P
9E$
NR l
"7$aD
"% @0A
Hap5
```

Figure 34 Strings -a gives large output, mostly undreadable or not important information.

The first line is used by OS to load shared libraries and resolve symbols at runtime. It is known as the dynamic linker. While it is possible to hook this function, the previous analysis does not indicate that this is the case and therefore it is a normal declaration of the dynamic linker at the runtime. This analysis did not show anything, and another tool must be tried – a debugger. GDB is the debugger to be used.

```
(jakub@kali)~[~/Desktop/volatility-master/dump]
$ gdb bash.10296.0x400000
GNU gdb (Debian 12.1-4) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word" ...
BFD: warning: /home/jakub/Desktop/volatility-master/dump/bash.10296.0x400000 has a section extending past end of file
BFD: warning: /home/jakub/Desktop/volatility-master/dump/bash.10296.0x400000 has a corrupt string table index - ignoring
Reading symbols from bash.10296.0x400000 ...
(No debugging symbols found in bash.10296.0x400000)
```

Figure 35 GDB debugger is unsuccessfully used on the process dump

Since the GDB debugger is also not working, another approach must be found. This would be to dump the `script.bash` file and investigate it. To dump it, the same commands are used as in chapter 1 of this analysis, while getting the `passwd` file.

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_enumerate_files | grep "script.bash"
Volatility Foundation Volatility Framework 2.6.1
0xffff88001f7a4860      268512 /home/rootkit/script.bash
```

Figure 36 enumerate\_files plugin is used to find the file, grep is used to specify the name of the file

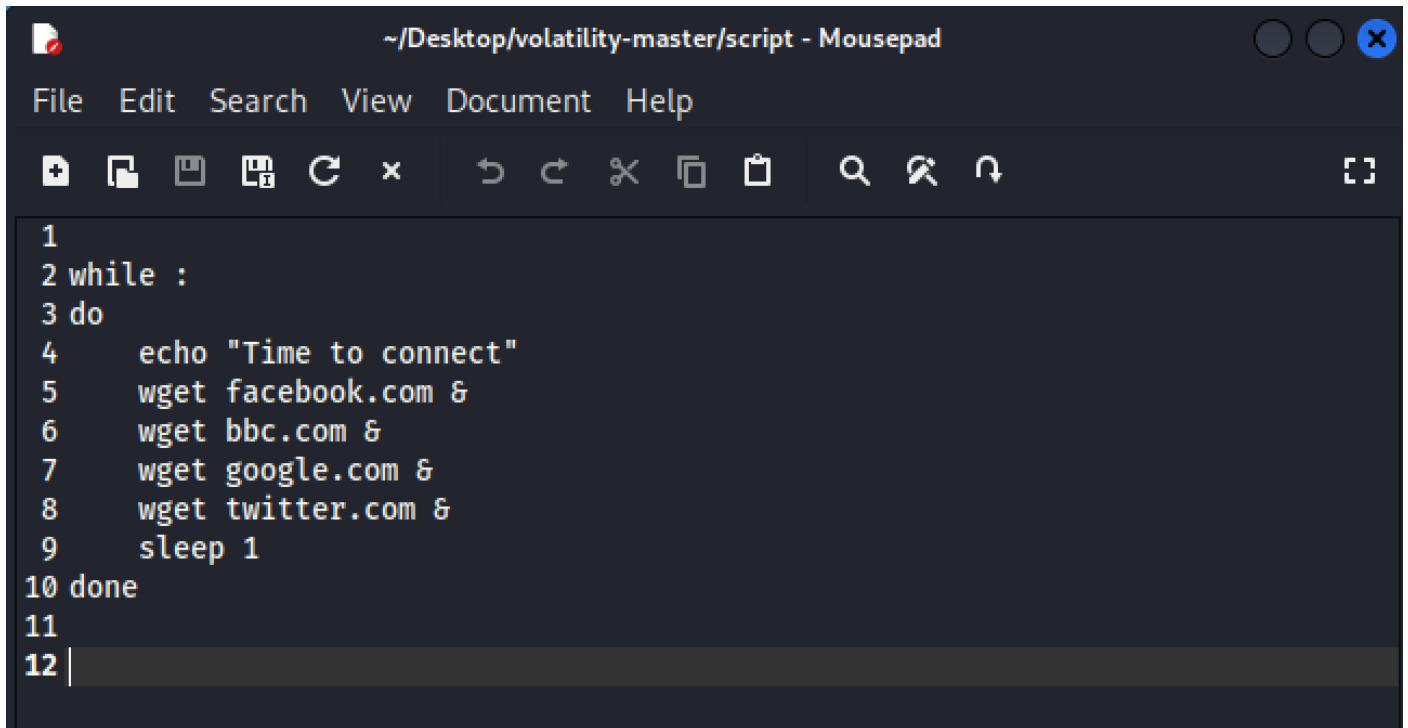
Address on which the file resides is returned by the plugin. Next step is to dump the script.

```
(jakub@kali)~[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_find_file -i 0xffff88001f7a4860 -O scr
ipt
Volatility Foundation Volatility Framework 2.6.1
```

Figure 37 find\_file plugin is used to get the file from the specified address

Now that file is dumped, it's time to see what this script is about.





```
1
2 while :
3 do
4     echo "Time to connect"
5     wget facebook.com &
6     wget bbc.com &
7     wget google.com &
8     wget twitter.com &
9     sleep 1
10 done
11
12 |
```

Figure 38 Code inside the script responsible for the suspicious process

The suspicious bash script contains a never-ending loop. In this loop, wget requests are made to multiple websites. Wget is a command used for file downloads from the internet. It supports multiple protocols, such as HTTP, HTTPS, and FTP. These requests are made in the never-ending loop with a 1-second pause between every loop (the sleep command). This explains the large number of wget requests found under the suspicious process.

## Conclusion

In this part of the analysis, multiple malicious or suspicious actions were performed. Privilege escalation and suspicious configuration of the network traffic indicate that this machine could become part of the botnet intended for DDoS attacks. Analysis of the suspicious process and the script that starts it uncovers that the script contains a never-ending loop which is sending wget requests to multiple websites with a 1-second pause between every loop. This could potentially be a DoS attack.

## 4. Statically and dynamically injected libraries

### Explanation of the statically and dynamically injected libraries

**A statically injected library** is a library that is added to a program at runtime via a process called library injection. This means that the library is loaded into the address space of the program during the program compilation, and the program can use the functions provided by the library as if they were part of the program itself. Statically injected libraries are used to add functionality to a program without modifying the original code [6].

This can be useful in situations where programmer want to add features to a program that they don't have the source code for, or when they want to add features to a program that is protected by anti-tampering mechanisms. It's important to say that library injection can be used for both non-malicious and malicious purposes. While it can be useful for debugging, it can also be used to inject malware or to perform other malicious activities.

**A dynamically injected library** is loaded into the address space of a process at runtime, but it is not incorporated into the executable file of the program. Instead, the program makes use of the functions provided by the library by calling them through a process called dynamic linking. This means that the library is not a part of the final executable and must be present on the target system for the program to execute [7].

Dynamic linking allows for multiple programs to share the same library, which can save memory and disk space. This contrasts with static linking, where each program has its own copy of the library code, which can cause issues with disk space and memory usage. The process of dynamic linking can be done in different ways, one of the most common is called Dynamic Linking Library (DLL) in Windows systems, and Shared Object (SO) in Linux systems.

Dynamically injected libraries are used to add functionality to a program in a way that allows multiple programs to share the same code. This can be useful in situations where multiple programs need to use the same functionality, and where disk space and memory usage are a concern. It's important to say that just as with statically injected libraries, dynamically injected libraries can also be used for both legal and illegal purposes. For example, they can be used to add functionality to a program, but they can also be used to inject malware or to perform other malicious activities.

### Identification of the dynamic injected libraries in the /bin/rm process

To identify the libraries, the first step is to find the running rm process. These processes usually descend from the parent bash process, since the rm is a bash command used to delete files. Therefore, all bash processes on the infected dump are inspected for this child process via `linux_bash | grep "rm"`, to filter out other processes.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash | grep "rm"
Volatility Foundation Volatility Framework 2.6.1
```

Figure 39 Output of the `linux_bash | grep "rm"` is empty on the infected dump

Since the output is empty, this means that the `"rm"` command was not used on the infected dump. The same inspection is performed on the clean dump, via the same commands as above.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash | grep "rm"
Volatility Foundation Volatility Framework 2.6.1
1232 bash      2022-11-10 10:34:48 UTC+0000  rm Ubuntu_3.13.0-170-generic_profile.zip
1232 bash      2022-11-10 10:34:48 UTC+0000  rm Ubuntu_3.13.0-170-generic_profile.zip
1232 bash      2022-11-10 10:34:48 UTC+0000  rm Ubuntu_3.13.0-170-generic_profile.zip
```

Figure 40 This time the processes are found on the clean memory dump

As per Figure 40, *rm* processes in the clean memory dump are descending from the bash process id 1232. Multiple commands have been tried to find out if these processes carry their process IDs, such as command volatility module *linux\_threads* or *linux\_pslist*, but these attempts were not successful. This means that it is needed to inspect the whole bash process to identify the dynamically injected libraries.

To list all the libraries and at the same time recognise which ones are dynamically and which are statically injected, the volatility plugin *linux\_library\_list* is used. This plugin outputs all libraries in the chosen process and shows which is dynamically and which one is statically injected according to True or False.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f ubuntuclean.lime --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_ldrmodules -p 1232
Volatility Foundation Volatility Framework 2.6.1
```

Pid	Name	Start	File Path	Kernel	Libc
1232	bash	0x0000000000400000	/bin/bash	True	False
1232	bash	0x00007fd24ef17000	/lib/x86_64-linux-gnu/libc-2.19.so	True	True
1232	bash	0x00007fd24f70d000	/lib/x86_64-linux-gnu/ld-2.19.so	True	True
1232	bash	0x00007fd24e6dc000	/lib/x86_64-linux-gnu/libnss_files.so.2	False	True
1232	bash	0x00007fd24eaf3000	/lib/x86_64-linux-gnu/libnsl.so.1	False	True
1232	bash	0x00007fd24f4e4000	/lib/x86_64-linux-gnu/libtinfo.so.5.9	True	True
1232	bash	0x00007fd24ed0d000	/lib/x86_64-linux-gnu/libnss_compat.so.2	False	True
1232	bash	0x00007fd24e8e7000	/lib/x86_64-linux-gnu/libnss_nis.so.2	False	True
1232	bash	0x00007fd24f2e0000	/lib/x86_64-linux-gnu/libdl-2.19.so	True	True

Figure 41 Output of the *linux\_ldrmodules* showing all the injected libraries and their categories

The Kernel column indicates if the library is loaded into kernel memory space. Libc on the other hand presents a collection of libraries loaded into the memory space of the standard C library. As per the explanation at the beginning of this part of the analysis, dynamically injected (or shared libraries) can be recognised by the *.so* extension, which means “shared object”.

- *libc-2.19.so* – a collection of functions that are commonly used by programs written in C. These functions include things like input and output, strings, memory allocation and much more.
- *ld.so* – known as linked or loader. This library is responsible for loading and linking other libraries
- *libnss\_files.so* – the library that provides an implementation of the Name Service Switch. This provides an interface for many other system functions that need to look up information about users, groups, and others.
- *libnsl1.so* – library for network-related functions and system calls.
- *libtinfo.so* – library providing a wide range of functions for working with the terminal.
- *libnss\_compat.so* – provides the backend for the Name Service Switch.
- *libnss\_nis.so* – another implementation for the Name Service Switch for information lookup in the NIS servers
- *libdl.so* – library to perform dynamic linking

## 5. Recovery of the first shared library in the /bin/l

To recover the first shared library the same steps as in previous chapters are performed, looking for *ls* commands in the bash history on the infected memory dump.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_bash | grep "ls"
Volatility Foundation Volatility Framework 2.6.1
2162 bash 2021-10-27 16:52:26 UTC+0000 ls /home/
2162 bash 2021-10-27 16:54:23 UTC+0000 ls
2162 bash 2021-10-27 16:54:27 UTC+0000 ls
```

Figure 42 ls found in the process id 2162 in the bash history

Now to list the libraries in the process, the `linux_ldrmodules` plugin is used.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_ldrmodules -p 2162
Volatility Foundation Volatility Framework 2.6.1
```

Pid	Name	Start	File Path	Kernel	Libc
2162	bash	0x0000000000400000	/bin/bash	True	False
2162	bash	0x00007fc6c1754000	/lib/x86_64-linux-gnu/libnss_files-2.19.so	True	True
2162	bash	0x00007fc6c1b6b000	/lib/x86_64-linux-gnu/libnsl-2.19.so	True	True
2162	bash	0x00007fc6c255c000	/lib/x86_64-linux-gnu/libtinfo.so.5.9	True	True
2162	bash	0x00007fc6c1d85000	/lib/x86_64-linux-gnu/libnss_compat-2.19.so	True	True
2162	bash	0x00007fc6c195f000	/lib/x86_64-linux-gnu/libnss_nis-2.19.so	True	True
2162	bash	0x00007fc6c2785000	/lib/x86_64-linux-gnu/ld-2.19.so	True	True
2162	bash	0x00007fc6c1f8f000	/lib/x86_64-linux-gnu/libc-2.19.so	True	True
2162	bash	0x00007fc6c2358000	/lib/x86_64-linux-gnu/libdl-2.19.so	True	True

Figure 43 Output of the `linux_ldrmodules`

As visible in Figure 43, there are multiple libraries. The first displayed library `/bin/bash` is an initialisation of the whole bash process. The first shared library would in this case be `libnss_files-2.19.so`. This library is used by the `/bin/ls` to find out and display information about permissions, owners and more of the file.

To recover this library, `linux_librarydump` is used to dump the library.

```
(jakub@kali)-[~/Desktop/volatility-master]
$ python2 vol.py -f cw1_machine3Infected.dump --profile=LinuxUbuntu_3_13_0-170-generic_profilex64 linux_librarydump -p 2162 -D reverse -b 0x00007fc6c1754000
Volatility Foundation Volatility Framework 2.6.1
```

Offset	Name	Pid	Address	Output File
0xfffff8801f470000	bash	2162	0x00007fc6c1754000	reverse/bash.2162.0x7fc6c1754000

Figure 44 Output confirming that the library was successfully dumped

To find the md5 of the library, `md5sum` is used on the kali machine. This command calculates and outputs the md5 hash.

```
(jakub@kali)-[~/Desktop/volatility-master/reverse]
$ md5sum bash.2162.0x7fc6c1754000
58a331855d43d41978648dc38505ac60 bash.2162.0x7fc6c1754000
```

Figure 45 md5 hash of the first shared library dumped from the infected memory dump

**58a331855d43d41978648dc38505ac60** is the md5sum of the dumped library, as it can be seen in the Figure 45.

## 6. References

- [1] Z. Wang, X. Jiang, W. Cui and P. Ning, "Countering kernel rootkits with lightweight hook protection." in *Proceedings of the 16th ACM conference on Computer and communications security* (pp. 545-554), Nov. 2009
- [2] J.M. Cruz, J.P. Dias and J.P. Pinto, "A hands-on approach on botnets for behaviour exploration." in *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*, April 2017
- [3] J. Junnila, "Effectiveness of Linux rootkit detection tools.", 2020.
- [4] Ubuntu Manual pages Available at <https://manpages.ubuntu.com/manpages/trusty/man3/probe::signal.syskill.3stap.html>
- [5] Ubuntu Manual pages Available at <https://manpages.ubuntu.com/manpages/bionic/man2/getdents.2.html>
- [6] J.D.T. Botero, "Differences between static and dynamic libraries", May 2021, available at: <https://www.linkedin.com/pulse/differences-between-static-dynamic-libraries-juan-david-tuta-botero/>
- [7] D. M. Beazley, B. D. Ward and I. R. Cooke, "The inside story on shared libraries and dynamic loading," in *Computing in Science & Engineering*, vol. 3, no. 5, pp. 90-97, Sept.-Oct. 2001, doi: 10.1109/5992.947112.