University of Luxembourg
Travail de fin d'études
Bachelor of Engineering in Computer Science

Development of an automated process to execute and analyse the network behaviour of malware in a controlled environment

Laurent Weber

Academic year 2008-2009

Tutors:
University of Luxembourg : Mr. Steffen Rothkugel
SES ASTRA : Mr. Alexandre Dulaunoy

Thanks

I want to thank Mr. Alexandre Dulaunoy and Mr. Steffen Rothkugel as well as Mr. Gérard Wagener who gave me the opportunity to work on this project. Furthermore, I want to thank SES ASTRA for providing me with an office and support and for using their infrastructure. I also want to thank the University of Luxembourg for their support. Finally, I want to thank everyone else who supported me during the realisation of this project.

Laurent Weber

Declaration of honor

I, the undersigned, declare that the attached assignment is wholly my own work, and that no part of it has been:

- copied by manual or electronic means from any work produced by any other person(s), present or past,
- produced by several students working together as a team (this includes one person who provides any portion of an assignment to another student or students),
- modified to contain falsified program output, or copied from any other source including web sites.

I understand that penalties for submitting work which is not wholly my own, or distributing my work to other students, may result in penalties under the University of Luxembourg's Academic Discipline Bylaw.

Laurent Weber

Contents

1 Preface

1.1 Summary (English)

The traditional way malware uses to communicate starts to get obsolete. Nowadays, malware in form of viruses, trojan horses and worms use new technologies to communicate with each other, and with the attacker. If traditional malware communication via, for example IRC control and command centers is still used, malware programmer starts to explore other technologies, like peer-to-peer (P2P), unknown or proprietary protocols.
The first part of this report will be dedicated to the conception as well as the implementation and deployment of a framework allowing the secured and automated execution of malware, in a controlled environment.
The second part will present the analysis of the report generated by the framework and the integration of the virtual machine management framework into the already developed malware analysis framework at SES ASTRA.

1.2 Summary (French)

Les méthodes de communication traditionnelles utilisées par les malware (codes malicieux) commencent à devenir obsolètes. De nos jours, les malware, sous forme de virus, chevaux de Troie et vers de toutes sortes, utilisent de nouvelles technologies pour communiquer entre eux ou avec l'attaquant. Si les malware traditionnels communiquaient par le biais de serveurs IRC de contrôle et de commande, les programmeurs actuels de malware commencent à explorer d'autres technologies, comme par exemple le peer-to-peer (P2P), des protocoles inconnus ou bien propriétaires.
La première partie de ce rapport se concentre sur la conception et l'implémentation ainsi que le développement d'un framework donnant la possibilité de l'exécution automatisée de malware dans un environnement sécurisé et contrôlé.
La seconde partie présente l'analyse des captures réalisées par le framework et l'intégration de celui-ci dans un framework déjà déployé chez SES ASTRA.

2 Enterprise presentation

2.1 SES ASTRA

SES ASTRA operates the ASTRA Satellite System, offering a comprehensive portfolio of broadcast and broadband solutions for customers in Europe and beyond. ASTRA broadcasts television and radio programmes directly to millions of homes, and provides internet access and network services to governments, large corporations, small- to medium-sized enterprises and individual households. ASTRA is headquartered in Betzdorf, Luxembourg, from where the company conducts centralized functions such as spacecraft and payload operations, 24x7 technical assistance and corporate activities. In addition, affiliate offices in key markets in Europe provide local sales and marketing support. ASTRA is part of SES GLOBAL, a family of satellite operators and network providers which, combined, offers global reach.

2.1.1 Activities

2.1.1.1 Broadband access and network solutions

Platforms for telecommunications operators, service providers, corporations and market institutions Rapidly deployable, cost effective and secure, satellite-enabled networks are an elegant solution for businesses with remote communications needs and internet providers looking to expand their markets. ASTRA can custom-build solutions for a wide range of business requirements, from one- and two-way satellite internet access for home and SOHO users to fully-managed IP network platforms for corporations. Solutions for ISPs, telcos and corporations

2.1.1.2 Satellite IP platforms

Through its IP platforms, ASTRA offers quickly deployable connectivity for remote communications networks to suit a wide range of applications.

2.1.1.3 Internet access

ASTRA internet access services enable telcos, ISPs and other providers to offer broadband-type internet connections to home and business users.

2.1.1.4 Business content delivery

Businesses can use our IP multicast and streaming services to distribute timely information to the field with exceptional ease and security.

2.1.1.5 Content to the home

Our streaming services deliver video, audio and multimedia to the home in real-time, while our "push" offering loads content onto local storage, for consumer enjoyment on-demand.

2.1.1.6 Point-to-point IP links

Corporations with remote networks can rely on ASTRA to bridge connectivity gaps using our IP trunking services.

2.1.1.7 Cable operator solutions

If you're a cable operator, we can help you enhance your offering for subscribers with the addition of internet and radio to your service portfolio.

2.1.1.8 Mobile Solutions

ASTRA satellites can deliver downstream Internet traffic directly to small mobile antennas.
source: http://www.ses-astra.com

2.2 TFE location

The "travail de fin d'études" is done with SES Security and Communication Team - SCT. This team is responsible, among other things, for the IT Security. Huge company networks are analysed and monitored there, that makes it a great place for my project as it's related to networking and IT Security. Gerard Wagener, who is actually working there as well, has build a framework (ANNE - Automated aNalysis and Network Emulation) for his TFE at the University of Luxembourg in 2006. This framework consists of a batch analysis of malicious software so it will provide an ideal input method for my framework. This work is complementary to ANNE and focuses on the network capabilities of malicious software. The ANNE framework uses the work of the CSRRTeam,[1]which provides a method to trace, handle and collect dangerous software, so my framework will also take advantage of this piece of work.

[1]Computer Security Research and Response Team Luxembourg

Furthermore, SES ASTRA has a framework that is analysing malware based on Windows emulation [4]. The framework that we will build, will provide a good mean to compare how malware behaves on an emulated Windows and on a native one. Our framework will also extend the existing framework for a more advanced analyse of the malware.

3 Objectives

Development of an automated process to execute and analyse the network behaviour of malware in a controlled environment

Knowing the amount of different operating systems, as well as the amount of different software for those it seems quite obvious that there is plenty of different malware out there. To have the ability to build an automated process that imports, executes and analyses the network behaviour of such a software a deep operating system, programming and networking knowledge is required. The most important part of this process is of course the analyse part, but to be able to perform this part a running framework is needed. This framework should be very robust, as it contains a daemon [6] responsible for the importation, and execution of the malware in virtual machines. The fact that a virtual machine could be crashed by a malware should be included in the conception, but this shouldn't cause the daemon to crash. So one of the major tasks of the daemon will be to manage the virtual machines, to see if they are still running in an appropriated way.
Furthermore, an simple API (Application Programming Interface) should allow other processes to interfere in a comfortable way with the framework. The best way of gathering information in a reliable and comprehensible way has to be found to grant the best possible analysis.
The framework should, furthermore, be customizable in a very eased way, for further usage required by the industrial partner. The MAF (Malware Analyse Framework) will be integrated in an already existing framework, the ANNE Framework. This framework performs analysis on given malware, but nothing which is capturing and analysing the network activity of the malware. Our framework will in this way, help the existing framework to perform a deeper and more complete analyse of the malware. As this was a need by the industrial partner the possibility to integrate our framework into the existing ANNE framework had a high priority.

4 Introduction

First of all we have to explain the aim of the framework. Why is it important to analyse malware?
It is important to stay in constant contact with the evolution of the technologies used by the malware programmers, if you want to find a good way to protect your network or machines against malware.
Being successful in fighting against malware, means playing a cat and mouse game with the attackers all the time. In order to detect new technologies used by the malware, a framework like the one we will present hereafter, could provide a big advantage.
Then, we have to clarify that our framework will only be useful for malware trying to communicate with other malware or an attacker in a given time. Software operating without network connection will not be analysed by the framework, as well as software meant to be executed at a special time in the future. Malicious software presenting the enumerated properties will be imported and executed in our virtual network too, as we cannot know what the software is up to, but as it is not generating traffic it will not be detected by the logging system of the network adapters.

5 The Malware Analyse Framework (MAF)

This part describes the design and implementation of the framework created to managed a given range of virtual machines and execute given malware.

5.1 User requirements

The features our framework has to implement are the following:

- Setting up the environment for the daemon of the framework, including networking, access rights and database.
- Starting and assure that always a given number of virtual machines are running.
- Take a given malware from the database and copy it to a free virtual machine.
- Execute that malware in a virtual machine.
- Manage crashed virtual machines and not responding ones.
- High reliability on the daemon.
- Text user interface for user or scripts.
- Logging features of the network traffic generated on the virtual network.
- Everything should be working fully automatically without need of any user.

5.2 Technical requirements

As the application is engineered for a UNIX based operating system, we need such an operating system to make this framework run. At the base this application was build and run on an Ubuntu operating system. This choice was motivated by the fact that we wanted to have optimal stability for the host and that we like open-source software. The whole framework is build using open-source technologies except the kvm module used to activate hardware virtualization for QEMU [1]. This module is proprietary, unfortunately, and of course the Windows image used to execute malware on it is not open-source at all.
Furthermore, the whole framework was implemented using a standard python. No modules from third parties had to be included, this choice was also performed in order to have maximum stability.

5.2.1 Setting up the Windows Image

A virtual machine emulates a hardware machine with the associated hardware. In order to have an operating system on that emulated machine we need an image file of the operating system. An image is a file containing the raw data that is stored on the hard disk of the machine.
The framework needs a prepared Windows image in order to operate like we wish. There have been as little modification to the standard virgin Windows image as possible to keep the results of the analysis as less tainted as possible.
First, as the daemon should be controlled remotely, it needs some way to interact with the network. Therefore we decided to install a SSH server. SSH runs on Linux therefore we used a Cygwin[1] environment which provides a working environment to execute *unix designed programs in Windows.
Then, the name of the local network has to be changed from "Local Area Network" to "LAN" this was done to avoid any problems related to spaces in the name of the network adapter.
Furthermore, the image has to be setup with an task scheduler entry to start a script, called *changeIP.bat* on startup of the machine.
This batch file gets copied in the Windows root folder *C:* before the booting of the machine. It is needed as each Windows image has the same IP address after the copying and this results in IP address conflicts, as we use many virtual machines running in parallel. The main goal of that file is to change the IP address of the booting machine to the IP address contained in the batch file.
This batch file contains a netsh command to setup the network environment, the command is the following:

netsh interface ip set address name="LAN" static "10.111.111.23" "255.255.255.0"

This sets the IP address to *10.111.111.23* of course this is only an example and the netmask to *255.255.255.0* these options have to match our virtual network. All IP addresses can be used, except the *10.111.111.1* (reserved for the bridge) and *10.111.111.2* (reserved for the tap1 device) and *10.111.111.254* (the default gateway address used by the User Mode Linux).
The script is assigning the IP addresses in a automated way and taking care that there are no duplicated IPs on the network, and it writes them in the database.
Finally, the Windows firewall has to be turned off so that an host can ping another host in the same network.

[1] Cygwin is a Linux-like environment for windows, it provides, for example an OpenSSH Server. Official web presence of the project: http://www.cygwin.com/

5.2.2 Modifying sudoers file

Another thing that is needed for the framework to run is a modified sudoers file. The user has to have the rights to execute some commands he usually is not allowed to, for example setting up tap devices. All these actions are performed in the startup bash script, and the privileges are dropped and keep to the strict minimum. The fact that the user is allowed to execute only a few commands seems to be ideal for our purpose and avoid an improbable but possible compromising at root level. The user should be allowed to perform the following tasks:

- /sbin/losetup: To be able to set up and down loop devices.
- /bin/chmod: To be able the change the properties of the files.
- /bin/chown: To be able to modify the owner of the files.
- /bin/umount: To be able to unmount devices.
- /bin/mount: To be able to mount devices.
- /bin/mkdir: To be able to create devices.
- /usr/sbin/brctl: To be able to create and delete networking bridges.
- /sbin/ifconfig: To be able to set network interfaces up and down.
- /usr/sbin/tunctl: To be able to create and delete tap devices.
- /bin/mknod: To be able to create new nodes.
- /sbin/route: To be able to configure new routes.

5.2.3 The database

To have the ability to trace back what machine was started when, and with which properties we use a mysql database. This database is a central part of the framework as every process running a virtual machine uses it, as well as the daemon instance.
The database contains two tables:

- The manageVM table, used to manage what images are ready to use and which have been used. This table contains the fields: id, status, and the time the entry was done.
- The managing table, used to store start time of the machine, ip, mac, vnc, the tap device to use, telnet port, the status as well as the path where the malware is located and the folder where the results should be stored. The id of the virtual machine used, this is taken from the manageVM table, and finally the starting time of the execution of the malware.

On one hand, the daemon uses the last entry of the managing table to compute the next IP address, VNC port, telnet port and mac address.
On the other hand, heavy usage of the database is done by each process, the status has to be updated according to the status of the machine running in the process. How the status are distributed in the manageVM table can be found in the following table:

Status	Requirement
FREE	When the image is ready to use.
USED	When the image is being used or has been used.

Figure 5.2.4.1: The manageVM table status distribution.

The distribution of the status of the managing table can be found here:

Status	Requirement
TODO	A malware gets copied into the database but no machine is ready.
BOOT	The virtual machine is beeing booted.
FREE	The machine is running, but there is no malware.
RUN	The machine is running and the malware is being executed.
UP	The machine is running, and waiting for malware.
STOP	The machine has run a given time and was stopped normally.
KILL	The machine had to be killed due to a problem.

Figure 5.2.4.2: The managing table status distributrion.

The status are set when they reach different parts of the code. For example the "TODO" is set immediately after the Text User Interface commits a malware, or the status "UP" is set as soon as the machine responds to a ping request. "KILL" is used when the machine was killed due to a fail during booting or during the setting up of the logging mechanism. "STOP" is used if the timeout time has been reached. This timeout is used to properly shutdown machines after a given time, as malware mostly act immediately after having been fired up the execution of the code is done very early so the traffic is also generated early in the uptime of the machine.

5.3 Getting started

Fist of all, we need to be sure that we always have images ready to use, to fire up our virtual machines. This is guaranteed by a simple script, described in section 5.4.9 The daemon script should now run until it gets stopped by the user. Despite this, no interaction with the framework is needed and the logs should grow automatically when malware is executed and it tries to connect to the internet or perform other network related works, for example network or port scanning.

5.4 Architecture

Here we'll present the architecture of the framework. The framework is build in different parts, acting together as if they were one.

5.4.1 The concept

The whole framework is using an host - guest infrastructure. This choice was performed in order to have a safe infrastructure to execute a malware. The idea is not to spread malware and help it to propagate and infect other machines, the malicious software should stay in our sandbox.
On one hand you have the host system, that is the operating system running on the physical machine. This system should never be compromised by malware, and stay clean and operational.
On the other hand, there are the guest operating systems, which purpose is to provide an environment to execute and analyse malware.
These guest operating systems are run on the host in so called virtual machines. There are several virtual machines software available, for example Qemu, Xen, Virtual Box and VMware. For our needs we decided to use Qemu, as this software is able to run with user privileges only, so provides us with a high security level as, even if the virtual machine gets compromised by the malicious software, it can't gain administrator rights. Another very good point of qemu is that it can be easily scripted, this means that you can easily use scripts to set it up and manage it. This would be much harder using some other virtual machine.
Qemu is able to run without any graphical user interface, so you can use it without any problems on a server. This doesn't mean that you cannot see whats going on on the machine. Qemu allows you to set it up in a way that it is using a VNC server. Through this server you can connect to the virtual machine and see what is going on the machine as if you were sitting in front of it. If used on a server you need to setup X-forwarding to be able to use it.If you don't want that you also have the possibility to connect through telnet to the monitor of qemu, and using this method you can easily use special commands to drop memory or make a snapshot.
Furthermore, a very positive point is that Qemu can use hardware acceleration if you have the adequate hardware. On the Thinkpad T60 or similar hardware like T61, which has been used for the design of the framework Intel VMX support was given, and worked fine after having been activated in the BIOS. This was great advantage for the implementation and debugging of the framework, as it was no problem to start 3 instances of Qemu at the same time, which would have been impossible without the kvm module.
Another very good point for Qemu is that it doesn't need any modification at the host kernel level, which allows an eased deployment on different machines.
Finally, we also used User Mode Linux (UML) to have a sort of virtual machine running

a Linux operating system. We choose UML as it is faster then Qemu and designed to run Linux based operating systems. Like Qemu, it can be run entirely as user, which make this piece of software pretty attractive for our purpose.

5.4.2 Life cycle of a malware in our Malware Analyse Framework

In order to permit to the reader to fully understand what we are talking about, we will now present a graphical, high-level representation of each step a malware runs through. Therefore we start at the beginning, at the Text User Interface, and walk on through the rest of the framework until the capture is finished. As said, this is a very high level explanation and we will discuss the single sections more in detail later in this work.

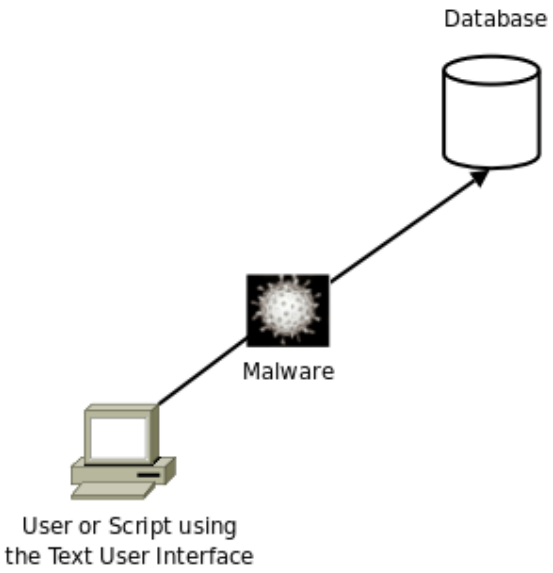


Figure 5.4.2.1 User or script submitting a malware to the framework

On this representation you can see the user, or a script using the Text User Interface to submit a malware (represented by a picture of the biological Rotavirus) to the database of the framework, where the malware gets stored in. This database is used as a queue where threated and un-threated malware is stored.

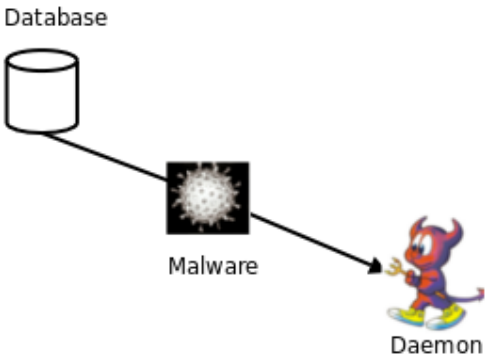


Figure 5.4.2.2: Daemon taking an new malware out of the database.

This figure shows what happens next with the malware. It get taken out of the database by the daemon and then analysed further on, this will be discussed on the following lines. We decided to use the BSD mascot daemon picture[2] named Beasti as a graphical representation of our daemon. The fact that the daemon is running should make clear that the daemon should run forever.

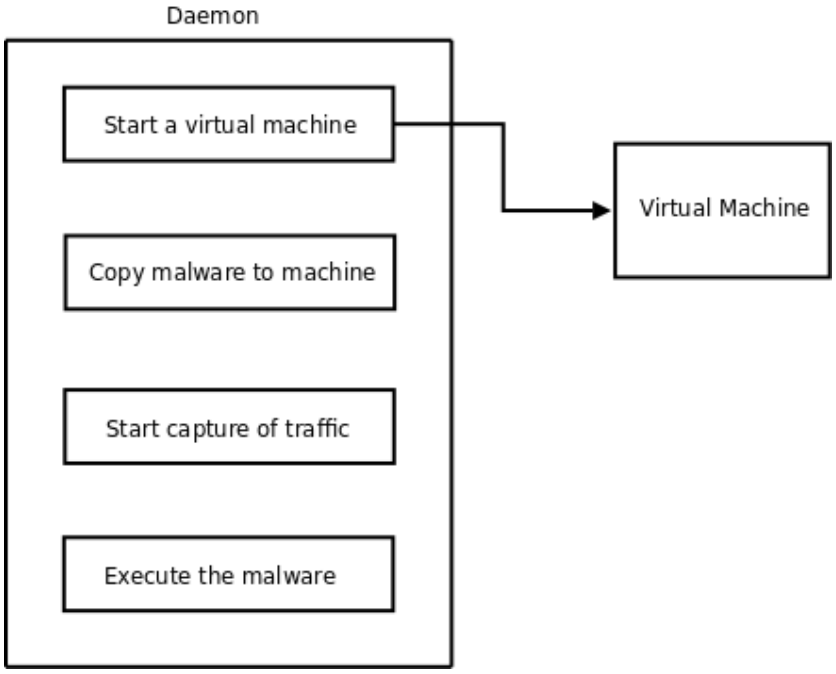


Figure 5.4.2.3: Daemon starts a virtual machine.

As soon as an un-threated malware is found in the database a new virtual machine is started by the daemon. Why we start a new machine for every malware is discussed in

[2]BSD Daemon Copyright 1988 by Marshall Kirk McKusick. All Rights Reserved.

the next section.

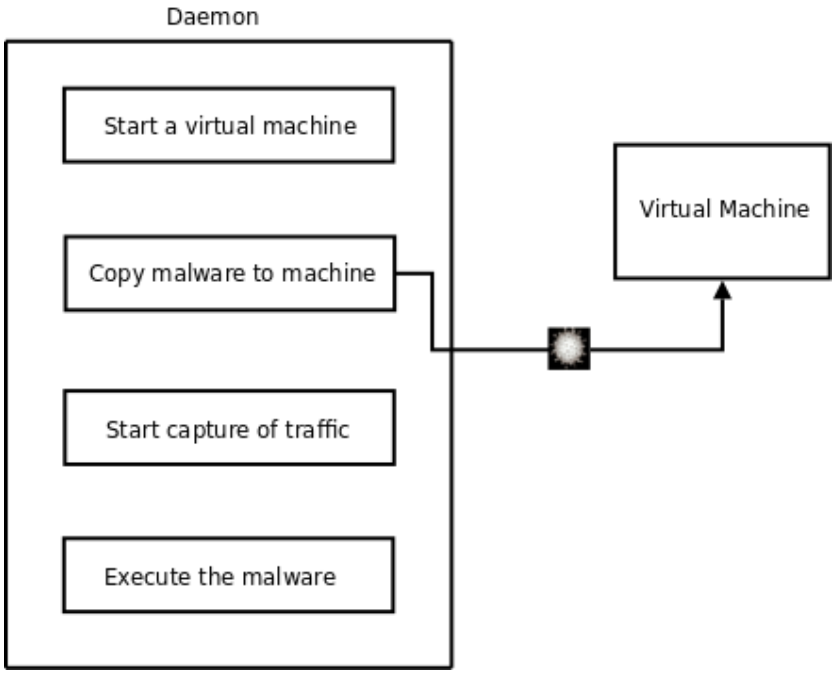


Figure 5.4.2.4: Malware is copied on the machine.

Once the machine has been successfully started, we copy the malware over the network to this machine.

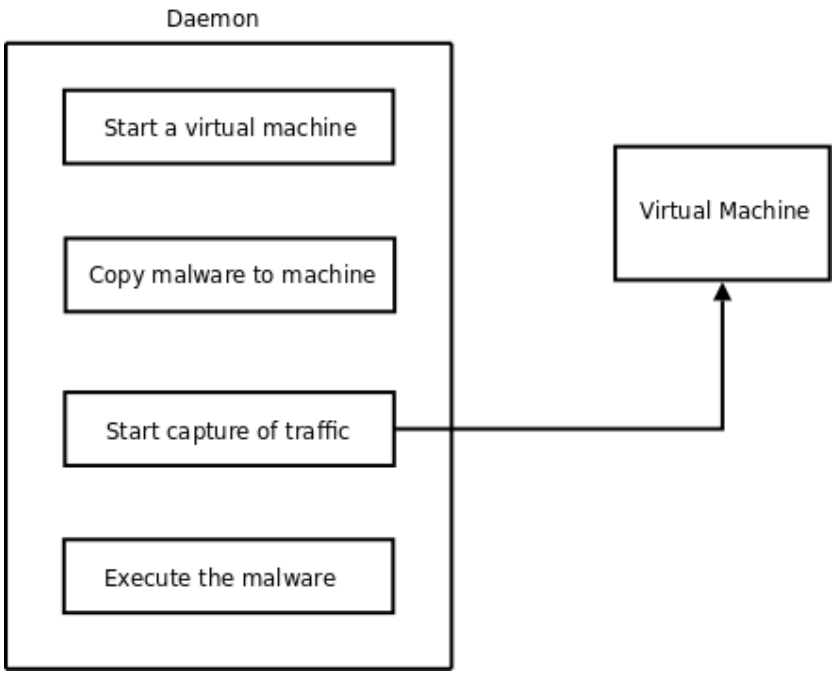


Figure 5.4.2.5: Capture is started.

After a successful copy the capture of the network traffic is started on that machine. Now every communication with the network is logged.

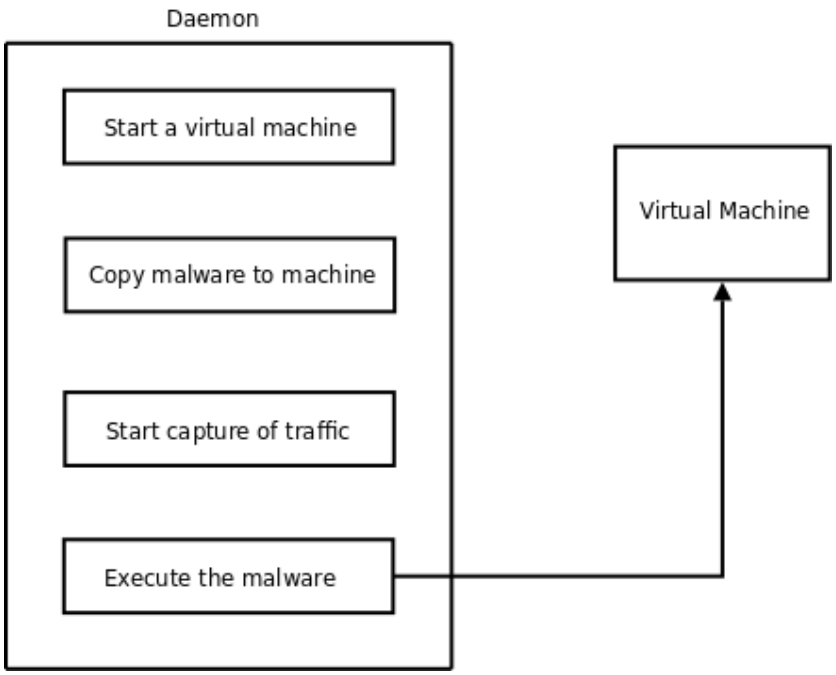


Figure 5.4.2.6: Malware is executed.

Finally, the malware is executed on the virtual machine by the daemon. Everything that gets logged by the network capture is most probably related to the malware.
Of course, the way the framework works in detail is a lot more complicated, and we will explain every action in detail in the rest of the document, for example why we first start the capture and execute the malware only after that, or which information we find in the captures that are not related to the malware and why these information are contained in the logs. As said, this was just a very high level representation of the life cycle of a given malware in our framework. Now we will present the more technical part.

5.4.3 Using one malware per virtual machine

Copying images is certainly the part that takes the longest in the whole execution of the framework. In this section we will explain why we don't simply execute all the malware on one, physical machine, or on some images and why we prefer to use a virgin image for every malware.
The main idea related to the fact that we only execute one malware on one virtual machine is that in doing so we can determine what malware does what. You could for example dig into what does a malware did on the file system. You are then sure that only this special malware was run on that image. Hence, it must be that malware that changed the image.
Another point is that if you would execute more than one malware on a machine you

could not know exactly what malware is doing what, as it could be possible that they both try to connect to the Internet. Using one malware per image lets you refine the analysis on each malware.
Using a physical machine to execute malware on it is not possible because we would have the same problems as described above. So in order to have an optimal solution we use a virtualized machine and a new, clean operating system image for every malware.

5.4.4 GNU Screen

Our application makes a lot of usage of GNU Screen, so it's mandatory to know what GNU Screen is and what it does.

5.4.4.1 General description of GNU Screen

GNU Screen is a terminal multiplexer. It provides virtual terminals in a single window. So multiple applications can be started with a single login, of course you can also have an interactive shell. So in our case we will start everything that is related to the framework in one window, the SSH script, the script copying images, the User Mode Linux, the tcpdump and the script starting the virtual machines. Each of those will get a terminal with an own name, so that you can easily recognize what window is doing what.
Another very interesting feature of GNU screen is that you can attach or detach sessions. So when you have started the screen session with your terminals, you are able to simply detach the session, so it will go to the background, and you get your prompt back. This is very useful when you are connected through a SSH connection to a server, where the framework is actually installed.
Later, you can reconnect to the server, at any moment an reattach the GNU screen session and you can continue your work. So in our case you can see the progress of the framework while you were doing something else. This is possible as the framework works in a fully automated way.
We have written a screen class that handles everything related to GNU screen. Creating a screen session, adding window to that session and naming them, closing window after usage. This is all done with the following commands:

screen -L -S screenName -m -d

This starts a screen session and the *-L* enables the logging. Everything happening in the screen is logged in a file that is stored in the present working directory and named *screenlog.0* (more on logging in section 5.4.6). The *-S* gives our GNU screen session the name *screenName*. In our framework we name the session *framework*. Finally the *-m -d* flags define that we want to start the session in the background, so it is started in detached mode. If we want to know what is going on in the screen session we have to

reattach to it, this can be done with the command *screen -r*. If we do so, we have an empty GNU screen session. So, how do we start applications in this screen session? In our framework we do it with the following command:

screen -S screenName -X screen -t winName prog

where the *-S* flag is used again, to define to which session we want to add our application, this is useful as it could be possible that we are running more screen sessions on the machine. The *-X* flag tells us that we want to execute a command. As a parameter of this command we give *screen -t winName prog*. This means that we want to start a new window with the application *prog* and with the name *winName*. This is very useful for later, so with a good chosen window name we immediately know which window is running what when we reattach to the session.
So now we are able to start GNU screen sessions and to add windows to this screen. This is good, but imagine you run the framework and come back later, then you will have plenty of useless windows, as the execution of the code is over, but the corresponding window not closed. This could happen for example after the pinging of a virtual machine. So we have to close windows after the execution of the code, after a successful ping reply for example or after the tcpdump capture is over. This has been implemented too, with the following command:

screen -X -p winName kill

This command executes a command, with the *-X* flag. The *-p* parameter is used to indicate that we want to apply the command to the *winName* window. Finally the command to be executed is *kill* which closes the window.
We had plenty of problems related to this GNU screen idea, this was caused by a bug in the GNU screen version hosted on the official Ubuntu repositories. After compiling the GNU screen version from the official git repository[3] of the project all the problems were magically gone.
Different reasons why we chose GNU screen instead of forks are discussed in section 7.4

5.4.4.2 The technology we use

The following section will explain why we chose to use GNU Screen and what advantage it provides us.
GNU Screen is just a tool we use in order to reach an asynchronous engineering technology. An asynchronous application is per definition non-blocking, this gives us many advantages. No process has to wait until the other finishes its execution. So, if even if a process crashes this doesn't affect the whole framework as only this one single process will crash, and not the others.

[3] git clone git://git.savannah.gnu.org/screen.git

Another advantage for this asynchronous version of the framework is that you can, without any difficulty, deploy it in a distributed computing way. You could image to use the different processes on different machines to speed the analyse up on huge data sets. A non-blocking approach gives you this flexibility, where a blocking approach would not allow you to do so.
An integration of the framework with a [5] mapreduce-like framework could also easily be realized as there are already libraries for python available.

5.4.5 The Configuration

All the configuration is done in one single configuration file. We chose to make only one file, even if different scripting languages are acting together, as it's easier to keep an overview.
The configuration file has a strict structure explained in detail later in this section. Anyway, the user has not to worry about how the configuration file is written, as an interactive configuration script is writing the configuration file for him. This means that the user has to answer some questions, and then the configuration file gets created by the script. This allows us to control first if the input given by the user is correct. Assuming he tells the configuration file creating script that he wishes to use */home/kabel/.ssh/malware.rsa* as a certificate for the SSH connection and there is no such file, the script will block until he gets a valid certificate as input. This control is done on several inputs, and allows us to filter out typing errors in a firstly, but also some errors related to user permissions. If the user forgets that he is not allowed to write to a folder he gets a warning too. This allows us to have a proper configuration file, that our scripts understand and that is, at a very high certitude, working. The configuration script proposes default values for all the questions, these answers should be the right choice for most of the cases, so that the user only needs to hit enter to come to the next question. Such a question with an proposed answer is:

Please enter the size (in Bytes) of the Image you want to use:[5368709120]

Here the proposed value is: 5368709120. The user can take this value by hitting return, or, if he has another image than we have, he can type in his value and hit enter after. If he does so the new value is written to the configuration file. Of course the user can also change the configuration file, located at */etc/framework/virt_conf.cfg*, manually but then it's more probable that an typo or another problem infiltrates our configuration file.
The configuration file builder is called each time the daemon is started, but if you already have an valid configuration file you can skip the part related to the creation of a new configuration file. This can be done by answering *"y"* when the application asks the following question:

Do you have already a configuration file and you want to use it?:[y/n]

As our application is highly configurable the configuration file has a certain length. We tried to keep the file as short as possible but long enough to keep all relevant information to make the application work on any computer and as any user. In order to keep the configuration file human readable and understandable we split it into different parts
We will now present a sample configuration file:

[DB]		
host	=	localhost
password	=	pulpfiction23
user	=	honeybunny
database	=	status
[Folder]		
cleanimage	=	/home/honeybunny/cleanImages/xpSp2Admin3raw1.iso
numberofcopies	=	10
infectimages	=	/home/lweber/infectedImages/
sizeofimage	=	5368709120
mountpoint	=	/mnt/
reserveimages	=	/home/lweber/reserveImages/
[Misc]		
debug	=	0
path	=	/home/lweber/virt_logs/
nbvm	=	2
timeout	=	10
[SSH]		
rsakey	=	/home/lweber/.ssh/qemu_rsa
[tcpdump]		
filter	=	src not 10.111.111.1 and dst not 10.111.111.1

Most of this is self-explaining. The first part, contains the database related part: Host where the database is located, password and username of the database user as well ass the name of the database we want to use.
The *Folder* section handles everything related to the images and their location: The absolute path to the clean image, the number of copies of that image that should be done (at the beginning), where the copied images will be stored, how big the image is (in bytes). (This size indication is used to check if the image is not corrupted before we use it. Generating a hash for the image would be even more reliable but takes way too long on large files.) The mount point where you want to mount the images that will be used. The reserve image folder, the folder where we want to store the copies done by the image-copy script described in 5.4.9.
The *Misc* section handles everything that doesn't fit in a specific category of the configuration file. So the debug level is stored here, as well as the folder where we want to store the logs of the framework. A very important information is the number of virtual

machines we want to be run by the framework, this value should be adapted to the capabilities of the machine the framework is running on. Finally, this section also contains the timeout value, the time how long the machine runs before it gets stopped.
The *SSH* section simply contains the path to the rsa-key needed for everything that is related to the SSH connection between host machine and guest network.
The *tcpdump* session contains the filter used by the tcpdump program. The default value used by the framework is to discard everything that comes from and goes to the tap device connecting the network to the host. This is done in order to get rid of the SSH traffic generated by the SSH connection we establish to execute the malware.

5.4.6 Logging

During the working on this application we have found out that logging is one of the major needs for this framework. The logging of the network traffic, which is the main goal of this framework, is one kind of logging that has been implemented, but that's not the only logging that is done in the framework. Indeed, we have to keep track of what happens when in the framework. We have to know exactly which process did what, and when it did it. This is certainly important in case an error happens, but not only. As described in section 5.4.4 we can detach a session at anytime, and reattach it at any time. Then we have to know exactly which process did what and when, in order to be able to checkout the results as fast as possible.
Furthermore, as the framework will be implemented further on, it is important for the next developer to have good debugging messages in order to be able to understand what is happening how and when, and most notably why. This might sound weired, but remember that everything is running in parallel. Many machines are set up and execute malware in parallel, all the time. Given these facts, the user or programmer might loose the overview. Therefore we mainly implemented three kinds of logging:

- Logging to a file, here we use a syslog like syntax to write messages down, here an example of an error message:
  Sun, 24 May 2009 22:54:42 ERROR IN: pingVM There was a problem in pinging a VM
  Sun, 24 May 2009 22:54:42 DEBUG IN: mvImage Ping is False - Killing machine.

- Logging in the GNU screen windows directly. Some debugging messages are written to the matching window in the screen session. This allows the user to be informed immediately about the advancement of the process, so for example the responses of pinging are always written down in the according window, then the user knows that the process goes on and can figure out when the ping requests got an answer.

- Screenlog. The GNU screen tool comes with the capability to log everything that happens in a session. We use this feature of the tool too. Even if mostly we

discard these logs, they might be important the day we get serious troubles with the framework. Due to these logs, we are able to understand what went wrong, and why we didn't get the result we wished, even if we reattach the screen session, where the problem occurred, a few hours or days later. Thanks to these logs we can go through the whole process and be sure to find the problem and then try to fix it.

5.4.7 The Bash Script

The first script that needs to be run before you can use MAF (Malware Analyse Framework) is a bash script. This bash script takes care of the setting up of the whole framework. It sets the rights on devices to a minimum and creates tap devices. This script needs to be run as a root user. It is the only part of the framework that requires root privileges, afterwards a restricted user can perform all the tasks, with one condition, that this user has some entries in the sudoers file, as described in section 5.2.2.
This script also launches some python scripts, like for example the interactive configuration file builder, which is discussed further more in section 5.4.5. Tasks that this bash script should perform are the following:

- Clean everything up from the last run, for example it could be possible that the application crashed.
- Setup the virtual network.
- Change the rights on the files, so that everything can be run with user privileges.

Here is a graphical representation on how the script works:

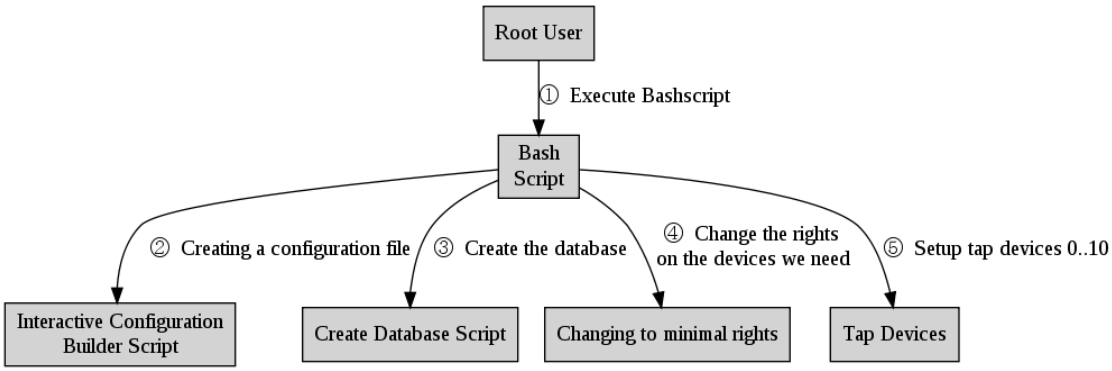


Figure 5.4.7.0.1: The script preparing devices for the framework

5.4.7.1 The cleaning up

For an optimal control of the environment we have to know exactly how our operating system is setup. As we make strong usage of the networking capabilities we have to be sure that the virtual network is up and running before we start to use it.
So we make sure to clean up everything we need before we start the main daemon process. This is done each time the daemon is started, also just after a reboot. We prefer to clean up twice rather than to have problems due to an unclean environment. The rationale behind this approach is that the daemon may have crashed before and was not able to clean up the environment. This task, which is crucial for the daemon, is performed by a bash script. It searches for tap devices that are up and shuts them down and after this it tries to remove them from a bridge, if such one exists and finally destroys the bridge.
The mounted devices are also an important point. As our framework mounts and unmounts constantly images to folders of the given mount directory we have to make sure, that before we start this directory is clean, and that there are no mounted folders from previous runs, else we could , in certain circumstances, encounter problems. This is done by the daemon when it is started. Here we assume that the user is a responsible user and that he takes care where he mounts his directories in order to not destroy other users mounted devices, even if the script is written in a way it should only unmount and delete the folder it uses itself, something could go wrong or a user could choose the same folder name as the framework does.

5.4.7.2 Setting up the virtual network environment

As the virtual network is the heart of the framework we investigated this part in a very detailed way, and tried a lot of different configurations.
First of all, we tried with vde, which is a wrapper for QEMU to connect to a virtual network. It even provides a virtual switch, called vde switch which does a great job. Despite the easy setting up we had problems to connect User Mode Linux to it, and we decided to give it up. Furthermore, vde switch had to be run as a root user, which is not a good idea if you execute malware, this could lead to a compromisation of the host operating system for example if a malware would be able to execute code through a buffer overflow in that piece of code, then it would have root privileges on the host.
Secondly, we tried to connect the different virtual machines through tap devices and bridges, which seemed to be more fault tolerant, and if there was a problem it was easier to spot the error than with the tricky and not very transparent vde switch. So we took this approach to provide us a reliable network that could be setup by scripts in an eased way. In opposition to the vde switch the tap devices only needed to be setup as root, but then were run as a normal user, which would dramatically reduce the effect of a compromising.

The work of the script is to create ten tap devices and to put them together on a bridged interface. In order to be able to access our virtual network from the host operating system we had to give an ip address to one of the tap device. So our scripts gives the ip address *10.111.111.2* to a tap interface. This allows us to connect from our host operating system to the network. Furthermore through this configuration of the network each machine can access the other if that is wished, and the host has also the ability to access the virtual machines which is a mandatory property for the proper operation of our daemon.

5.4.8 The User Mode Linux Image

The main goal of the whole framework is to observe malware networking, they need to communicate with the attacker or with other malware, so virtual machines should be able to communicate between each other.
User Mode Linux provides us a possibility to do that with a very low execution overhead. Furthermore, it allows us to play with malware without having to fear that our host system could be compromised, as it runs a kernel as normal user, so it is like in a sandbox. We use the User Mode Linux as a gateway for our framework. It is a standard Ubuntu without any modification, except that it has been setup with a static IP address, the IP address *192.168.1.254* and of course the netmask has to be adapted too.

5.4.8.1 Adjusting the access rights

Assuming that these virtual machine will deliberately be infected with malware we have to take care that the permissions of the resources don't allow any malicious software to break out of the sandbox and attack our host or/and the Internet.
To give the malware as less chances as possible the setup script also changes and limits the rights on certain devices, for example */dev/net/tun* or */dev/kvm* to the minimum they need to run.

5.4.9 Copying Images

During the implementation of the framework, we encounter certain troubles related to the speed of execution of the framework. Indeed, copying the images just before using them made took more than 5 minutes. Of course as more machines were running and copying their image at the same time the longer this period got. All in all, if you do a capture of 10 minutes, that makes 15 minutes for every machine.
Soon we found out that this was unacceptable. First it made the debugging process a real horror as you had to wait very long until you saw a result, and in that time your PC was unusable as the CPU load was way to high. Secondly, one of the idea of the framework was to be fast. It should not last 5 minutes before you could execute the malware, as, for the nearly 8000 malware we have here this would last 80 000 minutes. Which represent

27 entire days, assuming you only run one machine, and that the daemon never crashes, and this is only to set the machines up, no capturing time is considered here. This is just way to long. So we decided to write a script that copies images in the background all the time, so at the end when an image is needed immediately it is ready to use and you can execute malware on it in less than 20 seconds, which is, of course a big difference to the 5 minutes mentioned above.
This script makes only sense if you don't restart the daemon all the time. If you start the daemon and then you let it idle, then it's a very good idea, but else you don't make a lot of win as each image will be taken as soon as it's ready to use. As in future, the framework will be included into the ANNE framework, which hangs on an honeypot that is capturing malware once in a while, the strategy chosen here, seems to be the best appropriated. Because then you can be sure that when a malware is caught, it can be immediately analysed by the ANNE framework and then our framework will analyse the network capabilities of the malware on an ready to use image in less than 20 seconds after having got the job. So you will have the results of the analysis in a very short time period. Here is how the script is working:

The script notifies the daemon that an image is ready to use through the database. In fact, it inserts a new row in the manageVM table which has as a status "FREE". As this script is also started into a screen window it prints out some information each time an image has been successfully copied. This is, like explained in section 5.4.6, to allow the user to follow the progress of the process.
This script works in a way that it copies an image after another until it reaches a given *maxImages* number defined in the configuration file, then it simply makes sure that there are always at least a given *minImage* number images present. Once this *minImage* number is reached it restarts copying images, it does this forever.

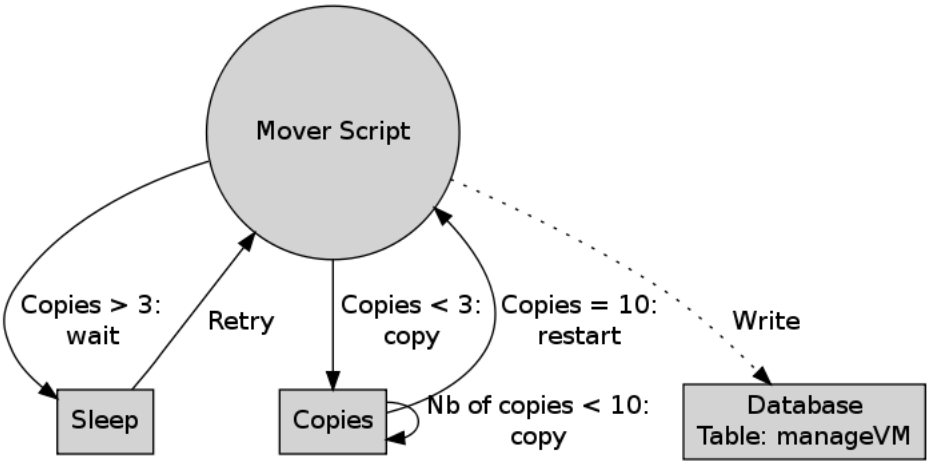


Figure 5.4.8.1: The script copying images in the background

The framework will start a machine as soon as an image is present and the number of virtual machines running is smaller than the number of virtual machines wished, as defined in the configuration file. So we could run the malware on a machine within seconds. This is, no doubt, a very good point for the framework.
The script copies the clean image from the */home/USER/cleanImages* to */home/USER/infectedImages* (where *USER* is the user name of the user executing the script.) Except if the user changed the default values in the configuration file, or during the interactive configuration file builder.
After this has been done the images are ready to be used, only the images from */home/USER/infectedImages* are used, the */home/USER/cleanImages* stays clean and will not be used to execute malware on it. This image is the base image, it is prepared as described in chapter 5.2.
The script will use the images when an image is ready and the wished number of virtual machines is not exceeded. It will start a GNU screen window for every virtual machine with each of the images, so that the machines are independent and can not cause the daemon to crash. The only way the daemon and the virtual machines are communicating is through the mysql database described in the chapter 5.2, and of course the short SSH communication described later in this section.

5.4.10 The Text User Interface

First of all, you have the text interface, on one hand giving you the ability to use the whole framework as standalone program, making it possible to start a given malware without having the need to modify any line of code. On the other hand you can use the framework from a script, like it will be used later on. The ANNE framework will interact with our framework and use it through the text interface. So this user interface gives a lot of flexibility to the whole project.
The user interface is not only used to insert malware in the database which forms the input queue, but also to query the database to find out what the state of the different machines are. When submitting a malware to the framework the text user interfaces gets an id back. This id can be used to query the database at any moment. So a constant interaction between the text user interface and the framework must be possible. This is realized through the mysql database, and the *-s* flag of the text user interface.

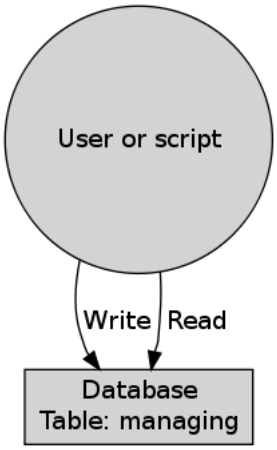


Figure 5.4.9.1: The Text User Interface in the framework

5.4.11 The daemon

Secondly, you have the main part of the framework, the daemon. This daemon has to take care of a lot of things, we'll enumerate them here and then explain how we realized the implementation of the whole daemon and which problems we encountered. The daemon has to take care of the following actions:

- Start a User Mode Linux virtual machine, used as a gateway.
- Start a given number of Windows virtual machines, used to execute the malware.
- Take care that the machines are running properly and stop them if they don't answer for a to long time.
- Import the malware from the database entry performed by the Text User Interface into the virtual environment.
- Store the results of the network traffic analysis in the folder written to the database by the Text User Interface.
- Execute the malware in an unused virtual machine.
- Take care that there are always a number of giving virtual machines running in parallel.
- Update the database to have a constant possibility to know the state of each virtual machine.
- Run the whole process forever.

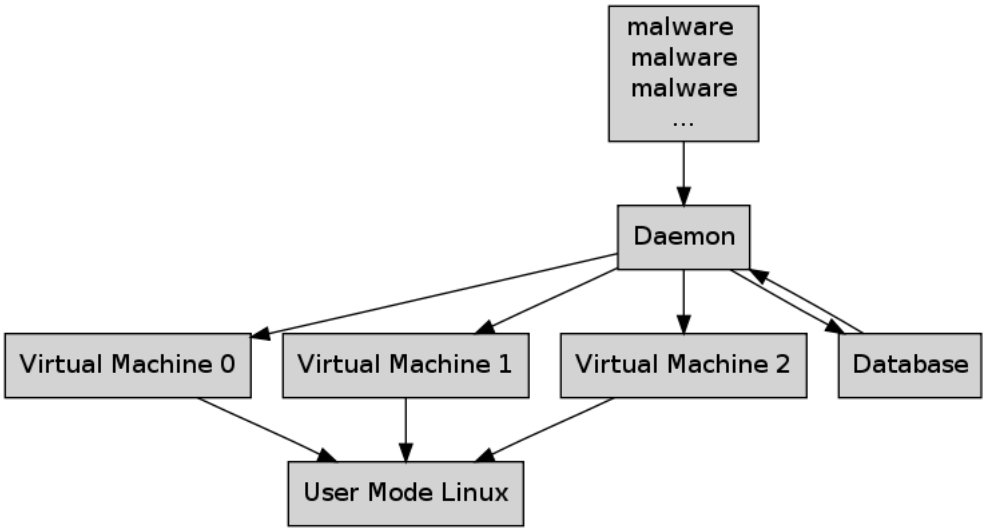


Figure: 5.4.9.0.1: The daemon

5.4.11.1 Starting a User Mode Linux as gateway

As we don't want the malware to access the Internet, neither our own enterprise network, we have to isolate it entirely from the other devices. But how to test what a malware wants to do in relation to network if we don't provide it with a minimal network infrastructure? So, to have a little virtual network we have also a virtual default gateway: as a gateway we use a User Mode Linux running Ubuntu intripid. This gateway is set up automatically to every virtual machine that is booting. Further details regarding how this process is done will be described in the 5.4.11.2 subsubsection.
The default gateway is set up in a User Mode Linux and not in a QEMU, because of the CPU load. A User Mode Linux needs less resources than a QEMU virtual machine, and as it's running Linux anyway, we think it's a good choice, because resources are as so often the limiting factor.
In a first period this gateway gives the malware the possibility to feel like in a real network, except that there is no Internet connectivity. Malware performing network scans will detect other machines and also be able to talk to other, similar malware running on another virtual machine. You could imagine a Peer-to-peer networked malware exchanging commands this way, or agreeing on a specific time, when to attack a victim.
The default gateway could run any type of services, for example an IRC server or an FTP server. For our tests we only used a fake DNS server[4] to log all the DNS requests.

[4]Mini Fake DNS server from: http://code.activestate.com/recipes/491264/ adapted and modified to fit to our needs

5.4.11.2 Starting a given number of Windows virtual machines

The Microsoft Windows virtual machines are the victims we expose to the malware. They are run on QEMU basis and are prepared like described in the section 5.2.1. At the launch of the daemon a parameter has to be given, holding the number of virtual machines that should be used. This number is written in the configuration file. Depending on the hardware, this could be more or less, but we wouldn't exaggerate as running to many virtual machines could lead in a very slow processing.
Once the text user interface launches the daemon the virtual machines start to be setup. The daemon starts as many GNU Screen window as there are wished virtual machines, so each virtual machine has it's own GNU screen windows and if a malware brings a virtual machine to crash it doesn't affect the whole daemon, but only this screen window dies. A lot of malware uses mechanism to detect virtualization software; if such an environment is detected it tries to make it crash [2].
Using an sqlite3 database at the beginning caused us much problems due to concurrent writing accesses to the database. After long implementation tests using system V semaphores we decided to rebuild the application to use a mysql database, was quickly done as all the sql-related actions were done in a single class. After this change we didn't encounter any problems anymore. This is due to the fact that mysql is a whole Database Management System (DBMS) that handles concurrent requests by it self, in comparison to sqlite3 which doesn't implement any logic to handle concurrent writing access to the tables it holds. The daemon reads the last given IP address as well as vnc port, telnet port, mac address out of the database and increases it all by one: those are the details used to start a virtual machine.
To make sure that each virtual machine has a different mac address and ip address the mac address is given in the start command of QEMU, and the IP address is written in a batch file. To make this batch file used by the virtual machine the image has to be mounted as loopback device, with the following command:

sudo mount -t ntfs-3g image mountPoint -o loop,offset=32256,force

and then copied in the root folder, where *image* is the image we want to use for the virtual machine and *mountPoint* is the mount point for the image, defined in the configuration file. After this the loopback device is unmounted again and used to start the virtual machine. This machine executes the batch file when Windows boots, as described in detail in the 5.2.1 subsubsection.

5.4.11.3 Managing the started machines

After a machine has been started by the script the managing work starts. The script has to make the boot process reliable and also make sure to check if the machine was started in a normal way and if the network is up.

This is done by pinging the host. When the host is up and running and the network was started, with a valid and known IP address, it is reachable by ping requests. The time the virtual machine needs until it is up and running and ping replies reach the daemon may vary. This is due to the fact that sometimes the hard disk has to be checked on consistency due to the copying of the images. This goes pretty fast due to the small size of an image and on good hardware, so it never takes very long until a machine replies, assuming no other problems were encountered. To be sure that we don't use an image with an wrong size, which indicates that there was a problem while copying, we compare the byte size of the clean image and the copied one. Therefore the byte size of the clean image has to be written in the configuration file. Of course, a checksum like an md5 or sha1 hash would be more reliable, but calculating a hash on a big file like our image just takes way too long, so we decided that the byte size fits our needs.

5.4.11.4 Importing the malware on the virtual machines

The malware that will be executed into the virtual machines has to be imported into them first. As described in 5.2.1 the virtual machine are running a SSH Server. This SSH server is used to secure copy the malware into the virtual machine.
This caused some problems, as the IP address of the host is changing all the time: different IP address but same fingerprint, so the host SSH client was complaining that it is not a valid host. This can be avoided by telling to the host not to check if the host is known. This is done in */etc/ssh/ssh_config*
To avoid to have to type in the password each time, which is not possible if you want to automatically import malware, we use certificates, which can be given as parameter:

scp -i cert.rsa malware Administrator@10.111.111.42:

This command is used to copy the malware file named *malware* to the machine with the IP *10.111.111.42* using the *cert.rsa* certificate. The file is copied as user *Administrator*, the *:* indicates that we want the file to be copied to the home directory of the user.

5.4.11.5 Executing the malware in the virtual machines

After the import of the malware into the virtual environment, it has to be executed. Again this is done through the SSH server and with a certificate.
First of all, the malware needs execute rights. This is easily accomplished by changing the mod of the file to give execute rights to the owner. Then the malware is executed, and the SSH connection to the virtual host is disconnected.

5.4.11.6 Logging the networking information

After having copied the malware to the virtual machine we start the logging process. We start it already before we execute the malware in order to be certain that we get all

traffic generated by the malware. To log we use the tool tcpdump on the host that takes as a parameter the tap device used by the virtual machine. Logging on the host instead of somewhere in the virtual network has a lot of advantages. So if a machine crashes we still have the logs and we mustn't fear that we won't have any space left on the virtual machine. Furthermore we don't need to care about how to get the capture to the host system before we destroy the used image.
Tcpdump is started with the following parameters:

sudo tcpdump -i tapX -w folder/malwareName.pcap -s0 -n filter

where *-i tapX* indicates the tap device we need to use. *-w folder/malwareName.pcap* tells the program where to store the capture with the malware name and the ending .pcap The *-s0* flag tells the tool to always log the whole packet. Would this option be omitted only 68 bytes of each packet would be logged. We want to know everything about a malware, so we log everything the malware is transmitting. The *-n filter* indicates the filter we want to use, this filter is defined in the configuration file and has the default value *src not 10.111.111.1 and dst not 10.111.111.1*. This tells tcpdump that it should not log everything that is going from the source IP address 10.111.111.1 to the virtual machine and also not everything thats going from the virtual maachien to the IP address 10.111.111.1 We need this to get rid of the SSH traffic we generate when connecting to the virtual machine through SSH to execute the malware. So everything that is logged now is generated from the malware, with one exception: our observation have shown that when the machines are started, they try to connect to *time.windows.com*, this certainly to synchronize the time through ntp (network time protocol). These requests can also be ignored as they are always the same.
Every tcpdump is started in it's own GNU screen window so you can follow the course of the captions. You see what virtual machine's traffic is still capturing, after the capture the window is closed. More on Gnu Screen in the 5.4.4 section.

5.4.11.7 Making sure that the given number of virtual machines is always running in parallel

Making sure that there are always the given number of virtual machines running is done through the database. This aspect is very important in order to not to waste time.
Each time a machine is booted, it creates an entry in the database, with the status "BOOT". Once the machine is reachable through ping requests, the status is updated to "UP". After the malware was copied on the machine and executed, the status is set to "RUN". If anything goes wrong during the booting or if there was a problem during the networking set up, for example a busy tap device that couldn't be freed, then the machine is stopped and the database field "status" updated to "KILL". If everything went smooth, the machine is stopped 10 minutes (or the timeout given in the configuration file) after the execution of the malware. As we only want to analyse immediately acting

malware, this is enough time to boot the machine entirely and execute the malware as well as seeing its consequences.

5.4.11.8 Managing the database

Assuming that the database is independent, not linked exclusively to one process, we have to take extremely care of how we write to that database. This caused us a lot of problems because threats started to write to the databases at the same moment, and then the database was locked. We implemented a solution with system V semaphores but this was rather complicated, and against the idea of the daemon, to keep it simple to avoid crashes of the daemon itself. So we threw this approach away and rethought the whole database connection mechanism.
Using mysql instead of sqlite solved also a lot of problems, and now the framework is using the database without any concurrent access problems.

5.4.11.9 Make the daemon process run forever

As the main goal of the daemon is to run in a reliable way as long as it is wished, the whole process has to take care that there is always a trace on how many machines are running and how many are free and can be used for other malware.
Again, this is mainly done with the database. The daemon queries the database to find out how many machines are free. This is easy to find out, simply check the last 'n' entries (where n is the number of virtual machines taken from the configuration file) and see if there are entries where the status is set to "KILL". Every machine having the status "KILL" or "STOP" has stopped to run, so if there are not more machines running than wished (in the configuration file) we can start new machines until we get that wished number of machines running in parallel.

5.4.11.10 The interacting of the whole framework

As the framework is rather complicated we'll try to explain it in the way it is run. So first the root user needs to execute a bash script. This is the only part that has to be done as root. We have to be root or else we wouldn't be allowed to do all the operations needed. The bash script has been described in detail in section5.4.7

What the configuration script does exactly is described in the 5.4.5 section. The database created by the *createDatabase* script, is described in the 5.2.3 section.
After this point, we don't need root privileges anymore, the rest can be done as a normal user with some sudo rights. The modification needed in the sudoers are explained in section 5.2.2
Of course, the bash file takes care of the actual situation. For example if the database already exists, it will not drop the existing values, as they might be important. Another example is the configuration file building script: if there is an existing configuration file,

it asks you if you want to keep this file or overwrite it with new values.
So now we'll show how a malware is inserted and run in the framework.

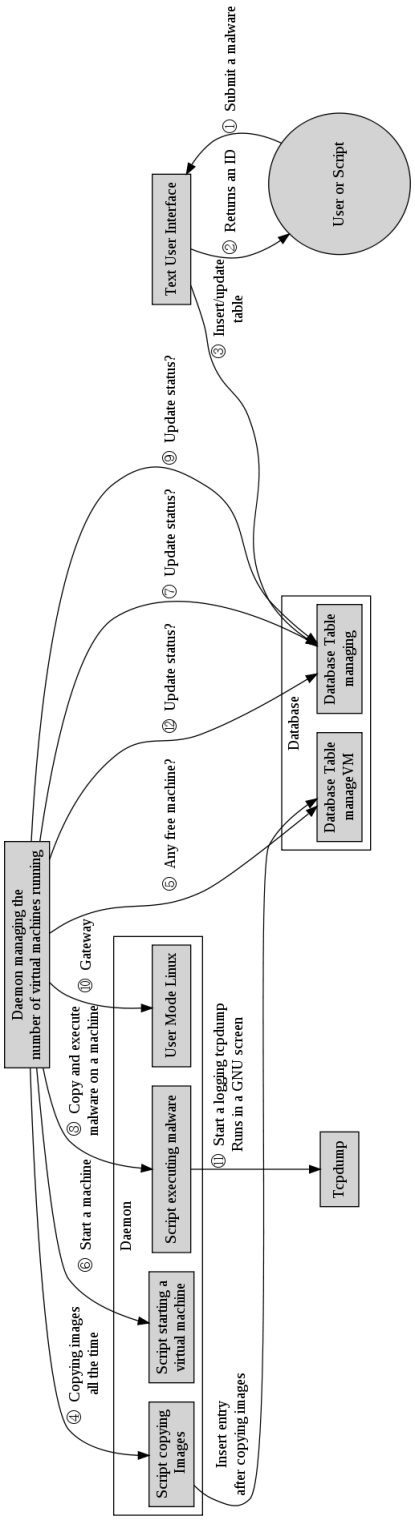


Figure: 5.4.10.10.1 The behaviour of the framework

On this graph you can clearly see that the database is the main point of this work. The two main parts of the application, the daemon and the text user interface, both, connect to the database to either write down new jobs or update the status of current ones.
The status hold by the database are the real important information, that has to be right all the time.
First, a user or a script uses the text user interface to commit a malware-job. This job is then inserted into the database and an unique id is returned. As explained in detail in the 5.4.10 section.
Then, the daemon starts different processes, all started in an GNU screen session. Each of these processes has his own job. One script is used to build image copies from the clean image, this is done all the time in the background. Another script is starting virtual machines every time one is needed. The SSH script is used to copy and execute malware on the virtual machines. It waits all the time and queries the database to see if an entry has an status set to "UP", if so, malware is copied on that machine and executed. Lastly, a last GNU screen window is used to start the User Mode Linux in it. So you can attach this window at any moment and see the status of the booting or the requests the fake DNS Server gets.
The SSH script, also starts a tcpdump program in a GNU screen window each time the execution of the program reaches a certain point. In order not to be flooded by the window in the GNU screen session, we have to close the window after using it. How we do this is explained in detail in the 5.4.4 section.
The processes in the daemon make heavy usage of the database, to know what they have to process next. Each process is updating the database after having processed its task.

6 The Analysis

6.1 Used Dataset

Before we start the analyse of the malware we have to know our enemy. Therefore we scanned the malware we used with the GPL licenced open source anti-virus toolkit for UNIX called Clam AntiVirus[1].
As we used rather old malware for our tests we had a good recognition rate. However some malware wasn't recognized at all, this malware seems to have a low spread rate, as all the other of that time period having been recognized.
Here is the table summarizing the detection of the malware.

[1]Official Website of the project: http://www.clamav.net/

Malware	Detection by ClamAV
0005e7b17e99a65518f270304b96b9c5.exe	Trojan.Eggdrop-79
003c057bc5e071e4c01003f0a4dbe1d8.exe	Trojan.Poebot-14
004b1c98c7a533e08550a6583285168d.exe	Trojan.Mybot-7794
0060f37823ad3d96bf2a9711a139615a.exe	Trojan.Mybot-7292
00b920257fd4b09759f122a050af733b.exe	Trojan.SdBot-4867
00c208a54242f24f503a24db0a30e707.exe	-
00ca5281eab44feec8a4ddeb79543a81.exe	Trojan.Dropper-901
00d858f0b4ce5cba9d94fc63c3850c2e.exe	Trojan.Poebot-108
00dccfeeb58e5f4d536d498317ad0cc2.exe	Worm.Sasser.H
00e47d25d56b516258a15facd6da23d0.exe	-
0101db68c80aff68cf2ce92569feb097.exe	Trojan.SdBot-4284
0123d39925b012a38b581492f9b35a11.exe	Trojan.SdBot-4155
01471043c1945ba838ff7898f4a72713.exe	-
016c1c4915b5658e1c4120775f51707f.exe	Trojan.SdBot-4169
018362267460f8a0d3ec05847d6cc738.exe	Trojan.SdBot-5137
020425481c927c537aab6090413bcea9.exe	Trojan.Delf-601
0410741e2e5bf23fdad90372a3b8d50b.exe	Trojan.SdBot-1210
04af7239845601e9d785a7824b6ca34e.exe	-
059b73e45d131569f008312f4eb2568e.exe	-
09adee9dd91cb978df6d7ba4c1bf9e76.exe	Trojan.Aimbot-25
09be1135f31788376e575777f2dc77a0.exe	Trojan.SdBot-4179
10be774286859a0a42f3d7d49e8c45cb.exe	-
09f6da7d496ee8351e4ad26ff015851b.exe	-
33c6589b5101f6918ce31b8639985bb3.exe	Trojan.SdBot-6778
340f782a9b5c782e662e57153f1d7919.exe	Trojan.Packed-142
3435f105762fe4cccc4050c31babebc7.exe	Trojan.MyBot-8956
3437ab44892cb4ded1384d309ae7f52f.exe	Trojan.Eggdrop-15
34465dc2b7efa34abcaf5bcdfabd6130.exe	Trojan.Ircbot-305
46c3701d771c978a282b17100dc0cba0.exe	Trojan.SdBot-1208
564be3ff18279a1f43fbf729c2c24d34.exe	Trojan.Mybot-6502
575063c315e58cde4812a289da68a3ac.exe	Exploit.DCOM.Gen
5784cd80a681996e83fbf481578102c8.exe	Trojan.Lineage-80
9c38226a4e42bdcfb57e7167493849f7.exe	Trojan.SdBot-4179
ae0b5e6c6ac05c6aae19b2317806e8bb.exe	Trojan.Mybot-6527
ae24016e72b127602e8bfde2b08ea69b.exe	Trojan.Mybot-5937
fffdbe89431a3dde4a82edfbe6b71d76.exe	Trojan.IRCBot-1067

Table 6.1.1: Malware scanned with ClamAV.

As you can see, a lot of bots are contained in the data set. This is exactly what we need, as bots usually connect to a network of a certain type, for example an IRC server that is used as a command center. What connections the malware tried to setup will be analysed in the three following sections.

6.2 Top10 list of DNS requests

During our testing phase, we logged several DNS request on our fake DNS server running on the User Mode Linux. Here is a list of the top 10 DNS requests performed by the malware:

Number of requests	Domain
14121	img.brainkill.net
281	home.paltalkdc.com
222	ircd.darkroot.at
140	home.paltalkdc.com
89	alabama3.isthebe.st
36	h4ck.bleah.info
24	kam.alf4-radmin.com
23	asn.ma.cx
8	power.prout.be
3	xt.ircstyle.net

Table 6.2.1: The 10 most used domain names.

This list has been generated from the logging done in the User Mode Linux. We didn't consider the requests for *time.windows.com* as these requests are done each time when a Windows virtual machine is booted. This request should synchronize the time through the network time protocol.
Furthermore, these DNS requests already show us that the different malware act in different manners. So, for example, at the first place in the table is a malware performing a huge amount of DNS requests to a specific domain: *img.brainkill.net*. This malware doesn't seem to care about the fact, that there is no Internet connection available and that the malware can't reach that server; it simply continues performing requests.
Doing assumptions now, on how the malware is programmed might be a little bit early, but everything seems to indicate that the malware is performing an endless loop, querying the DNS server for a specific domain, and if that domain is not reachable it continues, certainly until it gets a positive answer from the server.
This behaviour of a malware is not very clever, as the malware generates a lot of traffic and might be detected easily due to that. A malware doing less requests, like for instance the last entry of the table with only 3 request in 10 minutes, might be less suspicious, as the malware is less active and shows up less often in the firewall logs or similar detection

mechanisms.
Performing DNS requests at all might not be a very good idea, as this can be logged in a DNS server run by a company. It would be less obvious to detect a malware using an IP address instead of first resolving a domain. At least, then the address, the malware wants to connect to, would not show up in the logs. Of course, this only works if the malware doesn't try to connect to a dynamic IP address.

6.3 Average packets send by the malware

Another analyse we did on the malware was to count the packets the malware sent. We will present this results in form of a table, showing the number of packets send by each malware and at the end we will compute and present an average of packets sent. This result was computed with the results we had from our captures: each capture lasted 10 minutes and was performed under the same conditions. So we think that the average presented here is representative, even if the data set we used was only a small one (37 captures). If you use a bigger data set, of course the average will change, but we belief it will stay close to these results.

Malware	**Packets sent and logged**
0005e7b17e99a65518f270304b96b9c5.exe	56
003c057bc5e071e4c01003f0a4dbe1d8.exe	24
004b1c98c7a533e08550a6583285168d.exe	506
0060f37823ad3d96bf2a9711a139615a.exe	0
00b920257fd4b09759f122a050af733b.exe	1728
00c208a54242f24f503a24db0a30e707.exe	15
00ca5281eab44feec8a4ddeb79543a81.exe	212
00d858f0b4ce5cba9d94fc63c3850c2e.exe	35
00dccfeeb58e5f4d536d498317ad0cc2.exe	4370
00e47d25d56b516258a15facd6da23d0.exe	2
0101db68c80aff68cf2ce92569feb097.exe	61
0123d39925b012a38b581492f9b35a11.exe	2
01471043c1945ba838ff7898f4a72713.exe	2
016c1c4915b5658e1c4120775f51707f.exe	0
018362267460f8a0d3ec05847d6cc738.exe	1712
020425481c927c537aab6090413bcea9.exe	675
0410741e2e5bf23fdad90372a3b8d50b.exe	2
04af7239845601e9d785a7824b6ca34e.exe	44
059b73e45d131569f008312f4eb2568e.exe	17
09adee9dd91cb978df6d7ba4c1bf9e76.exe	4
09be1135f31788376e575777f2dc77a0.exe	302
09be25601f6d0a80b692a23e5cada5f0.exe	15
10be774286859a0a42f3d7d49e8c45cb.exe	1706
09f6da7d496ee8351e4ad26ff015851b.exe	2
33c6589b5101f6918ce31b8639985bb3.exe	930
340f782a9b5c782e662e57153f1d7919.exe	176
3435f105762fe4cccc4050c31babebc7.exe	23
3437ab44892cb4ded1384d309ae7f52f.exe	1019
34465dc2b7efa34abcaf5bcdfabd6130.exe	341
46c3701d771c978a282b17100dc0cba0.exe	56
564be3ff18279a1f43fbf729c2c24d34.exe	6
575063c315e58cde4812a289da68a3ac.exe	2289
5784cd80a681996e83fbf481578102c8.exe	20
9c38226a4e42bdcfb57e7167493849f7.exe	4
ae0b5e6c6ac05c6aae19b2317806e8bb.exe	514
ae24016e72b127602e8bfde2b08ea69b.exe	115946
fffdbe89431a3dde4a82edfbe6b71d76.exe	40
Average	*3590*

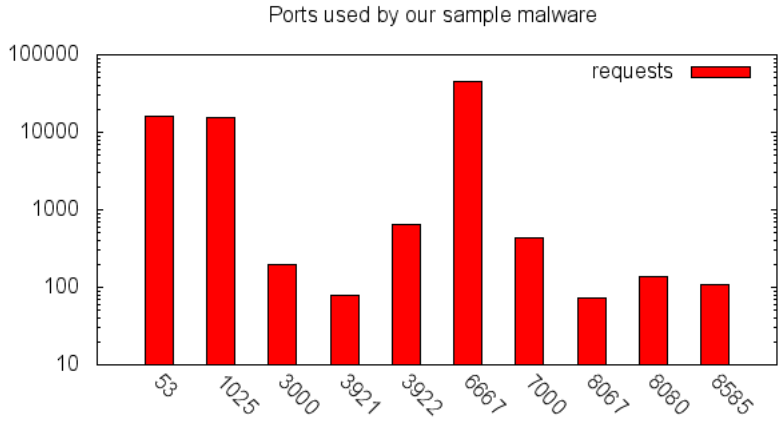
Table 6.3.1: Packets sent by malware

This indication is interesting, as the amount of packets sent by a malware shows us how much traffic it generates and so how much noise it makes in the network. Being a successful malware is the same principle as being a successful burglar. The less noise you make the more chances you have to be successful and stay undiscovered.
In our analyse the average packets transmitted by a malware during a capture of 10 minutes is 3590 packets.

6.4 Most used ports

Analysing the network activity of the malware we executed, we found out that there was, indeed, a lot of activity going on.We filtered out the ports numbers of the different tcpdump-logs and analysed them on occurrence. As suspected, the most activity was on port 6667, which is the port most irc servers are running on. As we used old malware for our tests, this confirms what is known about old malware. Most of them are controlled by irc command and control centers. This clearly shows that our framework is working like it should, and gives a certain certitude on the logging performed. On the following histogram we present the results of the logs. This graphic represents the occurrence if each port.

The x-axis represents the ports and the y-axis how often a connection was established to that port. The histogram uses a logarithmic scale representation.



Histogram 6.4.1: Port usage by the malware.

As you can see, port 53 also got a lot of requests. This port is the port used by the DNS server. So malware was trying to translate domain names to IP addresses. These requests have been logged by the fake DNS server installed in the User Mode Linux, as described in section 5.4.8 and the list of the 10 most asked domains is presented in section 6.2

6.5 Advantages of our solution in comparison to other approaches

The following section will describe what the advantages of our solution are.
There are a lot of malware testing frameworks available. What are the advantages of our framework? When you are fighting malware activities and want to protect your network or your systems against new, spreading malware, you have to be able to get a fast analyse of the malware, in order to know what it tries to do, and where it is sending stolen information.
As nowadays most malware use encryption and/or compression to protect its source code [3] from being reverse-engineered, at least in an automated way, you have to figure out other possibilities to find out what the malware is up to. The manual reverse engineering of a malware costs too much time, and due to packers it is hard to automate this process. By logging the network traffic generated by the malware, we have a way to find out what the malware wants to communicate and to whom. This lets protect us against the given malware. For example, if the malware tries to connect to port 2342 we could simply modify our firewall policy in a way that it discards everything going to port 2342.
All in all, we could say that a static analyse of the malware is not a fast solution, as it involves a lot of manual work. Therefore our framework is much faster. On the other side, executing a malware is not a perfect solution either, as by this way you can only observe one part of the malware, the one that generates traffic. You can not figure out what the logic of the malicious software is, as you only see one way of execution. If the port 2342 is open, our malware might use this port to communicate with the attacker. If this is the case you will never find out what the malware would have done if the port was closed. If you want a complete analyse of the malware a reverse engineering is certainly a better approach than our framework.

7 Problems and Choices

Due to the complexity of the task, it was sure that we would encountered different problems. As expected, we encounter some problems, which were more or less time consuming. We'll present some of the major problems in this section.

7.1 The snapshot problem

7.1.1 Description of the encountered problem

The virtualization software we use, Qemu, is a very advanced tool. It has a monitor, which allows you to perform different tasks like, for example, dump the memory, or turn the machine off. We use this monitor, which is accessible over telnet to turn off the machines, as described in the previous sections.
We also wanted to use this monitor to perform a snapshot of the virtual machine, once it was booted. A snapshot stores the state of the image at a given moment. In theory this snapshot can be used to restart the machine from this stored point.
In our continuous wish to speed up the framework, we considered and tried out this technique. The idea was to use 3 snapshots of virtual machines having a different IP address. We would then always use this 3 IP addresses in order to get rid of the problem described in section 7.2.
We found out that performing a snapshot of an virtual machine image at a given time was no problem, but unfortunately restoring the status of the image and continue the usage of the machine always failed. This seems to be a bug in the software. Or at least we didn't manage to make it work. Even not with in the developer version, which we compiled and tested.
So we had to give up this idea.

7.1.2 The solution

This problem was solved by a work around. As we could not find the bug in the source code of the QEMU developer version, we implemented an other way to spare time. This method is discussed in detail in section 5.4.9

7.2 The IP address problem

7.2.1 Description of the encountered problem

One problem we encountered very early in the development phase was a problem related to the IP addresses that have to be unique. In deed, if you use the same image file and start 3 machines simultaneously, you will get IP address conflicts, as each machine will have the same IP address, assuming you don't run an dhcp server and use fix IP addresses.
A lot of solutions have been tried and implemented to solve this issue in the best way. The first idea was to simply start a dhcp server, that distributes IP addresses when a dhcp-lease request gets in. That idea seemed very good at the beginning, and worked fine, but the dark side of that idea was that it started to be impossible to trace the machines, as you could not really know for sure what machine got what IP address, and how should the further processing with the copying of the malware then go on? So, we tried to give IP addresses related to the mac addresses of the machines, which seemed to be a good idea, as the mac address of the network devices can be passed as an argument when calling qemu. In order to make this work we had to write an entry for every IP address that could be given by the dhcp-server in the configuration file. This idea seemed okay, but not ideal, as a mechanism had to be implemented to know when the dhcp server gives an IP address to a client, some sort of trigger.
Another approach was the idea to use only a given number of machines, lets say 3, and use them with fixed IP addresses in order to be sure that we know the IP address of each machine. The drawback with this idea was that you loose a lot of time, due to the fact that the images have always to be overwritten after usage, which of course could have been sped up by the script like the one described in section 5.4.9. Furthermore, the point that made us give up this idea, was that we had to prepare each image manually. This means that you have to make a copy of the image, boot it, change the IP address, shutdown the machine and go on. Imagine you have 10 machines you wish to use, this would lead to a lot of work, done manually and would cost a huge amount of time. The solution we found than does all this in an automated way, and lets the user invest his time in the evaluation of the log files and not in the setting up of machines.

7.2.2 The solution

The task of setting up the machines, so that they would be ready to use without more preparation than described in section 5.2.1, has been implemented the in following way: First, before anything else, the image of the Microsoft Windows System gets mounted as a loopback device by our script.
For this, the image has to be in a raw format, and not in some compressed format like qcow2, which we used at the very beginning of the project because it made copying much faster. The fact that there is no compression anymore is the only negative point of this

technology, but this issue has been fixed with the script described in 5.4.9.
After the image was mounted as loop device, it can be used as a normal part of the file system. In this way you can access everything that is located on the *C:* drive without any problem. We use this possibility to copy a file on the drive, containing a short batch script that setup the IP address as well as the netmask we want to use for this specific image. The content of this batch file has been described in section 5.2.1. So we only need to make sure that this image contains an IP address that no other machine on the network is using at that moment. This solves the problem in an elegant and simple way. No interaction with the user is needed, it is fast, much faster as if the user had to do it manually. In fact, in practice, the mounting of the image, copying the generated batch file onto the file system and after unmounting the file system again takes less than 10 seconds. Knowing the startup time of Windows this is at least 10 minutes faster.
As described in 5.2.1 the batch file is executed at boot time by a scheduled job. After this the image has the correct IP address and the right netmask to be reachable on the virtual network.

7.3 The networking problem

7.3.1 Description of the encountered problem

After having solved the IP address problem, the next network-related problem made it's apparition. We wanted to have a simple network, easy to set up and easy to maintain with low fail potential. This network should make every machine accessible through all other. This includes the User Mode Linux gateway as well as the other virtual machines, running in our case, a Microsoft Windows operation system.
Building a network with virtual machines, is a wish that many User Mode Linux seem to have, therefore there are solutions for this. The main solution is using a software simulating a switch. This software is called vde_switch. We tested this software and found out that is doesn't fit our needs. This software has to be run as root user, which is not good for a framework that is build to execute malware, because if a malware is able to execute code on that vde_switch, then this malware would be root on our machine and able to compromise not only the whole machine, but also the whole private network, as well as we would allow, if something like this happens, the software to connect to the Internet and continue it's spreading.
Furthermore, this approach seemed easy to setup, but while testing it we had plenty of problems due to unclean setup of the switch. The machines were not reachable. We didn't manage to make it work in a reliable way, so we searched for another solution.

7.3.2 The solution

Another approach that had to work according to our researches was using tap devices. A tap device is a virtual network kernel driver. They simulate network devices in such a way that usual network applications can use them. A tap device doesn't contain a physical network device. Each virtual machine, no mater if it is a qemu machine or a User Mode Linux machine gets his own tap device. Using a tap device in an automated way is easy to setup an easy to manage. Failures only occur if an tap device is already in use so we only have to take care that a tap device is free before we use it. This is done by carefully freeing the devices after the usage. Building a bridge with all the tap devices used made a small virtual network, exactly what we needed. Pinging machines through the network is no problem anymore, as well as pinging the virtual machines through the host. To be able to communicate from the host system to the virtual network we need to assign an IP address belonging to the network to one of the tap interfaces. Once this is done everything works like a charm.

7.4 The forks problem

7.4.1 Description of the encountered problem

The framework makes very large usage of parallel execution of code. To implement this in a decent way wasn't that easy. In fact, we tried to solve the problem using forks. Every new machine was started in a new fork, all the actions performed by the different forks were related to one machine. So the iterative processing through the code was chosen in a first implementation. Using forks forced us to make an synchron communication through the code. This idea lead to different problems. The debugging of the application started to get really hard as soon as the forks were implemented. This made the whole developing slow.
The replications of an error in the code were hard to catch in a decent way. So if an error occurred, most of the time, this resulted in an fork crashing. Somehow that was not important as the framework was built in such a way that a fork crashing could not made the daemon crash. The problem was, again, the time cost. As said, catching an error in a decent way was not always an easy job, and an error could not always be detected as soon as possible. So it costs time until a machine is free again, this time could be spared. So we rethought this part and built another solution fixing this problems. Despite the better solution, some malware had already been tested using the forked system. This means that we have a working solution implemented with forks.

7.4.2 The solution

The solution to this problem has been implemented as follows. Instead of forks we now use GNU screen, longly presented in section 5.4.4. This approach is significantly different

from the approach with the forks. Indeed, we use this GNU screen in an asynchronous way, which allows us to get rid of all the problems mentioned in the section above. First of all we start these GNU screens once, at least for the copying image part described in section 5.4.9 and for the script performing all the SSH related actions. The script starting a machine is still only called when needed the same is true for the tcpdump. So if the virtual machine crashes before the tcpdump could start, we don't even start it. This makes us win a big amount of time, and the script are still able to run in parallel without any problem. A big advantage of using GNU screen is that the debugging is also facilitated extremely. It is easy to find out where the problem occurred, if some occurred, as you can print immediately to the window. As we split the complex problem into small parts, we know exactly what part is doing what and usually every part gets it's own, named after the script, GNU screen window.
Choosing an asynchronous implementation with GNU Screen instead of an synchronous implementation using forks has another big advantage. Indeed, it simplifies the data processing a lot. Many huge projects are using this technology for their data processing, for example the MapReduce [5] programs, that are running on Google's cluster each day.

7.5 The database problem

7.5.1 Description of the encountered problem

As the database is used with very high density in our project, we had to be sure that everything related to the inserting, updating and reading of the database was done correctly. A failure in updating the status of a virtual machine would make fail this machine to be used in a proper way. This would cost a lot of time.
The first implementation used an sqlite3 database. Sqlite is an embedded relational database, it is very small, and uses a simple file to store the database. We found this a good idea, as in this way we would be able to backup the database by only moving one file, or drop the whole database with only one delete command. There would be no need to connect to a database system first. Unfortunately as sqlite locks the whole database for every transaction, we encountered a lot of problems due to a locked database, when we wanted to update or insert something into it. As our framework always runs more instances of windows machines in parallel, we had to figure out how we could avoid deadlocks. A lot of engineering and prototyping was done in this part of the work, but no suitable solution was found.

7.5.2 The solution

As definitely no solution fitted our needs best, we decided to switch the database system, and use mysql. The fact that doing backups is here a little bit more work is not important, as we don't need backups of the database. In fact, the database is only for

managing the current session, afterwards it gets nearly worthless, except perhaps for debugging purposes.
Once we reimplemented our sql class in a way it should use a mysql database instead of an sqlite database [7] our troubles went away. This version of the framework is using a mysql database with 2 tables as described in 5.2.3

8 Future work

The Malware Analyse Framework is now a running project. We think the requests of the industrial partner have been fulfilled, and that our framework is easy to use and customizable with ease.
The fact, that this work was related to the research work of a PhD student doing his researches in the same company, made that the framework had to be highly configurable, as it is not sure at the moment in what case of application the framework will be used. In order to give the company the most flexibility, two versions of the code are ready to use, and have been tested and both are totally operational. One is using forks and the other is using GNU Screen. Even if both are working, we strongly recommend the usage of the version using GNU Screen as the stability and reliability as well as the debugging are better.
A possible scenario could be the further development of the framework in order to perform some fuzzing on the malware. So you could imagine that if the malware wants to access an IRC server, our framework could start an IRC server and make the malware believe that the IRC server in our virtual network is the one it wants to connect to.
In order to speed up the whole process it would be good to have a framework that uses snapshots. So, one future work would be to investigate this aspect and to, either fix the software if it is a bug, or figure out why it is not working at the moment with our set-up. After that, the framework would really be ready for deployment in a productive environment.

9 Bibliography

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[2] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

[3] Éric Filiol. *Les virus informatiques : théorie, pratique et applications.* Springer, 2004.

[4] Alexandre Dulaunoy Gérard Wagener and Radu State. Automated malware analysis. *hack.lu*, 2007.

[5] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters. *MapReduce J.*, 2004.

[6] Mark Mitchell and Alex Samuel. *Advanced Linux Programming.* New Riders Publishing, Thousand Oaks, CA, USA, 2001.

[7] Michael Owens. Embedding an SQL database with sqlite. *Linux J.*, 2003(110):2, 2003.