Behavioural Analysis of Malware Using Custom Sandbox Environments

Master Thesis
January, 2020

Tarik Muhovic

AALBORG UNIVERSITY

STUDENT REPORT

Aalborg University
Networks and Distributed Systems

Networks and Distributed Systems
Aalborg University
https://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Behavioural Analysis of Malware Using Custom Sandbox Environments

Study:
Networks and Distributed Systems

Project Period:
10^{th} semester, Fall 2019

Project Group:
NDS10

Participant(s):
Tarik Muhovic

Supervisor(s):
Jens Myrup Pedersen
Egon Kidmose

Copies: 1

Page Numbers: 100

Date of Completion:
January 10, 2020

Abstract:

The project and its contents were made for the 10^{th} semester at Aalborg University. The task for the project was to explore the idea of understanding if changing different parameters called artefacts within an established virtual environment would lead to a behaviour change for malware samples by creating and using a custom sandbox environment. Using software such as the malware analysis tool Cuckoo Sandbox and the Virtual Machine (VM) manager called VirtualBox, a systematic way of testing malware samples in different environments for behaviour change, was made.
As such, a system was made consisting of Cuckoo and VirtualBox where two custom VM images were created with one resembling a normal virtual environment and one where all references to the virtual environment were taken out by manipulating different artefacts in the system.
During the project, a test setup was created which showed that malware changed behaviour between different tests made when implementing different artefacts. Some malware did however remain dormant because of lack of additional artefacts in the test environment. Overall, out of 21 randomly selected malware samples, 9 were observed to have a change in their behaviour. Because of the small sample size used for testing, concluding that the different artefacts had a definite impact on the behaviour of malware would require more malware and repeated tests to be performed. The project was therefore a success where a system was created that showed behavioural change in different types of malware, but further work would be required in terms of system scaling and repeated tests in order determine what caused each specific change in behaviour.

Preface

About

The report is written and completed by Tarik Muhovic as a master thesis at Aalborg University under the study Networks and Distributed Systems during the 10^{th} semester, fall 2019.
The project and the related problem were made based on the occurring inaccuracy of malware testing in closed of environments such as virtual ones and how malware behaviour would change upon the detection of said closed environments in order to avoid being systematically tested.
The focus of the report was also twofold, in that it was made with being worked on in the future in mind. Therefore the report is both focusing on the problem described as well as being informative for people developing on the system at a later point in the future.

Acknowledgements

We would also like to thank VirusTotal for providing malware samples for the project in order to optimally do malware analysis [51].

Report reading guide

Sources will be referenced throughout the report using the IEEE reference style. All the referenced sources can be seen in the bibliography and are listed in alphabetical order.
A source will be referenced with the following method and layout, with numbers and square brackets:

> "Example text [1]."

Where the source referenced in the text is number "1" in the bibliography.

Sections and chapters are referenced with chapter and section numbers followed by one another. Example:

> "Example text (see section 2.3.1)."

Where the section referenced in the text is chapter 2 section 3 and subsection 1.

Acronyms

AAU Aalborg University. 24, 42–44, 48

ACPI Advanced Configuration and Power Interface. 51

API Application Programming Interface. 44, 46

DL Deep Learning. 2, 3, 79

DSDT Differentiated System Description Table. 51

FADT Fixed ACPI Description Table. 51

GPU Graphics Processing Unit. 42

JSON JavaScript Object Notation. 26

MAC Medium Access Control. 51

OS Operating System. 41, 44, 47, 61, 62

REST Representational State Transfer. 44

RSDT Root System Description Table. 51

SSDT System Service Descriptor Table. 51

UI User Interface. 50, 70

VM Virtual Machine. iii, 16, 30–33, 37, 41, 42, 44, 45, 48–51, 53, 57, 60, 61, 77

WPAD Web Proxy Auto-Discovery. 72

Table of contents

1 | Introduction

Over the last years, the increase in use of personal computers and phones has led to a major increase in cyber-attacks where some companies such as AV-Test record over 350.000 new malicious programs each day [6] (see Figure 1.1).

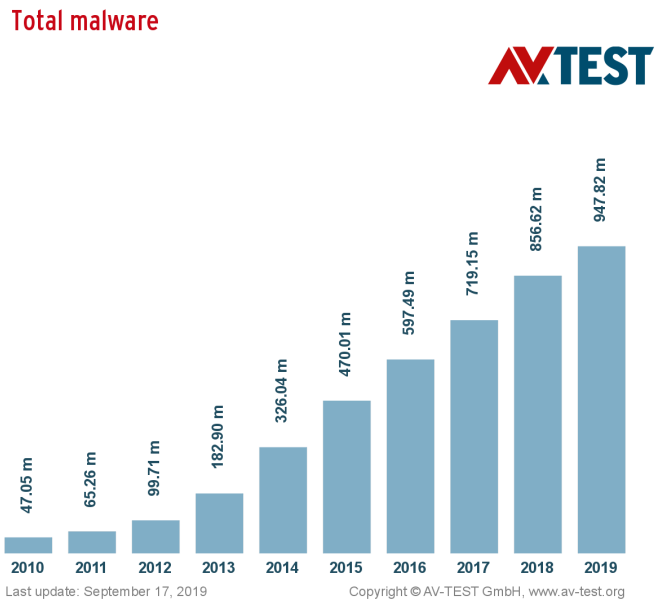


Figure 1.1: How found malware has increased over time [28]

In order to beat the criminals and help protect the users, advanced methods are being developed to analyse these large amounts of malware such as the use of machine learning where learning about how malware behaves is done by observing different types of malware [27]. In the context of machine learning for anti-malware purposes, researchers use a large amount of data in the form of malware samples that are already classified as some type of malware. The samples then have their behaviour or other features tested against software that is known to safe and not malicious in order to train a machine learning algorithm and model to know the difference between malicious and non-malicious software. A high accuracy is required since most anti-malware is deployed on personal computers where the anti-malware runs. If the machine learning algorithm and model aren't good enough, false positives can occur which has been observed to have happened in the past. Here companies such as Malwarebytes encourage people to label samples themselves which Malwarebytes can use to improve their machine learning modules in their software [36].
The types of machine learning come in two different forms, unsupervised and supervised learning. Both methods improve the rate and accuracy of how malware is detected and if a piece of software is actually malicious or not. In other words, how fast and how few false positives are detected are

what is improved upon. Unsupervised learning includes having a large data set without knowing the right answers for the task. This method requires a lot of manual labour to label different data but is valuable for threat detection since it requires less resources to execute and can be done with automation. Unsupervised learning helps threat detection by improving the rate at which classifiers are applied to malware by learning how the malware is constructed in terms of how it interacts with a system such as system calls and file searches. Supervised learning on the other hand are where both the data and the right answer are available. Here a model is created to fit the right answer where in the case of malware, the right answer is doing something malicious on a system that is predefined. This can usually be done by knowing how previous malware behaves and apply those findings to new malware and observing if the same outcome can be reached (see Figure 1.2).

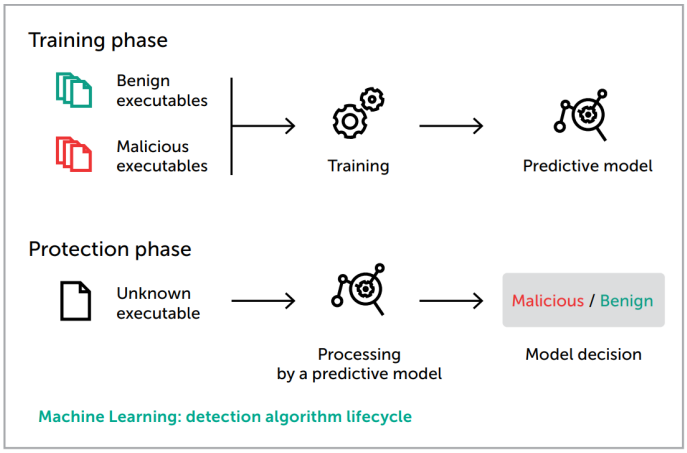


Figure 1.2: How supervised machine learning is done [27]

There are however rare cases where samples used for training a model are insignificant. These samples typically are very rare but also cause a lot of damage such as samples that have zero day exploits and come in the form of a combination of a computer worm and ransomware which has the ability to infect and spread quickly over a network. These samples usually have only been seen once and are therefore hard to use for training basic machine learning since there isn't enough data to train with. In this case special kinds of Deep Learning (DL) are applied where representations of input features are made and tested with [32]. DL has in the past been observed to be successful in areas such as computer vision, language processing and other tasks where you want to have high level meaning from low level data. In some cases, DL can be fine-tuned enough to surpass human ability for a specific task [45]. The difference between DL and normal machine learning is that DL uses feature learning through patterns instead of task specific learning that humans tell the model to look for. Using DL in an area such as malware, the model can learn to extract features and classify data into various classes. A way of applying malware to DL is to create image signatures of malicious and non-malicious software which the DL model can use to understand the difference between malicious and non-malicious. An example of a simple DL model applied to malware can be seen on Figure 1.3. Here, the different image signatures can be seen as well as the outcome determined by the DL model. The different signatures generated showed that certain types of software have common types of behaviour which could be showcased in the form of images. Using this technique, different malware types could be identified via their signature.

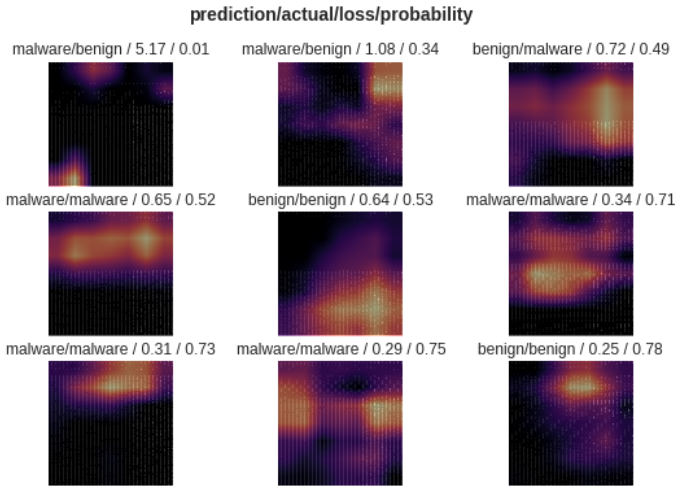


Figure 1.3: Image signatures generated by running a simple DL model to identify malware samples [32].

A big problem with using machine learning to find new malware is the degradation of the model that is used to find new malware. As malware gets detected, malware authors rewrite their code to avoid new detection methods and as such, the model that was used earlier become worse over time as new malware is made. This is however counteracted by allowing the models used to be updated over time as new malware is discovered. If the model used isn't updated, the degradation can look similar as seen on Figure 1.4.

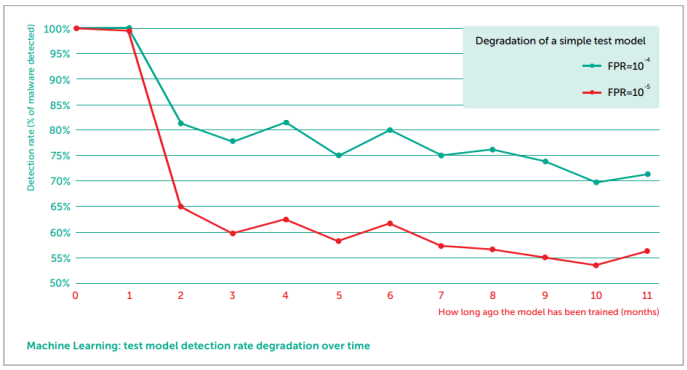


Figure 1.4: Degradation of machine learning models [27]

The degradation of a model also applies to the machine learning being used [42]. If the data samples used to train the machine learning algorithms isn't correctly labelled or corrupt, the model becomes inaccurate since the difference between malicious and safe software can be harder to distinguish. This results in more manual labour being required to fine tune the model and algorithm to prevent problems which could result in unreliable results.

1.1 Discovering Malware

Finding all of the malware mentioned earlier is however not a trivial task. Unless the malware is an already existing copy of a detected malware, special methods need to be used in order to figure out if a particular piece of software is malicious or not. For simple malware this task can be easy and doesn't require any special equipment such as machine learning to find. This type of malware can be found by using the process of reverse engineering of the software in order to see what it does [2]. Reverse Engineering of software can be done by using what is called a disassembler which shows what the executable is doing in assembly code. Tools such as "IDA Pro", "WinDbg" or "x64dbg" can be used for debugging executable running on Microsoft Windows machines. An example can be seen in the following code where a simple for loop is complied, then disassembled into assembly code.

```
1 int main() {
2   int count = 0;
3   for (int i = 0; i < 10; ++i) {
4     count++;
5   }
6 }
```

Figure 1.5: Source code

```
1          main:
2  400522  push   rbp
3  400523  mov    rbp,rsp
4  400526  mov    DWORD PTR [rbp-0x4],0x0
5  40052d  mov    DWORD PTR [rbp-0x8],0x0
6  400534  cmp    DWORD PTR [rbp-0x8],0x9
7  400538  jg     400544 <main+0x22>
8  40053a  add    DWORD PTR [rbp-0x4],0x1
9  40053e  add    DWORD PTR [rbp-0x8],0x1
10 400542  jmp    400534 <main+0x12>
11 400544  mov    eax,0x0
12 400549  pop    rbp
13 40054a  ret
14 40054b  nop    DWORD PTR [rax+rax*1+0x0]
```

Figure 1.6: Decompiled assembly code

A problem with using disassembler is that modern compilers are very good at optimisation and will omit and reorder a lot in order to make the program perform faster and be smaller in size. It is therefore typically not an option to disassemble a piece of software and expect the original source code. Using a disassembler where possible, if the software code reveals anything that can do harm such as encrypting files on the machine, it can be marked as malicious. However, newer types of malware put up several measures in order to stop unwanted people from viewing the code of the software. This could come in the form of packing the software or using polymorphic code which would protect the code from being read by reverse engineering which makes it harder to identify the malware since the code doesn't follow standard application binary conventions such as stripped symbols and obfuscation [22] [67]. Because of this other forms of malware testing are required where some methods include trying to reverse the packed malware all together [11] [63]. With the aforementioned machine learning

aspect of malware testing, it has also become popular to test malware in controlled environments where reverse engineering can't be achieved. These controlled environments are called sandboxes and have the purpose of simulating a real system where the malware has been executed on. Sandboxes are usually made by running a closed off virtual environment where the malware is executed and observed what happens for a period of time. If it is observed that the malware sample does something harmful in the virtual environment, such as removing or replacing critical system files, it can be classified as malicious.
This method of testing comes with both pros and cons. The pros of it is that testing can be automated which means that a lot of different potential malware samples can be tested over a longer period of time. The testing can also be combined with machine learning and visualisation where an algorithm can look for specific behaviour can classify malware all by itself without human interaction, if trained enough [48]. The cons of the testing are that not all malware behaves the same. This comes from the knowledge of basic software programming and how certain things are executed. For software to do something it needs to be triggered into doing it and with malware it is no different [7]. Malware developers implement certain triggers in their malware which look for specific things within a system called artefacts before it actually activates itself and does anything malicious. This can also mean that the malware can be programmed into looking for specific environments such as a sandbox. A malware developer wouldn't want their malware activated in an environment where it can be potentially detected.
Because of that, testing malware by automation can become a problem which rises several questions. How can one know if the triggers within the malware sample are activated? How long do you need to test the malware for before moving to the next sample? Does the malware sample have sandbox protection implemented such as looking for specific artefacts?

1.1.1 Activating Malware

The answer is that you can't know exactly 100 percent of the triggers and protections but can, however, cover as much as possible of the common methods used to identify a large number of malware. Examples of observed triggers found in large sets of malware by simulating user activity [23]:

- File Creation
- File Modification
- Browser Activity
- Explorer Activity
- Certain Commands
- Specific File Present

From [23] it was found that there were potential false negatives (malware that didn't get activated but should have been) where known installed malware did not get activated which skewed the result of the experiment. There are several different factors as to why some malware didn't get activated where some are unrelated to the actual experiment. One case could be that the malware itself didn't execute properly i.e. something went wrong even though user activity triggered it. The system being

tested on could be incompatible with the malware or the hooking of the system doesn't work. It might also be a case of delayed activation as to not inform the user of potential problems. As mentioned earlier, malware can also be pretty sophisticated in that there are built in measures set in place to detect if the malware is being used in a place that it shouldn't be such as a virtual environment. Knowing what can trigger a large set of malware can result in several actions being performed on the system by the malware. This can be achieved by implementing different kinds of artefacts which are events or settings that malware responds to within a system. The outcome of the artefacts could result in the following actions being done by the tested malware. To capture the malware behaviour, tools such as Event Tracing for Windows (ETW) facility was used by [23] but can also be done via a memory dump and process tree analysis:

- File Creation
- File Reading
- File Modification
- Registry Querying
- Registry Modification
- Network Operation
- Process Creation

As seen from the list above there were several different outcomes of implementing artefacts which leads to malware changing its behaviour and execute its payload where any of them could have malicious intent. The problem with the testing done by [23] was that it wasn't very accurate in what action done in the system activated the malware. A malware trigger could be delayed within the malware as to not reveal what the cause of the activation was. Determining what malware looks for, such as specific artefacts before being triggered, during testing, is therefore something to be looked at and will be one of the main areas for the project.

1.2 Master Thesis Topic

The problem to be looked at for the master thesis will be around the topic of malware behaviour in a closed sandbox environment. The actual problem will be finding out if implementing common malware triggers or artefacts in a sandbox will result in more malware changing its behaviour during testing of said malware in different virtual environments as well as finding out what triggers or artefacts actually matter. Malware behaves different depending on what triggers for activation the developer implemented and as such can either be activated within a sandbox environment or remain dormant throughout the testing procedure of the malware sample. It is therefore necessary to look at the common artefacts and typical triggers by looking at what common triggers and artefacts other people have found through reverse engineering and testing of malware samples. By knowing how the malware behaves in a general sense, what it looks for within a system to trigger as well as how it detects virtual environments, a test platform can be created to analyse different malware samples with and without the triggers that they use. By doing so for a general amount of samples, a more efficient sandboxing environment can be created that can help in testing a large batch of

malware samples. The outcome of creating a test platform is to see more malware samples change their behaviour during testing while triggers and artefacts are implemented compared to when the triggers and artefacts are not implemented. The tests would in turn give results in terms of better understanding of how malware behaves and activates during the run time of a system. The data collected from the experiments can be used further in malware research.

The report for the thesis will start out with a preliminary analysis on malware and testing. Next, a discussion on the testing procedures and sandboxing will be done which will motivate why specific setups are desirable. The discussion will end with the type of platform that is going to be used for testing malware to determine potential triggers or artefacts that can be implemented. This will be followed by the system design for the project. The system design will describe the architecture and methods used for the malware testing procedures. Once the design has been set in place, the actual implementation of the malware testing will be discussed. This is followed by a testing and demonstration section which will present the result of testing with and without triggers/artefacts. A conclusion and discussion on future work will then end the report.
The project will therefore contribute with the following:

- Proof-of-concept testing of malware in different virtual environments.
- Implementing different system artefacts for altering malware behaviour.
- Assessment of malware behaviour change with implemented triggers/artefacts and without.
- Implementation of the testing platform with system modifications.

2 | Preliminary Analysis

Traditionally, malware and anti-malware research has consisted of running said malware in an environment where everything within the system can be monitored. With hardware, software and techniques evolving, malware developers keep changing and improving the effectiveness of their malware which in turn means that anti-malware researchers have to improve their malware detection methods.
As seen in Chapter 1, methods such as reverse engineering can't be used on all malware and results in more sophisticated methods being used such as machine learning where the behaviour of the malware is analysed. But as seen, the models used with machine learning can become outdated fast. Learning how malware behaves in the form of the triggers they use can therefore be another step in learning how to better test malware.
This chapter will take a look at general malware and how it behaves in terms of evasion techniques as well as what kind of triggers can be found to activate malware. The analysis is motivated by the problem of exploring a different way of testing malware by looking at specific triggers and artefacts that can increase the resolution of what actually activates malware more precisely. The analysis can then be used to explore how testing of said malware samples can be done.

2.1 General Malware

Malware in its general term is a piece of software that has the intention to and is designed to cause damage either in the form of rendering systems unusable of collecting personal information. Typically, this can be achieved by having the user execute a malicious piece of software that then can render a user's machine unusable or spread, typically over a network, to other machines to maximise the amount of infected machines that can be reached. A lot of malware uses methods to spread as fast as possible when it is first released since anti-malware research can be fast with finding and patching operating systems and anti-malware software to handle the newly released malware. Because of this, some malware delays itself with executing until it has spread to enough systems and machines to cause as much damage as possible in one fell swoop.
How malware spreads and infects machine can be done in several ways and to understand that, understanding the common types of malware that exists is needed. On the following Table 2.1, different types of malware can be seen as well as a small description of what it does.

Type	Description
Virus [55]	A virus modifies other system files in a way that when the file is executed, the virus is also executed. With how malware and anti-malware has evolved over time, virus based malware has become more uncommon. When a virus successfully infects other files, getting rid of it can become hard since antivirus software sometimes only gets rid of the original virus and not the infected files
Worm [54]	A worm is a self-replicating type of malware that has the purpose of spreading itself as much as possible. Compared to other types of malware, worms usually don't need any end-user actions to take place to spread itself. As such, worms can infect systems, devices, networks and infrastructure without the user doing anything. An example is e-mail based worms where the worm spread to a network after being opened in a e-mail by a company employee, infecting a company within minutes.
Trojan [62]	Trojans, as the name implies tries to hide itself as legitimate software. Compared to worms Trojans needs the user to activate them by running it. A typical example is a Trojan that masks itself as a system cleaner or anti-virus application which is likely to prompt the user to run it.
Ransomware [58]	Ransomware tries to take control of a system and locks the user out of it by holding it ransom. Just like a worm, ransomware can spread very fast and infect several machines across a network and if successful, can encrypt all files on a system, rendering the system inaccessible. Ransomware typically asks for a payment in the form of crypto currency and in return the files on the machine will be decrypted, but in some cases don't actually do it after the payment has been done.
Adware [53]	Tries to expose the user to malicious and unwanted advertising. Even though the adware itself can be harmless, the resulting ads that end up being accessed by the user can host malware. The damage done by adware can usually be remedied easier than other types of malware.
Spyware [59]	Malware that spies on its infected user. Spyware usually tries to gain access to personal data by infecting a user's machine with tools such as key loggers. Spyware can also be found in legitimate software that intends to monitor a system such as a company employees computer. This is often referred to as legal spyware or stalker-ware which are a problem in of itself [30].
File-less [56]	Takes advantage of exploits within other legitimate software and uses its memory as its host. This type of malware can be hard to detect since it takes advantage of already installed software that has legitimate uses such as Microsoft's PowerShell or Command Prompt.
Hybrid	A combination of the above mentioned types of malware that, when combined, can be very powerful. An example is malware that appears as a Trojan, spreads like a worm and infects like a virus or ransomware.

Table 2.1: The most common types of malware.

Malware keeps evolving and as such looking at new trends in what kinds of malware to look out for is needed [41]. Even though the end result is the same, how malware is made keeps changing by making it harder to analyse what the program consists of and how it executes its malicious behaviour.

2.1.1 Evasion Techniques

Since malware is malicious, developers want to hide their malware from anti-malware software. To do so, several methods can be used where the most to least common are as follows [57] [50]:

- **Fingerprinting** environment when executed to avoid detection [31] [33]. Also includes virtual environment where artefacts can be found.
- **Confusion** of automated tools. Avoid detection by dynamically chaining server used by the malware [14].

- **Time based** evasion. Malware can avoid detection by only executing at specific time, dates or when certain actions by the user are done.
- **Obfuscation** of internal data (packing, polymorphic code) to avoid detection by automated tools [65].

A trend that has been seen is that more malware being discovered is moving from the common to the least common evasion techniques by applying more evasion techniques in the malware [20]. This has resulted in a reduction in new malware over the years but has led to an increase in more dangerous malware that have a higher economic impact such as ransomware and cryptomining [46].
Another type of malware that has been recently seen is the so called Stegomalware which is considered one of the most sophisticated ways of applying obfuscation. [61] [10]. Stegomalware applies steganography which helps the malware hide from being detected by concealing files or network traffic via hiding and extracting malicious data within its own resources which is then executed dynamically on a system [60].

2.1.2 Time and Logic Dependant Malware

When talking about malware there are also another type that can be harder to work with than your typical malware. Normally when a malware binary is run, the machine gets infected right away and the malware payload has been set and executed, but some type of malware takes some precautions in the the form time or logic. The malware delays itself until some requirements are met either in the form of the so called time or logic bombs.

Time Bombs

Some malware that has the intention of either not be detected immediately or by maximising its spread, can be programmed to delay its malicious execution. Examples include executing during a specific time during run-time of the operating system or on a specific date.
Running on a specific date can be very powerful if the goal of the malware is to spread to as many machines as possible first. By doing so, the malware can create a coordinated attack by remaining dormant until a specific date has been reached.
Determining if this type malware is dangerous can be hard to achieve since some testing of malware can't wait until some arbitrary date has been reached. As such time specific malware can go unnoticed during testing.

Logic Bombs

Similar to time based malware, logic bomb based malware waits to execute until a set of requirements have been met. If the requirements have been met, the software sets off some malicious code that harms the host machine. Because of the nature of logic bombs, it is not necessarily only malware that incorporates it as seen by legitimate software having them [13]. Other cases where the logic bomb was intended for malicious purposes have also been seen where targeted attacks against several companies was done [44] [43].
Most malware makes use of logic bombs in form of the aforementioned triggers. Malware waits until something happens on the system which will trigger malicious code to execute within the software.

2.2 Malware Triggers

From what was seen from Section 2.1.1 and Section 2.1.2 it was seen that for malware to execute malicious code, some requirements are needed in the form of triggers. When looking at how malware activates, some can be trivial in that they can easily be reverse engineered to see the software's code and what needs to happen in order to trigger malicious code. However, a lot of malware obfuscates in order to avoid this and as such knowing beforehand what triggers or artefacts are needed cannot be done. By testing a lot of malware and using different triggers or artefacts, it can be observed that some malware will execute after a trigger has been done. From [23] it was seen that different malware can include a lot of different categories of triggers which can be seen as follows:

- File Creation
- File Reading
- File Modification
- Registry Querying
- Registry Modification
- Network Operation
- Process Creation
- Browser activity (related to network operations)
- Current window capture mode
- Mouse Movements
- Scrolling documents that are opened
- Specific Date Reached
- Virtual Environment Artefacts

2.2.1 Activating Malware Through Specific Triggers

Forcing malware activation can be done by simulating different triggers/artefacts on a system. By implementing specific triggers/artefacts within a system, malware can react to it and think it is running in a real system. It is however not trivial as to how the triggers and artefacts are implemented. As seen by [23] previously, implementing triggers to activate malware can bring an certain uncertainly as to what type of trigger or artefact actually triggered the malware. It was seen that knowing exactly what made the malware execute was hard to find out without knowing how the actual malware was designed since it can, as mentioned earlier, incorporate logic and time bombs to delay itself after being activated.
As such, other ways of changing malware behaviour through triggers/artefacts was done by [9] where the actual malware was analysed to see exactly what needed to be done for it to execute malicious code. This however requires that the malware can be read via reverse engineering and isn't obfuscated and packed so much that the code becomes unreadable. Tying triggers or artefacts to specific malware

can therefore be hard unless the parameters for testing can be changed such as increasing the time a specific piece of malware is tested, allowing more time for the malware to activate in cases where it might delay its execution.

2.3 Malware Platforms

When it comes to malware and the platforms it is being released on it can be seen that some platforms are more popular than others [12]. With Windows based PC's being the most used in the world it makes sense that malware developers target it since it has the highest chance of getting a victim infected with malware. With Windows being the preferred target for malware, anti-malware researchers also tend to focus on malware specific to that platform since it is easier to obtain and test malware on a Windows based environment.
However, other platforms such as Android based phones are also popular for infecting users since the process of getting malware onto someone's phone is easy with the platforms open based structure where offloading applications can be done without authenticating the source. This is a contrast to Apple based devices where applications are in a walled garden structure where it is hard to get malicious software onto the App Store where all apps must be downloaded from. Because of the open based nature of Android, the platform is also interesting to look at from a research perspective since applications running on Android are typically Java base which means that reverse engineering an Android application with a Java decompiler is easier than on Windows.

2.4 Problem Statement

The question that arises is how can malware be tested by implementing different triggers/artefacts which would result in change in behaviour for the malware? The challenges seen through this chapter come in the form of knowing how malware behaves, both in regards to activation and evasion techniques and how to implement artefacts/triggers that could lead to more malware being discovered which would otherwise remain dormant.
As such, the project and the proposed solution will focus on getting malware to change its behaviour by using triggers/artefacts implemented into a sandbox environment. Thus the problem statement for the project based on the preliminary analysis is as follows:

"How can a custom sandbox environment be designed by implementing artefacts to change malware behaviour and possibly trigger otherwise dormant malware?"

3 | Testing Malware and its Complexity

From Chapter 2 it was seen how different types of malware behaves and the different types of malware that exist as well different malware triggers. It was also seen how some platforms such as Android was attractive to do malware research on since reverse engineering malware made as Android apps was easier to do since most of the applications on Android are Java based. As mentioned during Chapter 1, the introduction, testing malware can be done in a lot of different ways where reverse engineering was one of them. As per [5], even though the amount of new malware seen has decreased the malware that is being released is increasing in complexity and is becoming more sophisticated as to how it is spread and executed. The authors also saw that even though malware types such as ransomware only accounts for under one percent of total malware on Windows, it is seen to have some of the highest impact in damage, both physically and economically since it renders the machines useless while infected. Due to the sheer amount of complex malware being released daily, the more popular technique to test malware is through the aforementioned sandbox environments which can either be on physical or virtual machines.
As such, there are several key areas that need to be looked into during this chapter to see how testing malware can be done in the most effective way. The key areas are as follows:

- What does malware look for when trying to see what environment it is running in?
- How do you simulate user activity?
- How do you make the machine look real?
- How can you extract data from an infected machine?

3.1 Malware Testing Complexity

To understand how malware can be tested most effectively, the first area to look into is the complexity of it and what is needed to be taken care of as to avoid malware knowing it is being tested. This includes the different key areas mentioned in the aforementioned list.

3.1.1 Reverse/Negative Triggers

As mentioned in Section 2.2, there are many different ways malware can be activated by depending on how the developer programmed the software. But looking purely for triggers in the system that will activate the malware is one of the areas where malware developers have gone a step further and have implemented evasion techniques as to not be detected by test environments. These techniques include the area of negative triggers.

Negative triggers are the triggers that the malware makes when it detects that it isn't in a normal environment such as a test environment. For an overall machine, there are several indicators as to what type of environment the malware is run in. The first area is the drivers used within a system. Many VM's use drivers that are modified and refer to the VM itself. A malware can see this and conclude that it should not run. Another area is the system itself. Depending on the wear and tear artefacts on the system, malware can decide not to run if the system doesn't seem like a regular system (system is stripped of everything except the bare-bones components). Reverse/Negative triggers can include many different things and can therefore be split into what is known as sandboxing artefacts, wear and tear levels and fingerprinting of a machine. As a result, anti-malware researchers have started to create "fake" virtual machines which hides a virtual machine as a real machine during the malwares detection stage [37].

3.1.2 Sandboxing Artefacts, Wear and Tear Levels and Fingerprinting

A technique that malware developers use to avoid their malware being detected by researchers is to take the reverse/negative triggers that can be found within the sandbox environment as mentioned in Section 3.1.1 and use them to avoid activating their malware in a test environment. Even if some of the most common components are spoofed to look like a real system, there are still many different elements within the sandbox that could be used to conclude that the environment is indeed a sandbox. Using different artefacts that are tied to the sandbox, sandbox fingerprinting can be done, which is a method to identify specific types of sandboxes. As seen by [64], using different characteristics of a sandbox, fingerprinting can be done to mark the system as a sandbox and avoid activating the malware. Fingerprinting is a real problem and a solution to this includes modifying the sandbox environment enough to not look like the previous made fingerprints of the sandbox environments. As per [29], there are two common scenarios when it comes to countering evasion techniques as seen on Table 3.1.

Evasion	Description	Counter	Description
A	The sandbox environment is typical of some known brand sandbox. The malware recognises it and evades detection.	A	Obfuscate and randomise the VM
B	The malware can detect the sandbox environment through a lack of user activity. For some malware to run, the user needs to enter a password from an email, click through a wizard or do other 'human' things. Many sandboxes do not emulate this and therefore do not see the malware detonate.	B	Simulate user activity

Table 3.1: Two common evasion techniques and their counters

Both solutions aren't trivial since malware developers will always continue to find new evasion techniques that build upon previous methods to always stay ahead of anti-malware researchers.

A way of fingerprinting a machine and find out of if a machine is indeed a sandbox is by looking for different artefacts within the system. As per [39], Sandboxes and personal computers have a lot of different system components that differ such as processes and files stored on the machine. The amount of artefacts will determine how much the system has been potentially used. Depending on the findings within the system a label can be put on the system which determines how much the system looks like a "real" system that a regular user would use. This label is called the wear and tear level of the system. The wear and tear level comes in two different types: direct user actions and indirect system actions. Former is actions done by the user and the latter is actions done by

the system itself without the user.
In the following Table 3.2, different categories (system areas) and their artefacts which were found by [39] are showcased. Note that not all artefacts are showcased here.

System Category	Artefact
System	Total Processes
	Windows updates installed
	System of System Events
	Application events
	Elapsed time since first system/ application event
	Drivers installed (GPU, LAN etc.)
Disk	Recycle bin size
	Number of files in recycle bin
	Size of temp files on system
	Number of system files
	Total size of process crash minidumps
	Num of crash minidumps
	Total size system thumbnails
	Files on desktop
Network	Number entries in the ARP cache
	Number entries in the DNS resolver cache
	Number of cached wireless SSIDs
	Number of active TCP connections
Registry	Size of the registry
	Number of registered uninstallers
	Number of programs set to autostart
	Legacy DLL reference count
	Number of registered application paths
	Number of Active Setup application entries
	Number of leftover registry entries
	Number of registered DLLs that do not exist on disk
	Number of entries in the UserAssist cache (frequently opened applications)
	Number of entries in the Application Compatibility Infrastructure (Shim) cache
	Number of Multi User Interface (MUI) cache entries
	Number of rules in the Windows Firewall
	Number of previously connected USB devices (DeviceInstance IDs)
	Number of previously connected USB storage devices
Browser	Number of installed browsers (Internet Explorer, Firefox, Chrome)
	Number of unique visited URLs
	Number of URLs typed in the browser's navigation bar
	Number of of HTTP cookies
	Number of unique HTTP cookie domains
	Number of bookmarks
	Number of downloaded files
	Time elapsed between the oldest and newest visited URL (days)
	Time elapsed between the oldest and newest HTTP cookie (days)

Table 3.2: Table showcasing different artefacts that can be found within a sandbox system.

Other examples of actual artefacts left behind by a virtual machine can also be found within the sandbox. These artefacts can be found easily by simply looking in the common areas of the system where such artefacts are usually located. Examples of artefacts in form of registry entries and files are as follows on Table 3.3.

By looking at the findings from [39], the difference in artefacts as seen on Table 3.2 differed a lot between different types of machines. As an example, [39] looked at three types of machines, a "real" one which was artefacts collected from a lot of real machines that people used, a common sandbox and a baseline which was a fresh install of windows. Here it was seen that the difference between a real system and the sandbox and baseline would be enough of an indicator to determine that the sandbox wasn't actually a real system since it resembled a baseline machine too much in terms of how

many artefacts were found on the system. As an example, the number of running processes running on each machine differed a lot where the real machines had an average of 94, sandboxes had 35 and the baseline machines had 41. As seen by the numbers, on average, real machines would have more than twice the amount of running processes. With those averages in mind, looking at the number of processes, or other artefacts, on a machine and comparing them to those measurements, determining if the machine is real of virtual can be done.

Category	Artefact
Registry	HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions HKLM\HARDWARE\Description\System\"SystemBiosVersion";"VMWARE" HKLM\HARDWARE\ACPI\DSDT\BOX_
File	WINDOWS\system32\drivers\vmmouse.sys WINDOWS\system32\vboxhook.dll Windows\system32\vboxdisp.dll

Table 3.3: Different artefacts left behind by a virtual machine.

These artefacts are common and are actively used by malware to detect if they are in a virtual environment [49] [47] [19]. The outcome of finding these artefacts is that the malware will decide that it shouldn't run and in some cases deletes itself completely from the system to avoid further detection.

During this section it was found out that there is a large difference between a "real" system and a virtual one such as a sandbox environment. The amount of artefacts found with each type of system differed a lot and was seen be a metric that could be used to find out if a system is real or virtual. In such cases, malware would be able to detect it and stop itself from doing anything malicious if it figured out that it was being run in a sandbox or virtual machine. This also means that in order to test malware using a base sandbox environment where artefacts are easily found would create a problem. In this case, a base sandbox would need to be modified to the point that it resembles a "real" system which could either be a physical system or a modified virtual machine.

3.2 User Machines vs. Sandbox Machines vs. Baseline Machines

To understand the difference of user machines and sandbox machines in terms of the difference it makes when testing malware, a look at what was discussed in Section 3.1.2 is done. During this section, it was seen how a "normal" machine used by a user and a virtual system behaved in terms of how many artefacts could be found within the system. Using several user machines as a base line it was seen that is in fact a major difference between virtual machines, user machines and fresh installed baseline machines. From [39] the three different areas are known as follows:

- User
  - Artefacts collected from a large amount of real machines.
- Sandbox
  - Artefacts collected from sandbox environments

- Baseline
  - Artefacts collected from fresh installed versions of windows

Here it was seen that the difference between machines that were actually used by people would have a lot more artefacts than virtual sandbox environments and fresh installed machines. As an example, key artefacts were picked out and can be seen on Table 3.4.

Artefact Name	**User**	**Sandbox**	**Baseline**
# of processes	94	35	41
# installed windows updates	794	19	2
Total size of recycle bin	2.5GB	50MB	0
# of files in recycle bin	109	3	0
Size of thumbnails folder	63MB	8MB	2.6MB
Size of registry	144.8MB	53MB	35MB
# of unique visited URLs	28K	14	2
# of HTTP cookies	3K	135	2
# of downloaded files	340	3	0

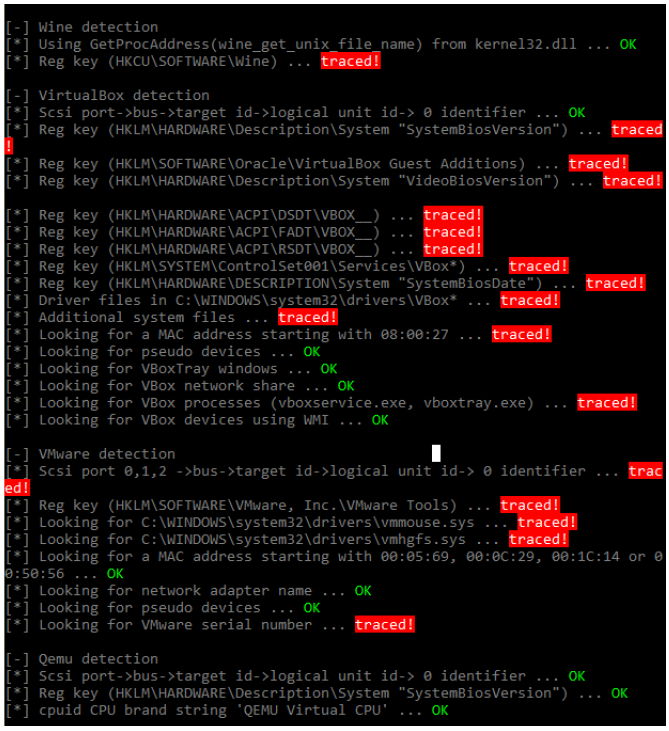
Table 3.4: Difference in artefacts between real, virtual and fresh machines [39]

As seen on Table 3.4, the user artefact category is always the largest in artefact count which makes perfect sense since the machines where the artefacts were collected from were from real machines that have been used for long periods of time. It can also be seen that the difference between virtual sandbox machines and the baseline was not that large and in some cases the baseline would have more artefacts such as the number of processes running.
As to how the artefacts were collected, [39] used a custom tool to scan a machine for known artefacts. Since the majority of artefacts are easily readable and available, such tools are not hard to create. Such tool is also important to use when a new sandbox for malware testing is going to be set up. Other than sandbox artefacts, which can also be found easily, the common artefacts can be found and quantified which can establish a baseline as to if the sandbox can pass a baseline test to see if the machine can be used for malware testing. If the baseline isn't met, the sandbox isn't ready to be used for malware testing.

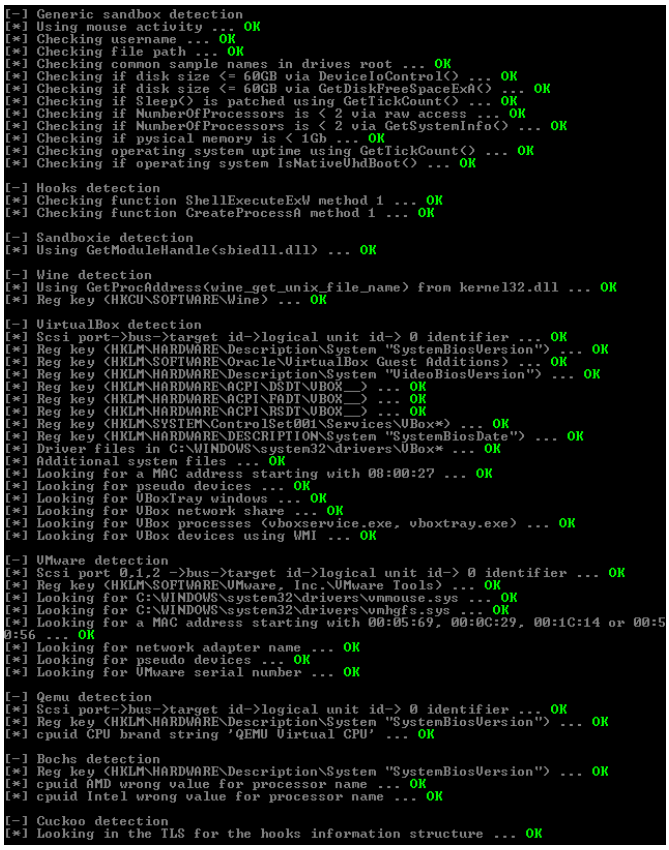
When discussing the difference between user and sandbox machines, deciding on what type to use for malware testing can be done. For user machines, artefacts seen in virtual machines such as Table 3.3 are not present which already removes extra checks needed to make the test environment. It is however more time consuming to use user machines since the wipe and install processes takes longer on a physical machine compared to a virtual one where the image can be replaced much more easily. With the resources available and the time constraint of the project, using a virtual setup is more efficient since a custom image of an operating system can be made before testing and re-flashed faster than doing the same process to a normal hard-drive on a physical machine. The only downside in using a virtual environment is the aforementioned artefacts that can be found by scanning the machine. The artefacts for virtual environments are therefore one of the areas that is looked at before creating the test setup that is going to be used for malware testing.

3.2.1 Finding Common Virtual Artefacts

To see the differences between a normal ”real” machine and a virtual one, scanning for common artefacts can be done. Tools for this exist such as paranoid fish (Pafish for short) [1]. Pafish scans for common artefacts found in different virtual machines and environments and showcases how easy it can be done as well as how common malware families does it. As mentioned earlier, common artefacts are stored as files either on disk or in the system registry. Finding those files can be done just by looking up the location of it which can be coded into the detection aspect of the program. Showcasing the results of Pafish can be seen on Figure 3.1 where a machine with artefacts and a machine without artefacts were scanned. As seen on the two figures and as expected, the machine which had artefacts present on the system were detected and displayed as such in the output of the program whereas the machine without the artefacts showed an output without any detection. In the example the machine with the artefacts were artificially created which mimic the location of real artefacts [37]. This was done simply by creating empty entries in the locations where the real artefacts would be located if the machine was in a virtual environment.



(a) Output from Pafish showing different detected artefacts



(b) Output from Pafish showing different missing artefacts

Figure 3.1: Pafish showing a machine with detected vs. missing artefacts

When preparing the system where malware will be tested on, Pafish can be used as a base layer of preparation for the system image. Using Pafish will show if the virtual machine being used is being masked as a real machine enough for common artefacts to not be detected. These anti-triggers will therefore immediately help with activating more malware being run on the system since the malware wouldn’t find any common artefacts on the machine. This, however, requires the artefacts to be

obfuscated while still maintaining the functionality of the virtual machine and not accidentally break the machine.
An extension of Pafish can also be made to include more than common virtual machine artefacts. Since a lot more artefacts can be present on the system as seen from Section 3.1.2, covering those within Pafish on top of what is already there will improve the base image used even more. This will also help with the comparison between a normal virtual machine and a custom base image with has those artefacts removed. It is therefore expected that if the malware run during the test has anti sandbox detection, the custom base image used would have an increased activation number of malware. Using Pafish will therefore show exactly what was done to the image in order to activate more malware during testing.

3.3 Data Extraction from an Infected Machine

A key area that hasn't been discussed so far is the data that will be generated when using a sandbox, whether it is on a virtual machine or on a physical machine. The data that is generated consists of a multitude of things that cover different aspects of the machine. From Table 3.2 it was seen that there are 5 major categories on a system where artefacts are located. It is also in these locations that changes can happen during testing of malware since it has been observed in the past and through [23] that malware can make changes such as:

- File creation/modification
- Web browser changes (default settings)
- Registry key creation/modification
- Windows related settings (firewall, network, standard apps)

Monitoring these key areas is therefore necessary in order to understand what a particular piece of malware is doing after it has been executed to run on a machine. Depending on what type of sandbox is used, the level of monitoring and logging of a system can differ. As an example, the Cuckoo sandbox has built-in monitoring and logging when a malware sample is being tested [15]. Here Cuckoo covers areas from the list above where several files are generated during and after testing such as:

- **analysis.log**
  Log file containing a report of process- and file creation as well as errors that occurred during execution.
- **dump.pcap**
  Network dump containing all network related activity during execution.
- **memory.dmp**
  Full memory dump of the entire environment during execution.
- **files/**
  All files that the malware operated on during execution (infected or modified files).

Sandboxes made to run malware are made specifically to not have malware interact with the log files as to avoid the problem of having infected log files. The exception is of course the files that the malware interacted with which are expected to be infected. Running a sandbox on a physical machine and not a virtual one can therefore create a problem when running tests and expect to have logs available afterwards. Not having the ability to extract information to a safe location makes it difficult to know if the files that are generated are clean of infected by the malware. Using virtual environments for malware testing is therefore preferred since there is more control over how the environments intact with the outside system as well as making it faster to take down and spawn new instances for testing. As an example, the architecture for the Cuckoo sandbox can be seen on Figure 3.2.

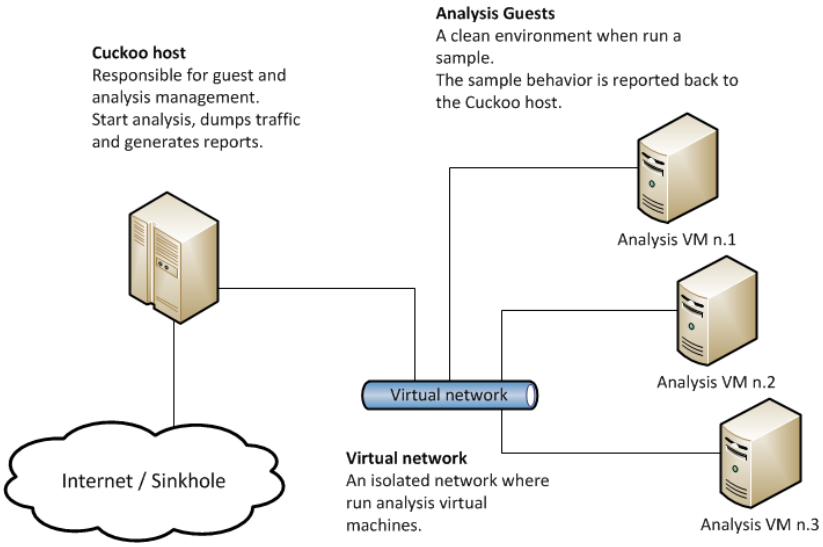


Figure 3.2: Overview of the Cuckoo sandbox architecture [18]

From the figure it can be seen how Cuckoo creates its own virtual network that connects the host machine to the virtual machines that are used for analysing malware. Running the virtual machines in an isolated network allows Cuckoo to safely execute malware and analyse it without having to worry about the malware infecting machines outside the isolated virtual network. Establishing a virtual machine with the same architecture as Cuckoo for this project in terms of separating the host machine and the virtual machines is preferred in order to avoid malware samples infecting other machines on the network.

3.4 Simulating User Activity

When testing malware on in a sandbox environment it is important to note that user activity is a big part of making malware think that a real user is actively interacting with the machine. One way that malware developers have tried to detect if they are in a virtual environment such as a sandbox, trying to look for user interaction has been done to determine if the machine is actually used by someone or not. Looking for user activity can be achieved by looking at user interaction elements such as mouse movement which was also one of the areas that Pafish was looking after when scanning for artefacts (Section 3.2.1).
Because of the increase in malware that looks for user activity, it is necessary to look into simulating a user to a certain degree in order to have malware behave in different ways and hopefully do

something malicious within the sandbox. As seen on Table 3.1 evasion B, simulating a user can either be random activity or through some exact actions done by a user. This depends on what the malware's purpose is and what it intends to achieve. An example could be a piece of malware that wants access to bank details from a user. Here the malware won't do anything until it detects that someone is perhaps typing on their keyboard and accessing some known banking site where the user would enter some personal details. Otherwise the malware can decide to remain dormant and not do anything malicious.
Simulating a user can be done in different ways and requires some setup prior to doing so. In a virtual environment it can occur that no actual human interface device (HID) is connected to the machine. If no HID is connected, malware knows that something isn't right and won't activate. It is therefore necessary to have the sandbox have HID's such as a keyboard and a mouse before trying to simulate a user activity.

Different types of tools for simulating user activity exist that can help make malware think that a user is interacting with the machine. The following list is just a sample of what can be found and used since there are many different tools available:

- **AutoHotKey [3]**
  Scripting tool made to automate and create macros for different tasks. The tool can do repetitive tasks where keyboard and even mouse movement can be done. Since AutoHotKey is open source, it is the preferred tool to use when starting out with user simulation/automation.

- **LoadRunner [34]**
  Tool used for simulating a lot of users in order to test software performance. Similar to AutoHotKey, LoadRunner can simulate keyboard and mouse activity to interact with different applications. LoadRunner is a commercial product.

- **AutoIt [4]**
  Lightweight scripting software for user simulation that can manipulate keystrokes, mouse movement and GUI windows to automate different tasks. AutoIt uses the BASIC scripting language which makes it more accessible to users without programming experience. AutoIt is a freeware product.

It is however important to point out that these tools are running on the sandbox. Because of that, malware can potentially look for these tools as well which would break the idea behind using the tools in the first place. Some malware even comes in the form of an user simulation where scripting tools such as AutoHotKey are used [26]. This type of malware will use tools such as AutoHotKey and execute different tasks such as looking for specific browsers, fetch malicious extensions and replace those already present and change windows specific settings.
Another possible way to simulate a user is through the use of an external device that can act as a HID but is in reality a device sending different commands to the host machine such as different keystrokes. An example of such tools is the Etherkey project which can take a micro controller and emulate a USB keyboard [24].
Other types of tools that can simulate user activity is malware analysis tools themselves such as malware sandbox tools. These tools have built in user simulation such as interacting with the system and moving the mouse cursor around the desktop. For these cases creating your own user simulation would be unnecessary when the work has already been done by the malware sandbox tools themselves.

3.5 Sourcing Malware Samples

Obtaining malware can be achieved in different ways. One way is to look for specific malware feeds which are a constant feed of new malware being discovered. Companies such as the renowned VirusTotal, who focus on analysing thousands of malware samples a day with many different antivirus software, have an API which can be used to fetch new malware samples which can be used for testing purposes.
It is also possible to find malware samples through what is called malware collections which can be found in many different places such as theZoo on GitHub [66]. These collections have a curated list of malware samples which can be used for different types of malware research.
Since malware exists in many different forms and can vary from completely broken and useless to high level threats, it is needed to be very specific about what samples are being gathered and used within the malware testing platform. In this case, the best option to use for testing malware, and the fastest, is to get hands on a curated collection of malware. As mentioned previously, things like malware feeds exist but can become difficult to use. The malware feeds are simply a feed of malware that has been found and analysed, doesn't matter how good or bad it is. However, some feeds have an indicator in the form of a rating as to how high of a threat an individual sample is. This can help quickly estimate if a sample can be used for testing specific applications. Since you have to check if each sample can be used, spending time developing and testing the actual system gets taken away. Because of this, another way to collect samples has been done.

A problem with some malware feeds is that they use a subscription based service for the public in order to download a large set of malware samples. Because of that, quickly getting a large set of malware samples can take time. However, some malware feeds offer malware samples for academic purposes.
Through Aalborg University (AAU) and employee Egon Kidmose, VirusTotal was contacted in regards to getting hands on a larger, curated batch of malware samples. As mentioned earlier, VirusTotal is a renowned platform for being the place where malware gets analysed and reported using all of the industry standard anti-virus and anti-malware software. As an example, scanning a simple file will go through upwards of 60 different anti-virus and anti-malware applications and report back their findings. These applications include known ones such as AVG, Malwarebytes, ESET, McAfee and more. The results of the scans include if the file was detected as malicious and what the result of it was i.e. what type of malware it is such as a Trojan. The output of them could look as follows after scanning a malicious file:

```
{
    "scans": {
        "Malwarebytes": {
            "detected": true,
            "result": "Trojan.VBClone",
            "update": "20190510",
            "version": "2.1.1.1115"
        },

        "McAfee": {
            "detected": true,
            "result": "Trojan-FGAU!BF9B26669990",
            "update": "20190503",
            "version": "6.0.6.653"
        },

        "ESET-NOD32": {
            "detected": true,
            "result": "Win32/VBClone.B",
            "update": "20190510",
            "version": "19331"
        },

        "AVG": {
            "detected": true,
            "result": "Win32:Malware-gen",
            "update": "20190509",
            "version": "18.4.3895.0"
        }
    }
}
```


Figure 3.3: Scans made by different anti-virus/malware companies

As such, using VirusTotal for malware sample analysis can quickly give an output of what the world's most renowned anti-virus and anti-malware applications find when analysing malware samples. By contacting VirusTotal, a batch of curated malware for academic purposes was obtained. This batch contains a wide variety of malware which is grouped together semi-annually and added to the archive. This means that the batch obtained from VirusTotal contains very recent malware that can be tested. The malware samples are also directly linked to VirusTotal's own site where the individual sample can be found. In total, from the 1s of January 2019 to the 10^{th} of May 2019, over 19 GB of malware samples were provided in the semi-annually archive. Older archives dating back to the beginning of 2017 were also provided but since that much malware can't be tested in a short period of time, using some malware samples from the newest archive has been done. The archive contains many different malware samples which means that only a few were selected to be used for testing purposes. Besides containing malware samples, VirusTotal has also provided an analysis of each malware sample that explains a little about how big of a threat it is as well as what the malware sample does in terms of type such as a Trojan.
In the archive provided the following structure is found:

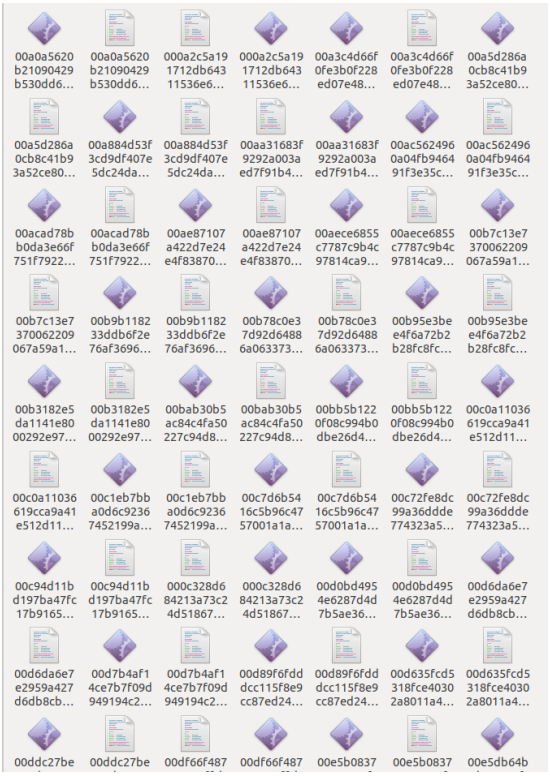


Figure 3.4: File Structure from the VirusTotal archive

Which means that the files come in pairs as follows:

```
1 Sample1.(exe)
2 Sample1.json
3 Sample2.(exe)
4 Sample2.json
5 ...
```

Figure 3.5: Malware archive structure

As seen above in the archive structure, each malware sample executable has as corresponding JavaScript Object Notation (JSON) file marked as ‘’.json”. This JSON file is what contains the aforementioned information about VirusTotal’s own analysis of said malware sample. Using this information, finding malware samples that fit with the coming testing procedures can be done faster than if each sample were to be tested first. One downside of the ‘’.json” analysis is that VirusTotal does not attach enough information to each malware sample. Having information such as how the malware behaves and how big of a threat each malware sample is would further help in filtering and identifying malware samples that would fit for testing in the project instead of selecting samples that are outdated and wouldn’t work in a modern system. This means picking malware samples still needs to be tested before determining if they can be used or not in terms of executing to be able to analyse their behaviour. To do so, the randomly selected malware samples were run for a short test within the VM to see if they would actually execute properly and not just crash upon starting.

3.6 Conclusion

From Chapter 3, the main take-away is that testing malware within virtual environments needs a lot of extra thought put into it in order to avoid having malware detect the closed off environment. Through the knowledge that we have about common artefacts within VM's and using tools such as Pafish, it can quickly be determined if a system is exposing any of the known artefacts or not.
During the chapter it was also learned how data extraction from infected machines could be done as well as how simulating a user could be achieved. In order to not invent the wheel again, tools exist that can handle these tasks in a safe manner in order to avoid spreading malware to innocent machines.
Sourcing malware was also discussed and was seen that gathering malware can be complicated where some sources are using subscription based services for the public. For academic purposes, big companies such as VirusTotal help by providing malware samples which can be used without payment.

4 System Specification

In order to design a system which can be used to solve the problem of systematically testing malware with and without triggers/artefacts as stated by Chapter 2 and Chapter 3, it is necessary to first specify what the system must do in order to fulfil its purpose of testing malware. The main goal of the system is to be able to test malware with specific triggers/artefacts and without in order to compare how effective implementing specific triggers/artefacts can showcase how malware behaves under certain circumstances. Further, the system can also be used to identify what artefacts matter and actually cause a change in malware behaviour. In this chapter, several use cases will be made which describe the different system components and how they interact with each other during usage of the system. Based on the use cases, a general overview of the system will be made which describes how the system will be used.

4.1 Use Cases

With the system overview described in Section 4.3, the different use cases for what the system should be able to do can be described in the following section. The use cases describe the system and how it performs different actions during usage which in this case will be during the testing of malware. It should be noted that the use cases define what the system should be able to do and not define different malware test that should be performed on the system.
The following list contains the different use cases:

- Use Case 1: Base Image testing
- Use Case 2: Malware test on Base Image
- Use Case 3: Modified Image testing
- Use Case 4: Malware test on Modified Image
- Use Case 5: Process Test Results
- Use Case 6: Compare Test Reports

To establish an overview of the system in terms of what defines it the terms modified and base are used. The terms are used to describe what the system as a base looks like and what the system looks like after artefacts and triggers have been implemented. This distinction needs to be clearly defined as to understand what changed within the system in cases where different malware behaviour has been observed. It can then be traced back and seen what changed within the system that could have influenced the behaviour change for a malware sample. As such, the two use cases, Use Case 1 and Use Case 3, explain how the system is tested for both the Base Image and Modified Image.

Using these two use cases will also lay the foundation for what is defined as a behaviour change. If a malware sample is run multiple times on the Base Image system i.e. nothing has been modified and the behaviour stays the same, but running the same sample on the Modified Image and it is observed that the behaviour change, it would be clear that what was changed within the system caused a behaviour change within the malware. As an example the behaviour change could be that the malware started sending request to the internet on the modified baseline and not on the baseline system. This would also require that certain parameters of the system being used stay the same between the tests as to avoid problems with malware that uses time bomb like activation as mentioned in Section 2.1.2.
The comparison between the base test (Use Case 2) and the modified test (Use Case 4) is done via a comparison in the results collected from the tests. By quantifying different events that happen during the tests such as system calls made by the malware samples process tree, a change in the amount observed between the tests can be seen. These are covered by Use Case 5 and Use Case 6.

Use case name	Base Image Testing - Use Case 1
Description	The Sandbox Guest needs to be prepared for testing and to achieve this a Base Image is made without any artefact modifications.
Actor	• User
Assumptions	Pafish is prepared to be run on the Sandbox Guest.
Steps	1. Windows VM is created with only changing what is needed to make testing possible. 2. Pafish is transferred to he Sandbox. 3. Pafish is run to check for artefacts, triggers and anti-triggers. 4. Output from Pafish is saved.

Use case 1: Base Image Testing

For Use Case 1, the idea here is to prepare the Base Image that is going to be used for testing malware samples and creating a so called baseline. With the Base Image, where nothing within the VM has been changed, some malware samples might detect that they are running inside a virtual environment if they have checks for it. This is because the Base Image hasn't had its artefacts removed that reference the virtual environment that it is in. As such it is expected that the malware samples would be less likely to execute and deliver their payload. Using Pafish will create a result that will show different artefacts that have been detected. The results will then be used in a comparison with the Modified Image.

Use case name	Malware test on Base Image - Use Case 2
Description	The Sandbox Guest is prepared for malware testing with the Base Image and a Malware sample.
Actor	• User
Assumptions	Base Image is set to run. A Malware sample is prepared to run on the Sandbox Guest.
Steps	1. Base Image is loaded using a snapshot. 2. Malware is loaded into Malware Analysis application. 3. Malware configuration is made pre testing. 4. Test is started and malware is run on the Sandbox Guest. 5. Test results are saved. 6. Base Image snapshot is loaded. 7. Repeat from step 2 with different malware sample.

Use case 2: Malware test on Base Image

For Use Case 2, malware samples are tested using the Base Image. To make the testing easier after every malware sample test, step 6 goes back and reloads a snapshot. This snapshot is a image of what the system looked like before malware was run in it. This makes it possible to test multiple malware samples after a malware sample has infected the machine it is running in.
After the test for a malware sample has been done, the results are saved.

Use case name	Modified Image Testing - Use Case 3
Description	The Sandbox Guest environment has been modified further than the original Base Image in order to see if different artefacts, triggers and anti-triggers have been modified.
Actor	• User
Assumptions	The Sandbox Guest has been modified and Pafish is prepared to be run on the Sandbox.
Steps	1. A clone of Base Image is made called Modified Image. 2. Sandbox artefacts have been modified by changing settings outside and inside the VM. 3. Pafish is transferred to he Sandbox. 4. Pafish is run to check for artefacts, triggers and anti-triggers. 5. Output from Pafish is saved.

Use case 3: Modified Image Testing

For Use Case 3, the same procedure as with the Base Image in Use Case 1 is done. The difference here is that the Base Image is taken and modified further by removing common artefacts from the system.

This includes references to the virtual environment as well as system settings such as hardware devices and other relevant settings such as network settings. Using Pafish again to find artefacts, the results are compared to the earlier results from the Base Image. This is done to understand if certain artefacts have been successfully removed from the system.

Use case name	Malware test on Modified Image - Use Case 4
Description	The Sandbox Guest is prepared for malware testing with the Modified Image and a Malware sample.
Actor	• User
Assumptions	Modified Image is set to run. A Malware sample is prepared to run on the Sandbox Guest.
Steps	1. Modified Image is loaded using a snapshot of the image that was modified earlier. 2. Malware is loaded into Malware Analysis application. 3. Malware configuration is made pre testing. 4. Test is started and malware is run on the Sandbox Guest. 5. Test results are saved. 6. Modified Image snapshot is loaded. 7. Repeat from step 2 with different malware sample.

Use case 4: Malware test on Modified Image

As with Use Case 2, Use Case 4 uses a prepared VM image and tests malware in it. Difference here is that the image used is the Modified Image. The idea here is that with all the artefacts removed from the system, the malware will have different behaviour than with the Base Image tests. When the tests have finished, the results are saved for later comparison.

Use case name	Process Test Results - Use Case 5
Description	The malware test has been run on either the Base Image or the Modified Image. The test results are saved with a memory image from the VM
Actor	• User
Assumptions	A malware test has been run for a malware sample without any error and the test results have been saved as well.
Steps	1. User has run the malware test either on Base or Modified Image. 2. Test results have been saved. 3. Malware Sandbox tool scans logs and memory image for malware behaviour. 4. Malware Sandbox tool saves results as a report. 5. Report is exported by the user and saved for later use.

Use case 5: Process Test Results

For Use Case 5, after a malware sample has been tested and the results have been saved, the data needs to be processed as to display what changes the malware did within the VM. The process is carried out by the Malware Analysis Tool which takes the saved VM memory image and scans through all changes that were made before and after the malware sample had been tested. When the analysis has been done, a report is made which is readable to the user.

Use case name	Compare Test Reports - Use Case 6
Description	After a malware sample has been tested on both the Base Image and Modified Image, a comparison between them can be made.
Actor	• User
Assumptions	A malware test has been performed for a malware sample on both Base and Modified Image. Reports have been saved and are prepared to be compared.
Steps	1. Reports for a malware sample are prepared. 2. The reports are loaded into the malware analysis comparison tool. 3. A comparison between the reports is made. 4. Any differences in the reports is logged. 5. Repeat for another malware samples results.

Use case 6: Compare Test Reports

For the final use case, Use Case 6, a malware sample has been tested twice, once on Base Image and once on Modified Image. For both tests a post analysis was done on the results and reports were exported by analysing the memory image that was saved during Use Case 5. The reports are then compared by loading them back into the malware analysis caparison tool. Any differences between the two reports are noted and determined if the change from Base Image to Modified Image had any impact on the malware behaviour. These differences include areas such as network activity, system calls and process/file creation within the system. As mentioned earlier, the differences are quantified as events that occurred during the test by monitoring the process tree of the malware sample. An example is having a malware sample have 10 percent of its events consist of system calls on the Base Image test while the malware sample has 20 percent of its events consist of system calls on the Modified Image. It can then be explored what those system calls consisted of which made up the increase in events.

4.2 System Requirements

The system requirements are based on a number of different things. The requirements consist of the different use cases as well as the system overview and what was discussed with the supervisors in terms of what the system could potentially become if it is going to be used in the future as a bigger system. In short, the purpose of the system is to analyse the behaviour of malware within closed environments by seeing how different artefacts impact its behaviour. The different artefacts that impact malware behaviour can then be used to simulate a more ”real” environment which would make it possible to explore how malware samples behave that include virtual environment evasion techniques such as looking for artefacts. The requirements listed influence how the system is going to be designed and how the system is interacted with when it is going to be used for malware testing. As such, a list of system requirements have therefore been laid out with what is considered a priority for the system to be made.

- System Scalability
- System Reliability
- System Dependency Reliance
- System Design Complexity
- Master Thesis Project Scope

System Scalability

One of the key areas and one of the important aspects of the system that needs to be taken into consideration when designing the system is the scalability of the project and system. The original idea behind creating the system was to make a system that would be easy to scale as needed depending on the amount of malware that should be tested at once. Because of this, the system would have to be made with scalability in mind. A way to achieve this is using open source applications and frameworks that allow themselves to be altered in order to be scaled into bigger system. As an example, the scalability would come in the form of having a centralised sandbox host which then can control several sandbox guests that run externally from the host itself.
The benefit of this would be to free up more resources on the host while still running many malware sample tests at the same time as well as vary the amount of artefacts that have been modified or removed which determine how ”real” a system is.
Another benefit is the future scalability of the system. If there are not enough guest machines available, adding more to the system and connecting them to the host would be possible, thus increasing the amount of malware samples that can be tested at the same time. This also makes the system more future proof since people would easily be able to add more systems without having to configure a lot themselves. Automation in terms of time allotted per malware test could also be increased as well as vary the settings for a realistic system between different types of malware by changing artefacts within a sandbox environment.

System Reliability

The second requirement it the system reliability. When using many different system components that utilise virtualisation and processing of memory dumps, errors can occur in many different places. It is

especially important when malware testing can take a long time and then suddenly crash because of something going wrong either with the host or guests. The causes can also vary a lot; the host could be overloaded because of too many malware sample request or the guest could be overloaded because the malware samples that were used for the testing made the machine crash. As an example with Cuckoo Sandbox, queuing too many samples would create problems when making memory dumps which would lead to memory related soft-locks on the host.
For systems that are intended to potentially be scalable to bigger systems, having system fall-back that can recover different things such as system crashing or data loss during processing is important.

System Dependency Reliance

A lesser but still important requirement is the dependency reliance. Relying on other peoples work or using tools or systems that don't allow for modification can cause problems when further developing the system. If something breaks, contacting the original author is the only option. Because of that, using open source tools and applications as well as making components yourself that are well documented are preferred.

System Design Complexity

The complexity of the system is also an interesting and important part to take a look at when considering that the system could be used by others in the future. Having the system be built as simple as possible with as much of the details written down is important for other people to understand when interacting with the system. The more components that get added to the overall system, the more the system can quickly become overly complex and confusing. Creating a system that uses the least components, protocols and subsystems is important which can also lead to a better performing overall system.

Master Thesis Project Scope

The last system requirement is the scope of the project. Since only one person is working on a project and subject that could be split into different smaller projects about malware behaviour and their triggers/artefacts, creating a system that can make a definite answer is not possible since you would have to test thousands of malware samples to have enough data to give a definite answer as to what trigger/artefact changes the behaviour of specific malware samples.
Therefore, the goal and scope of the project is to create a proof of concept system that can be used for malware sample testing as well as come with systematic way of testing malware samples with and without triggers/artefacts. This includes the different ways of testing as well as how common triggers/artefacts are implemented into a system. If the implementation works, looking into more uncommon triggers/artefacts can be done which would require testing over a longer period of time.

4.3 System Overview

The idea behind making a sandbox for testing malware is to increase the knowledge on malware behaviour by observing what happens when malware is executed when certain triggers/artefacts are present and what happens when it isn't. The system is a proof of concept which will be the base for a potential larger system. It is therefore important to lay out the major different components which are an essential part of the system. The system itself consists of three components, *Malware Database*,

Sandbox Guest, *Sandbox Host*, where the *Malware Database* is the malware used for testing on the *Sandbox Guest* and the *Sandbox Host* which controls the *Sandbox Guest* and takes malware samples from the *Malware Database*. The *Sandbox Host* also handles logging as well as the processing of the malware samples after the tests have been conducted within the *Sandbox Guest*. Each component and its set of sub component are used to represent different use cases that can be seen under Section 4.1. The idea behind the components is to show how a system consisting of several components can interact with each other and how malware samples are tested. When the Sandbox is ready to be used, malware samples from the Malware Database are loaded into the Sandbox Host and prepared to be executed within the Sandbox Guest. This means that only one sample is present at a time. When the test is completed for a malware sample, data generated on by the Sandbox Guest is then saved onto the Sandbox Host where the logs can be processed to see what happened within the Sandbox during testing. Using this setup, different use cases can be tested where different conditions for the Sandbox are made such as simulating user activity.
On Figure 4.1 the different components, their sub components and their interaction can be seen.

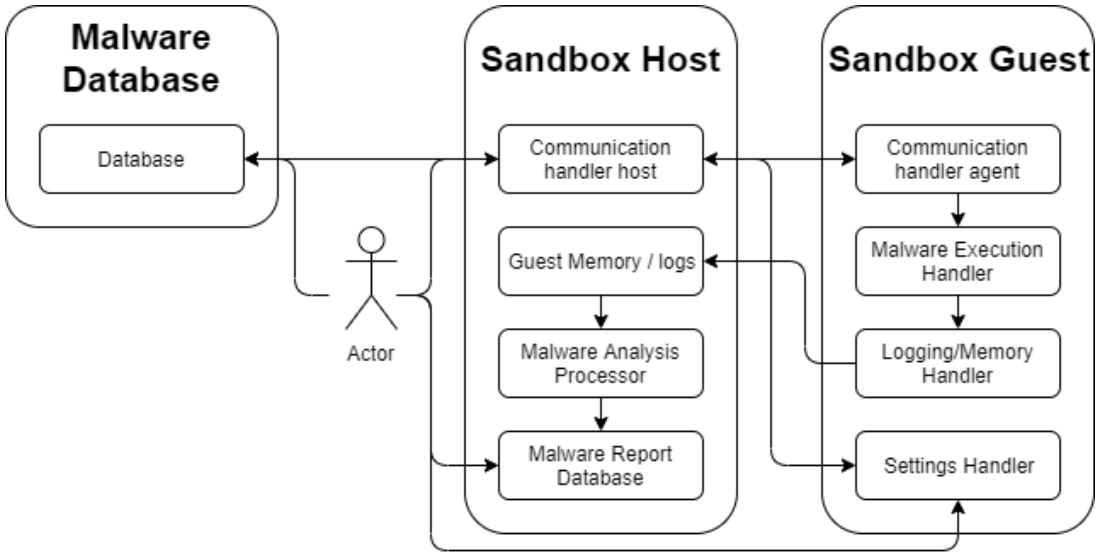


Figure 4.1: Diagram showing different system components

Malware Database

Inside the *Malware Database* there are 1 component.

- **Database**
  Here the entire malware collection is located which is interacted with by the Actor and the Sandbox Host. Besides the malware executables, information about each sample is also located here which describes what type of malware each malware sample is.

Sandbox Host

Inside the *Sandbox Host* there are 4 different components.

- **Communication Handler Host**
  Here the Sandbox Host can communicate with the Sandbox Guest in order to load and execute malware samples and after that, retrieve information such as logs and memory images that need

to be processed. From the Sandbox Host, updates can be sent to the Sandbox if anything needs to be changed in terms of running malware samples and execution them. The communication handler also has access to the malware database which is where malware samples are taken from.

- **Guest Memory / Logs**
  Here information about tested malware samples on the Sandbox Guest is stored. This includes logs as well as memory images that were saved from the Sandbox Guest. From here processing of the data can begin.

- **Malware Analysis Processor**
  Here the data that was collected from testing malware samples is processed. Processing the data by looking at a memory image and malware process tree reveals what happened during testing in terms of what the malware sample did such as changing system settings and creating more malicious files. As mentioned earlier this is measured as events that occurred during testing.

- **Malware Report Database**
  After the malware sample test data has been processed, a report is made which sorts everything found during processing in a more readable manner. This can then be used to compare different tests to find any change in malware behaviour by looking at the numerical number of different events that happened during testing such as the amount of file creations and registry edits. As an example for automation purposes, looking at the events can be done and if the different tests differ a certain amount, a flag can be set to indicate that the malware sample potentially changed its behaviour from one test to the other.

Sandbox Guest

Inside the *Sandbox* there are 4 different components.

- **Communication Handler Agent**
  Here the Sandbox Guest can communicate the Sandbox Host. The agent makes sure to fetch information from the host such as what malware sample to test and what settings to apply before executing said malware sample.

- **Malware Execution Handler**
  The malware execution handler, which is a part of the communication handler agent, can control how the malware sample is run on the machine when everything is ready. The execution will start the malware sample and if needed, start the user activity simulation. Here the user activity simulation for the Sandbox is done if specified by the Sandbox Host. This is where scripts to simulate user such as mouse movements are done.

- **Logging / Memory Handler**
  Here the Sandbox Guest logs everything relevant to malware testing such as network, memory and other system relevant dumps. The logs can then be transferred to the Sandbox Host when the malware sample testing is done.

- **Settings Handler**
  Here, the settings relevant to the VM are stored. This includes the triggers/artefacts that were implemented as well other relevant settings such as network related settings.

4.4 Malware Samples Test Requirements

In order to score different malware and understand if any behaviour change was observed between the different test that will be made, it is in order to look at the different criteria for what defines a change in behaviour for a malware sample.
In the case of testing malware in a sandbox, some objective criteria can be made, however, those criteria can't be associated with every piece of malware since not one piece of malware that will be tested will behave the same as another piece of malware. Because of this, a border of what is enough and what isn't can't be established is an indicator for a change in behaviour.
As an example, here is a list of changes to look out for between testing different malware samples. If a change between tests can be observed, it can be concluded that a change in malware behaviour was also observed.

- Score created by sandbox (if available).
- Creates executable files on the file system.
- Change in number of system calls (API and Process).
- Change in network traffic (links and downloads).
- Change in Windows related settings (firewall etc.).
- Process injection.

It is however important to note that if a change in behaviour is observed, knowing exactly what caused that behaviour in terms of artefacts is impractical with the test design established. The reason for this is that the changes that are made from one test to another are so large, that pinpointing what specific change could have caused the malware to behave differently is virtually impossible. A way to alleviate this problem could be to test different configurations where the artefacts used differ between configurations. Doing so could help pinpoint what combination of artefacts impact a malware sample the most.

For the testing of malware there are several methods as to how testing can be done. Depending on the time requirement and the results one might expect, the testing can either be done over a short period of time where the malware and system are monitored or over a longer period of time where the test concludes when the malware becomes inactive over the course of the testing procedure. The benefits of the former is that malware might timeout or remain dormant for a short period of time and then re-activate later, but comes with the cost spending more time with each malware sample. With the estimated time for testing malware being almost an hour per sample for short tests, having longer tests beyond that would prevent the testing of a large amount of malware samples. The benefits of the latter are that each test is much shorter since the monitoring simply waits for the executable process to finish. The isolation means that the tests can be done much faster. However, the short tests could miss something crucial that could happen as a result of the malware delivering its payload that could impact the system after the malware sample has stopped its execution. In terms of testing with potential triggers/artefacts, one could test a single change within a system at a time, but comes at the cost of there being a lot of different artefacts and combinations to test for. It is therefore important to lay out that testing with artefacts will happen by testing with many

artefacts at a time. This is because the testing of malware samples focuses on its behaviour, not the actual artefact. If the malware changes behaviour because of the many different artefacts that are implemented, it would be a success.
Implementing the triggers/artefacts is done in the same vein as altering different things within the system, since the artefacts are related to different components on the system. The malware is run on the system and a system-wide logging procedure occurs as well as taking a memory image of the systems last state.
As for the requirements for what happens between the tests, the design of the malware analysis is important to lay out (is described in more detail later). The basic understanding here is that one test will, as mentioned earlier, consist of a Base test. This test will only include what is necessary to execute malware in a virtual environment i.e. data collection applications and the virtual environment itself. The other test that will be conducted after the Base test will come in the form of a Modified test. This test takes the same base from the Base test and build on top of it in terms of removing or changing different artefacts within the system. An example of removing artefacts could be the references to the virtual environment that the malware is tested within.
After the tests have been conducted, a comparison between the Base test and the Modified test will be done. This comparison will show what happened for each test as well as show if anything changed between the two tests. It should be noted that even if one malware sample changes behaviour, another malware sample might not. This all depends on how the malware was programmed and if it was even made with the thought of trying to detect virtual environments and sandbox environments. A good indicator here would be the score generated for each malware sample. The score can be used as an indicator if anything differently happened between the tests. Of course, the tests might not change in score and because of that, a more in depth investigation of the results are needed just in case something else changed that wasn't reflected in the score.
The process of testing and analysing the malware samples can be seen on Figure 4.2.

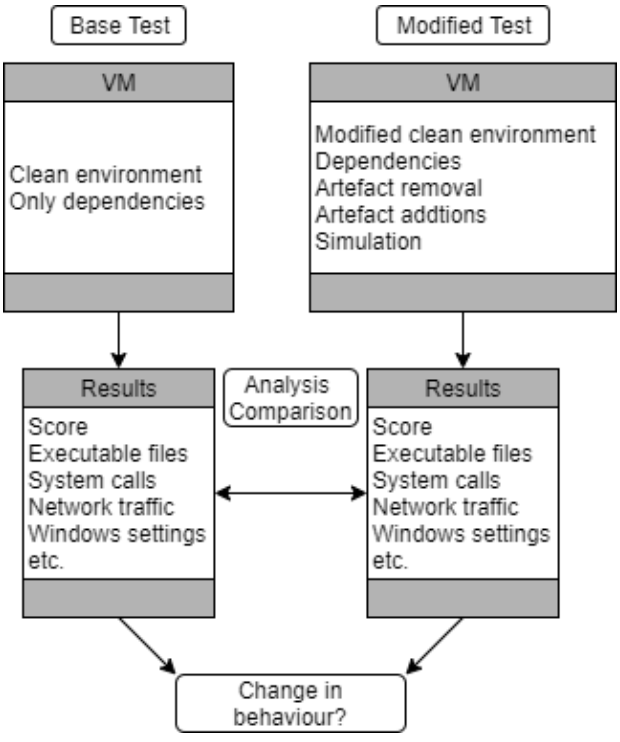


Figure 4.2: Process of testing and analysing malware test results

5 | System Design

The system design chapter will describe the general design for the system and look into the different considerations made when deciding how the system should be made.

5.1 Creating a Malware Test Setup

To create a setup for testing malware and how triggers/artefacts affects its behaviour, a test setup is needed. Testing on your own hardware can be risky in that malware could potentially escape a virtual environment and infect the actual machine and not only the virtual machine. In this case a platform at Aalborg University will be taken advantage of in order to test malware in a more controlled environment. From Chapter 4 it was learned that the setup where the testing of malware should take place needs some flexibility in the form of running and saving snapshots of virtual machines. This is needed since modifying the virtual machines needs to be done for multiple tests in order to see a behaviour change for malware that is tested. Not only does the modifications need to see a change in malware behaviour but also to eliminate common artefacts that are found within commonly used virtual machines such as VirtualBox and VMware. From Figure 3.1 it was also seen that depending on the virtual machine used, the amount of artefacts could differ a lot since their implementation of how they handle virtualisation differs. Another aspect is the security of the test setup. Running live malware, especially malware that can spread through the network, is very dangerous and if the local network contains other machines the malware could potentially spread to them. The design of the current setup at Aalborg University will be looked and thereafter the design of the proposed setup will be looked at.
Because of what was mentioned above as well what learned during Chapter 2, Chapter 3 and Chapter 4, some restrictions need to be applied to the system for both security and usability. As such, the restrictions are as follows:

- Using VirtualBox and Cuckoo Sandbox
- Restrict access from VM to outside internet
- Managing snapshots and modified Operating System (OS)
- Upload malware samples into the sandbox

To summarise the list of restrictions from above there are some key aspects that are crucial for the system to function properly and at an acceptable level.
The first area is that the system components to be used is VirtualBox and Cuckoo Sandbox which are compatible with each other and is the recommended way to run a malware sandbox without building something from the ground up.

The second area is restricting access to the local and outside internet. A lot of malware operates by communicating with the outside internet for downloading malicious content but also uses the local internet in order to sniff out other machines on the network in order to infect them. Making sure that the VM cannot access anything outside of its own environment is therefore a high priority. A downside of doing so is that an artefact has been introduced in the sense that there is no internet available which some malware can use to determine if they are potentially inside a sandbox.
The third area is the management of snapshots used for testing. Testing malware is as mentioned earlier done through the use of multiple snapshots where one is used as a baseline and one for the artefact implementations. Managing them properly in order to test multiple use cases is therefore needed.
The fourth and last area is how malware is uploaded to the VM. The malware should be contained on a system where the malware cannot execute by accident. An example here is using Linux for malware storage and Windows for the VM.

5.1.1 Current Setup

To be able to implement a modified version of a sandbox using Cuckoo, understanding the current setup at AAU is needed which is available to use.
The current setup consists of a bunch of high end servers that are made to run for long periods of time for applications such as machine learning. For this project, the purpose goes away from the machine learning aspect with really long testing and more towards smaller tests. The high end hardware such as Graphics Processing Unit (GPU)'s is also not needed since no intense use of GPU powered rendering is needed.
The AAU setup is interacted with as if it was any other computer where are full desktop is available for the user to use. As such, creating a setup using the current AAU setup would work as if you already exist inside a virtual environment, meaning that the system you are working in is the actual sandbox environment. As an example on Figure 5.1, a dummy setup showcasing how a sandbox environment on the current AAU setup is seen.

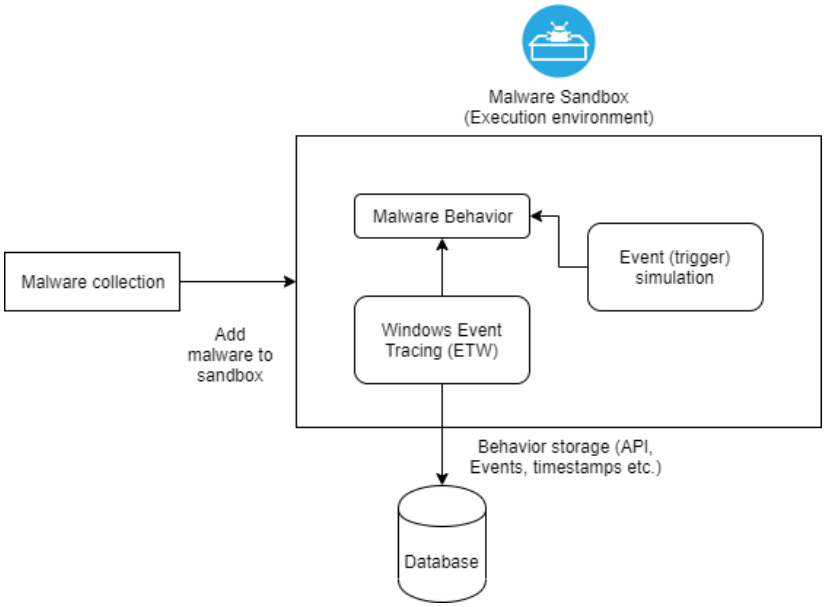


Figure 5.1: Example architecture of a malware testing environment

It can be seen how external malware is loaded into the sandbox after which monitoring, logging and simulation happens within the sandbox. In this scenario, no tools for handling the sandbox are taken into account and as such, handling the execution, monitoring and logging needs to be made and implemented within or outside the sandbox. This shows how the current setup has some flexibility in that one if free to implement whatever they want within the sandbox but can become a problem to implement. One of the problems with Figure 5.1 is that such an environment is very risky to create since everything is handled within a single environment. What is needed here is to modify the setup to create a second layer that can handle the actual sandbox. By having the second layer, more control over testing, including scalability, can become available and avoids being locked down to a single sandbox instance.

5.1.2 Modified Setup

Since the current setup is insufficient in terms of reliability and scalability, a modification of the setup is required. The modified setup is based on the architecture of the Cuckoo Sandbox which includes separating the host from the sandbox environments which decreases the chances of malware infecting the host running and collecting data from the tests. An overview of this setup which runs on the AAU setup can be seen on Figure 5.2.

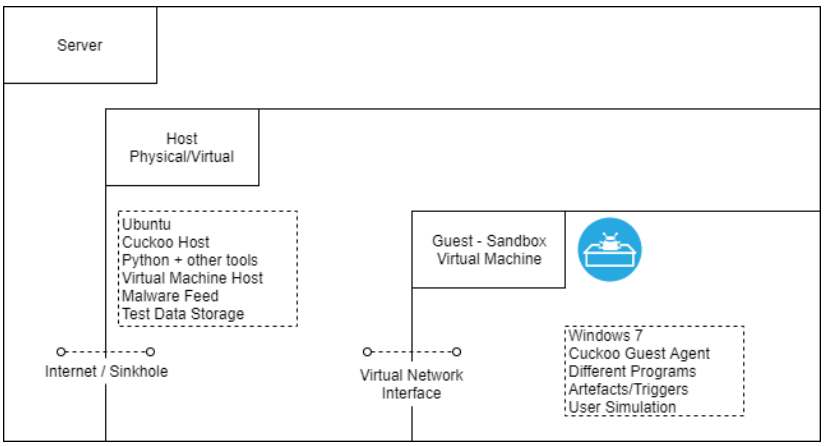


Figure 5.2: Architecture of modified setup

The setup as seen on Figure 5.2 might look fine, but after looking closer at the different components it can be seen that it is impossible to implement with what is available with the current AAU setup. Since the virtualisation part of it uses VirtualBox, having nested VM's is impossible since it isn't supported by VirtualBox since the CPU within a VirtualBox VM doesn't have the virtualisation extensions that a physical CPU has. As such, the setup needs to be reworked yet again. Since nothing major is allowed to be installed on the physical server a different approach can be made. Instead of using nested VM's, Docker can be explored as an alternative to run Cuckoo. In this scenario, Cuckoo and the Sandbox VM run separately, but will still be able to communicate with each other by forwarding the necessary network related traffic through the server and virtual network interface. The architecture for the second modified setup can be see on Figure 5.3.

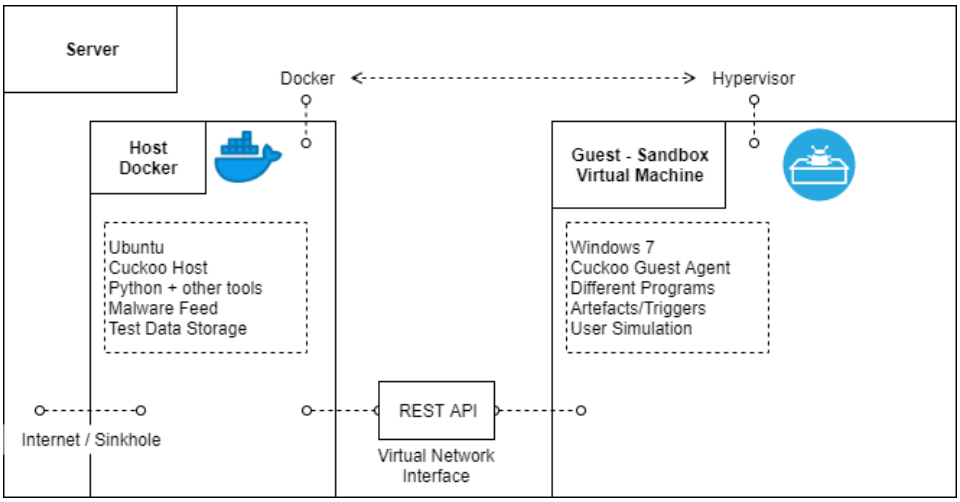


Figure 5.3: Architecture of second modified setup

As seen on Figure 5.3, the architecture is different in the sense that the VM and Cuckoo Host have been separated. This avoids the problem which was mentioned earlier where the VM couldn't run inside another VM since now there is only one VM. However, since the AAU setup doesn't allow programs to be installed directly on the server as root, the Cuckoo aspect needs to be encapsulated somehow. As shown by the figure this is achieved by using Docker [21]. Docker is similar to the way virtualisation works in that the application that you want to run is isolated from the physical machine, which in this case is the AAU server. However, Docker doesn't rely on a lot of the requirements that an actual VM needs such as an OS for it to function. This makes Docker ideal for applications such as micro services and distributed systems where small to medium applications can be duplicated for scalability. This also makes it possible to explore running the Cuckoo Sandbox within Docker to avoid the problem of not being able to run Cuckoo on the AAU server. As for the more in depth differences between Docker and Virtual Machines, more info can be read about it in Appendix A (Docker and Virtual Machines) where the differences between what and how the different technologies work compared to each other is explained in more detail.

In theory, now the only work that needs to be done around the design aspect of the system is linking the communication between the VM and the Docker Host where Cuckoo is located. From the figure it can be seen that it is achieved through two communication pipes known as Docker-Hypervisor and Representational State Transfer (REST) Application Programming Interface (API). The communication pipes are where, in theory, data is sent back and forth where the data sent consists of Cuckoo logs and monitoring and managing the VM such as starting the machine and resetting the snapshot. This can be achieved by setting up some specific network settings that point the two machines to each other. For the VM, using the built in ipv4 settings can be used and for the Docker-Cuckoo setup, using Docker and how it exposes services through IP's and ports can be utilised which routes the Docker containers to the host machine and to the VM.

However the setup as shown on Figure 5.3 is also deemed to be a problem to get up and running. The problem with constructing such setup is that Cuckoo, VirtualBox and everything needed to make them communicate with each other isn't officially supported when using Docker containers. Because of this, setting up such system is going to be difficult. One of the problems with using the modified setup is related to Cuckoo and its interaction with the virtual machine. Cuckoo wasn't made to run inside of a Docker container and as such a custom Docker image for Cuckoo and its dependencies must be made. The problem with this is, as mentioned, the interaction with VirtualBox.

Traditionally, Cuckoo is running on the same machine as the virtual machine and as such, can see the installation and tools for VirtualBox. In this case the tool ”VBoxManager”, which comes with the VirtualBox installation, is used by Cuckoo to manage the virtual machine in regards to snapshots, shutdowns, reboots etc. When inside a Docker container, Cuckoo can't see VBoxManager which makes it not function since no interaction can be made with the virtual machine. A workaround is therefore needed that can take inputs from Cuckoo and pass it to ”VBoxManager” which resides on the physical servers.
However, some people have tried to get such system design up and running. Known as ”Blacktop” on GitHub [8], they have tried to take the Cuckoo Sandbox and implement it within several Docker containers (one for Cuckoo and several for other services). With limited success using other specific remote virtualisation software such as VMware ESX, VMware vSphere and XenServer. Implementing their work becomes a problem since the VM to be used for the project is VirtualBox.

Blacktop Docker-Cuckoo Design

As mentioned earlier, Cuckoo and its dependencies weren't meant to run inside of Docker containers. Taking a look at the work that Blacktop have made it is clear that a lot has been modified in general in order to make Cuckoo work as well as modifying the network related settings in order to route different data properly. At first it seems like using the project might work but as mentioned earlier, there is no current support for VirtualBox which is what is used for the project. The problem with the support is that the tool which manages the actual VirtualBox called VBoxManager can't be directly accessed through Docker. However, Blacktop with some external help from another user called Ilyaglow [25] have come up with a solution to make VirtualBox compatible with Cuckoo inside a Docker container. Doing so requires exposing the Docker container for network access and utilising a dummy version of VBoxManager that runs inside the Docker Container. The trick here is that the dummy application uses the exposed network to communicate outside of the Docker container and communicate with real VBoxManager. In short, an application is made which just redirects commands to the real VBoxManager outside of the Docker container. An example of it can be seen on Figure 5.4.

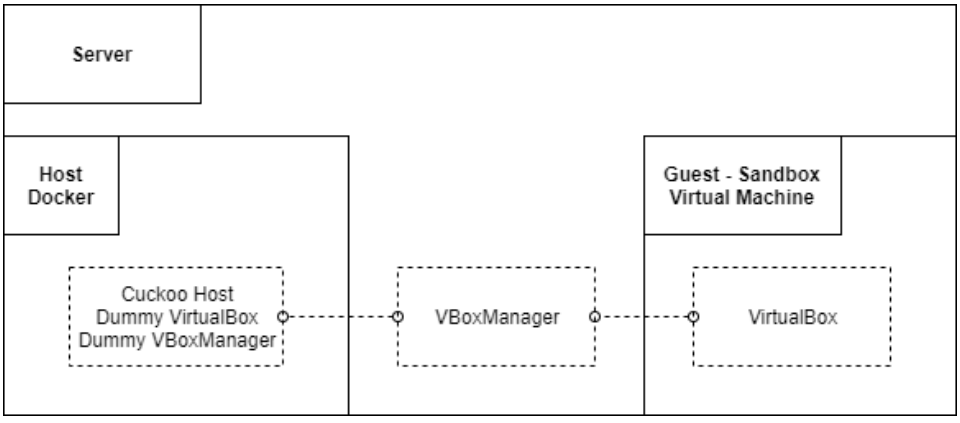


Figure 5.4: Diagram showing how VBoxManager works with Docker

On the figure it can be seen how the dummy VBoxManager communicates with the real VBoxManager which then interacts with VirtualBox.
With all the fixes to interact with VirtualBox being installed, setting up Cuckoo in Docker is done by executing the docker-compose script. The script, which comes in the form of a YAML file, handles

setting up all the different Docker containers that are needed to run Cuckoo Sandbox. The script also handles exposing the containers for network related settings which are needed to communicate with VirtualBox outside the Docker container. The different applications are set up as different services which run independently of each other but can still rely on each other. This is done by running each services dockerfile configuration which constructs its Docker container. As an example, the following code on Figure 5.5 is the docker-compose information for setting Cuckoo up in a Docker container. Other services which aren't shown include the web interface, API, MongoDB (database), Elasticsearch (RESTful search engine) and Postgres (database).

```
1  services:
2    cuckoo:
3      build:
4        dockerfile: vbox/Dockerfile
5        context: .
6        args:
7          - CUCKOO_UID=1000
8      command: daemon
9      ports:
10         - "2042:2042"
11     volumes:
12       - ./cuckoo-tmp/:/tmp/cuckoo-tmp/
13       - /mnt/cuckoo-storage/:/cuckoo/storage/
14       - ./vbox/conf/:/cuckoo/conf/
15     env_file:
16       - ./vbox/config-file.env
```

Figure 5.5: Snippet of what is inside a docker-compose file

From the .yaml file, the dockerfile `vbox/Dockerfile` on line 4 is referenced which as mentioned earlier constructs the Docker container wherein Cuckoo is running. A snippet of that file can be seen on Figure 5.6.

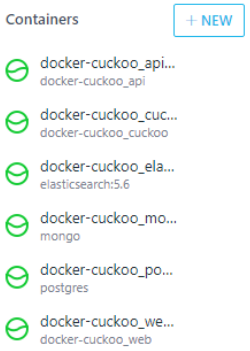
```
1  FROM blacktop/volatility:2.6
2
3  LABEL maintainer "https://github.com/blacktop"
4
5  ARG DEFAULT_CUCKOO_UID=1000
6
7  ENV CUCKOO_VERSION 2.0.6
8  ENV CUCKOO_CWD /cuckoo
9
10 # Install Cuckoo Sandbox Required Dependencies
11 COPY 2.0/requirements.txt /tmp/requirements.txt
12 RUN apk add --update DEPENDENCIES
13 COPY 2.0/conf /cuckoo/conf
14 COPY 2.0/update_conf.py /update_conf.py
15 COPY vbox/docker-entrypoint.sh /entrypoint.sh
16
17 RUN chown -R cuckoo /cuckoo && chmod +x /entrypoint.sh
18
19 WORKDIR /cuckoo
20 VOLUME ["/cuckoo/conf"]
21 EXPOSE 1337 31337
22
23 ADD /cuckoo/machinery/virtualbox_websrv.py /machinery/virtualbox_websrv.py
24 ADD /cuckoo/common/config.py /common/config.py
25
26 RUN chmod 644 /common/config.py /machinery/virtualbox_websrv.py
27
28 ENTRYPOINT ["/entrypoint.sh"]
```

Figure 5.6: Snippet of what is inside a dockerfile file

From Figure 5.6, the dockerfile goes through several steps. First, the backbone is gathered on line 1. On this line one would usually specify the OS that the container should be based on. In this case the container is based on a modified Linux Alpine OS which Blacktop has made. From there requirements and dependencies to get Cuckoo installed are done from line 10 to 15. On line 19 to 21 the working directory, volume and ports to expose are defined. Then on line 23-26 the modified VBoxManager is added as a python script with some configuration files. On the last line, 28, the entrypoint is defined which is the file that will be run when the Docker container starts. The entrypoint file specifies how the container should run such as what programs to start.
When the dockerfile and Docker-compose files have been set up, running the docker-compose.yml script gives the following output which can be seen on Figure 5.7a and Figure 5.7b.

```
sudo docker-compose -f docker-compose.vbox.yml up -d
Creating docker-cuckoo_cuckoo_1          ... done
Creating docker-cuckoo_elasticsearch_1   ... done
Creating docker-cuckoo_mongo_1           ... done
Creating docker-cuckoo_postgres_1        ... done
Creating docker-cuckoo_api_1             ... done
Creating docker-cuckoo_web_1             ... done
```

(a) Starting docker-cuckoo using docker-compose



(b) Interface showing the different launched Docker containers

Figure 5.7: Setting up docker and its output

At first it seems fine with the Docker UI reporting that the containers are running as expected, however, the containers soon after run into problems. Since the project made by Blacktop hasn't been worked on in years and not actually testing the VBoxManager implementation, the Docker containers run into fatal errors which results in crashes happening. At this point, working with several years old software that someone else has worked on becomes a problem and a cat and mouse game where every time a fatal error is fixed, another one appears in some other area which needs to be fixed. Because of this and with the limited available time to actually work on the project, another design needs to made. As such, the concept with using Docker containers, which in theory would be good to have working because of the scalability it provides, can't be used since getting it to work would require a lot of time for debugging and fixing the problems with the original project by Blacktop.

5.1.3 Finalised Third Setup

Since the option of running Cuckoo inside of a Docker container turned out to be harder than expected and unfeasible at this stage of the project, turning over to a more traditional setup is done. Since the AAU setup doesn't allow software to be run directly on the machine (machine is shared with other people), using a separate machine is done. This machine doesn't have restrictions and can be configured in any way one would desire. Instead, the machine allows one to do whatever they want with it. With the requirements of Cuckoo in mind such as running a Linux based environment, a spare machine was chosen that could support this as well as have enough power to support running malware samples. When doing malware analysis with Cuckoo, memory dumps from the VM's are used to see what happened during the execution of the malware samples. As such, enough CPU cores and RAM was needed to make sure that Cuckoo, VM and the malware analysis could run properly.
With that in mind, the system can be made as a traditional system with system components used that can be seen as follows on Table 5.1

Component	Info
Processor	Intel i7-3770k 8 core @ 3.40GHz
RAM	16 GB
OS	Ubuntu 18.04
Storage	500 GB
Programs	VirtualBox 6.1 Cuckoo 2.0.7

Table 5.1: Table showing what components was used for the malware analysis.

As seen on the Table 5.1, the system used is very basic and several years old in terms of processor power. However, the system is still very capable albeit a little slower than if a processor form 2019 was used such as the i9-9900k. With the physical system in place, the design within the system can be made. With what was learned from Section 5.1.1 and Section 5.1.2, the system is made with similar design principles but differ in that Cuckoo, which is run as root on the system, can see the VirtualBox installation and can modify it according to what kind of tests, VM and snapshots are used. The system can be seen as follows on Figure 5.8.

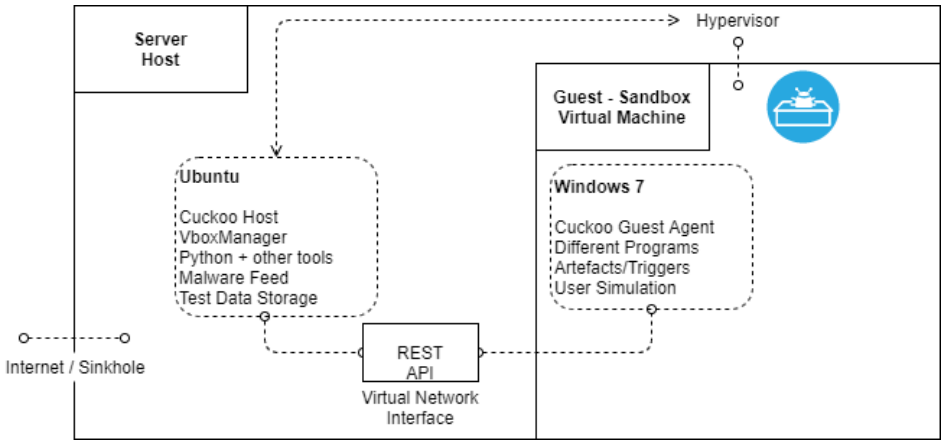


Figure 5.8: Architecture of third and final modified setup

As seen on the figure, the system still works by having two communication pipes from the Ubuntu OS to the VM. Here, Cuckoo uses the Hypervisor to control the VM and the virtual network interface to communicate directly within the Windows 7 VM directly through an agent process. The agent process, which is a python script, runs with the Windows 7 VM and allows for monitoring and user simulation during run-time which is then reported back to Cuckoo running outside the system by using some specific network settings (see Section 6.4).
With this system design in place, malware analysis via Cuckoo is possible and is what will be used for the different tests that will be done to determine if changing different artefacts within the VM will lead to a change in the malware sample behaviour.

5.2 Malware Sample Processing

With the system design set in place, the processing of malware samples can now be designed. By using Cuckoo's handbook [16], the flow of uploading a piece of malware, executing it within the VM and doing an analysis on the results can be done. The flow can be seen on Figure 5.9.

With Cuckoo, a malware sample can be uploaded directly through the command line or through a web User Interface (UI) where analysis results are also reported. From the web UI, the malware sample is uploaded to the Cuckoo Core and different options can be chosen in regards to what happens with the VM. When ready to start, the Cuckoo Core activates the VM and puts the malware sample locally in the Windows 7 VM. From there, the sample is executed and the Cuckoo Agent reports back to Cuckoo Core with what happens within the VM such as file and network changes. After the test is completed, which is defined by the malware sample activity, if it's still running, timed out or simply finished by shutting down, a memory dump of the entire VM is done and stored on the host machine. This memory dump contains the last state of the VM and will show everything that was changed from before to after the malware sample was executed. The analysis part is done by tools such as Volatility and MalConfScan which give a detailed report of what happened within the VM.

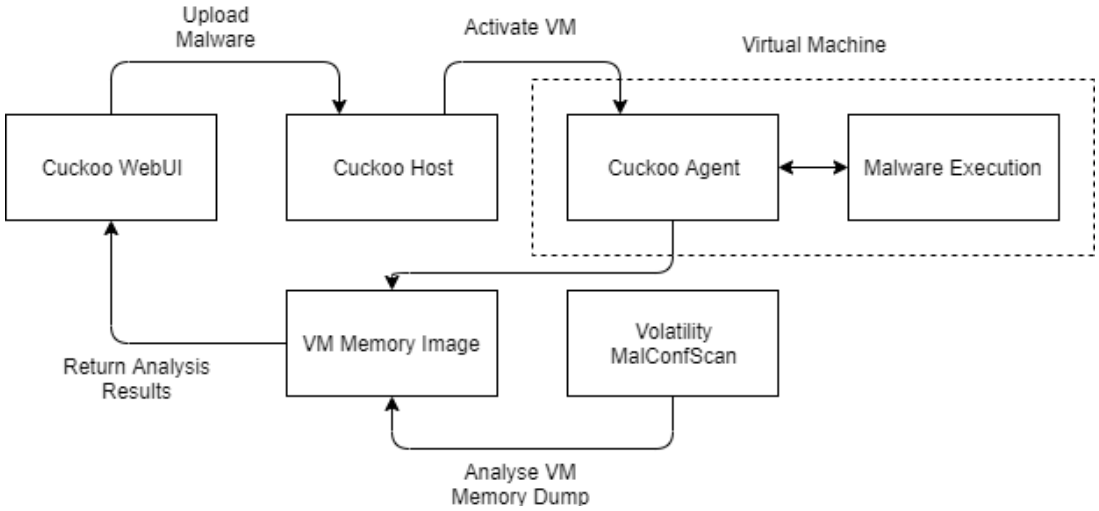


Figure 5.9: Simple Diagram showcasing the malware analysis process

When the malware sample analysis is done, a report is made which can be viewed within the web UI. From the report, noting down the different criteria for how malware behaviour is defined (see Section 4.4) and comparing it between the baseline test and the modified test can be done. Doing so can determine if the tested malware changed its behaviour between the different tests. Since running a single sample can take anywhere from 45 minutes to 1 hour the design also allows for malware samples to be queued which makes it possible to test a large batch of samples without interacting with the system. Cuckoo makes sure that the correct Windows 7 VM is used and after every test reverts to the proper snapshot as defined by its configuration.

Running a malware sample test outputs a log to the Cuckoo Core terminal which can help with knowing when the analysis is done as well as any problems that could have occurred during testing. The example output can be seen on Figure 5.10.

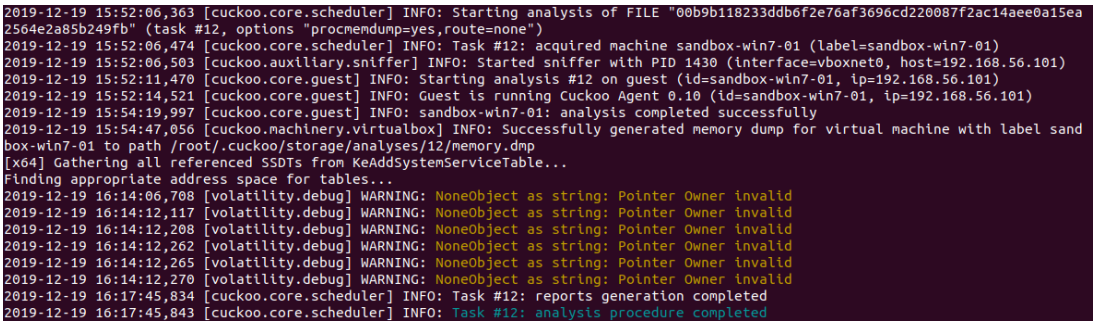


Figure 5.10: Example Output from Cuckoo During a Malware Sample Analysis

5.3 Virtual Machine Snapshots

In order to test malware and determine if removing and adding different artefacts has an effect on malware behaviour, having several data set to compare against each other is needed. In this case the optimal choice is to have several VM images to compare against each other. For this scenario it is decided to have two different snapshots, or images, where one image is used as a Base Image and another image is used as a Modified Image. For the Base Image the idea is to only change as much as needed within the VM in order to make testing functional. Changing anything else could have the result of removing or changing different artefacts within the system, which for a Base Image, is not wanted. As for the Modified Image, removing or changing artefacts is applied. For the Modified Image, trying to remove as many artefacts that reference the virtual environment, which in this case is VirtualBox, is done in order to look like a ”real” system. From a design perspective this is achieved by going through all known common artefacts within the VM and removing or changing them to remove their reference trace. As an example, the list of common artefacts includes, but are not limited to, the following list:

- Changing BIOS information.
- Changing Windows Serials.
- Changing Hardware serial, firmware, model numbers and vendors.
- Changing Medium Access Control (MAC) addresses and interfaces.
- Changing Advanced Configuration and Power Interface (ACPI) system description tables [38] (Differentiated System Description Table (DSDT), Fixed ACPI Description Table (FADT), Root System Description Table (RSDT) and System Service Descriptor Table (SSDT)).

Besides changing information within the system, creating different artefacts is also wanted. Creating artefacts has the chance of increases the number of malware which deliver their payloads. An example is a piece of malware that looks for a certain program such as Adobe Reader or for specific files such as Microsoft Office documents and image files. Without those specific programs or files, some malware might not detonate and will be reported by the analysis tool used as safe or dormant executable.
For all cases mentioned above, some things can be changed outside the VM but in many cases, the only way to change things is to go directly into the VM itself and change it while the system is running. In this case, every time the VM machine boots up, a script would need to run and change settings and registry entries in order to remove or add artefacts. This can simply be done by adding

a script to the system startup that executes with the system before malware is run on the system. Since the system works with using snapshots, after the script has been executed it can be deleted in order to remove any trace of it before malware is executed. The snapshot would then be reverted and the script could run again before the next malware test. For the cases of the two image snapshots that are to be used, the Base image will not feature a script to change anything while the Modified image will.
With that in mind, the two images, Base and Modified and their differences can be seen on Figure 5.11.

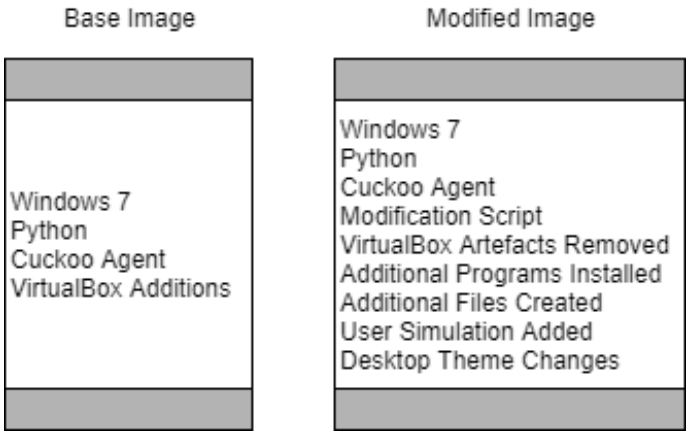


Figure 5.11: Differences between Base and Modified Image

As seen from Figure 5.11, the two images differ a lot but are also what is needed to see if changing a lot of things within a VM is enough to change malware behaviour that is executed within the VM. As such, the design of the images will carry over to the implementation that will be used for the testing and analysis of malware.

6 | Implementation

In this chapter the implementation of the system and all of the different system components will be described in detail. This includes everything from malware samples to malware execution within the system. From what was learned in the previous chapters, the system needs to fulfil different criteria in order to systematically test malware samples. The system needs to be able to support handling different VM images that are going to be used to see if malware changes behaviour when executed in different virtual environments. From the system requirements, key areas to work around were the scalability, reliability and complexity of the system which need to be taken into consideration when building the system. This included being able to expand the system if needed in order to test different kinds of implementations in terms of artefacts and triggers put into the system. In the system design chapter it was learned how using known software and tools such as Cuckoo and VirtualBox as well as changing their configuration would help with fulfilling the different requirements as well as support the different use cases that were laid out in terms of testing and comparing the results. This also includes how malware is tested within the system in terms of creating and running the two different VM images as well as how data is processed and compared in order to evaluate the artefacts implemented and the change in malware behaviour.
The system is intended to be used for malware testing both for this project but also for future work done at Aalborg University. This means that the implementation will go into detail of every major component, how it was developed and used within the system. For this going into detail in areas such as components that were set up using commands as well as including a description of different approaches that were taken to make setting up the system easier for future work. The reason why this area will go into depth with the different components is that AAU wants to potentially use the system in the future. Because of this making sure that the system is set up properly and is well documented is a high priority. The chapter contains the following components that will be described: *System Overview*, *Cuckoo Sandbox*, *VirtualBox*, *Networking and Data Transfers* and *VM Snapshots*.

6.1 System Overview

The system that is made is based on what was seen during the system specification and system design where the figures Figure 4.1 and Figure 5.8 were used for the system implementation.
As mentioned above the system consists of several components which make up the entire system where the following components that make up parts of the system are described as follows.

- **Cuckoo Sandbox**
  The Malware Analysis System. This component handles the malware samples by loading them into the VM and execute it for malware testing. Cuckoo also handles the processing and comparison of the results from running the malware sample tests

- **VirtualBox**
  VirtualBox is where the malware testing actually happens. As a standalone product, VirtualBox can't be used for automated malware testing and as such needs to be modified to be able to test malware with the Cuckoo Sandbox System.

- **Networking**
  Another important component is the networking aspect of the system. Since the system is connected to the internet, having your network settings allow the VM to access the network outside of the virtual environment can become a big problem if everything isn't set up properly. In a controlled environment, networking would be set up for the VM to access the internet but since the system to be implemented is a proof of concept running on a network with other important machines, that can't be allowed. A simulated network which could include using the Tor network could also have been a possibility.

- **VM Snapshots**
  The snapshots are what makes it possible to automate malware testing. The snapshots are what contains the current state of the VM that is used and is what the Cuckoo Sandbox will revert to before each malware test. The snapshots are therefore in a state where they are ready to be used for testing purposes.

In order to make the installation process as smooth as possible, everything was made in the form of installation shell scripts where each was carefully made to install properly. The different scripts can be found in Appendix C. The list of scripts that were made look and have the following order:

1. **installprereq.sh**
   Installation of all pre-requisites.

2. **installpostreq.sh**
   Installation of Cuckoo and its components.

3. **installvirtualbox.sh**
   Installation of VirtualBox.

4. **installantivmdetection.sh**
   Installation of Anti VM Detection scripts.

5. **installNetworkConfiguration.sh**
   Configuration of the Host Networking.

6.2 Cuckoo Sandbox

As mentioned earlier the Cuckoo Sandbox is the component responsible for running the malware tests as well as analysing and comparing the results of said tests. To get started with Cuckoo, their own guide as well as guides found on the internet were used for installing and running everything needed to get Cuckoo up and running. Since Cuckoo is open source, a lot of different ways to install and run Cuckoo can be found on the internet. It has therefore been decided to stay as close to the original guide made by Cuckoo themselves as possible.
As for Cuckoo itself, the application consists of two parts. The first is the Cuckoo Core and the

other one is the Cuckoo Web Interface.

In order to install Cuckoo a lot of dependencies are needed to be installed. These dependencies are simply to make Cuckoo function as well as make other applications that also need to run function properly.

```
sudo apt-get update && sudo apt-get upgrade -y && sudo apt-get autoremove -y
sudo apt-get install DEPENDENCIES

// Install Python, MongoDB, Postgresql, Tcpdump, M2Crypto, Volatility, Distrom and YARA
```

The above installed applications include a lot of different types of applications that each has something to do with Cuckoo. In short the applications handle the following areas:

- Database management (MongoDB, Postgresql)
- Network Packet Analyser (tcpdump)
- Cryptography support and security for Python (M2Crypto)
- Forensic analysis on memory dumps (Volatility)
- Binary stream instruction decomposer (Distorm)
- Malware Identification and classifications (YARA)

After the aforementioned applications have been installed, we can progress to install the Cuckoo Sandbox. To do so, we go into superuser mode and install it in a `virtualenv` which isolates Cuckoo in its own environment:

```
virtualenv venv
sudo su
. venv/bin/activate
sudo pip install -U pip setuptools
sudo pip install -U cuckoo
```

After Cuckoo has been installed, running the following command can be started to check if everything is in order:

```
cuckoo -d
```

As for the configuration of the Cuckoo Sandbox, some things that we want to do such as analyse memory dumps as well as specify our VM, need to changed. This is achieved by going through Cuckoo's configuration files and changing thing accordingly. This is done as follows:

```
sudo nano /root/.cuckoo/conf/cuckoo.conf
    memory_dump = yes

sudo nano /root/.cuckoo/conf/virtualbox.conf
    mode = gui
    machines = sandbox-win7-01
    [sandbox-win7-01]
    label = sandbox-win7-01
    snapshot = clean-final-X
    tags = windows_7, 64_bit, python

sudo nano /root/.cuckoo/conf/memory.conf
    guest_profile = Win7SP1x64
    delete_memdump = yes

sudo nano /root/.cuckoo/conf/processing.conf
[memory]
    enabled yes

sudo nano /root/.cuckoo/conf/reporting.conf
[singlefile]
    enabled = yes
    html = yes
    pdf = no
```

Now that everything has been installed and configured, Cuckoo can now be started together with its web interface in order to start testing malware samples. However, we also need to install VirtualBox which is the next step.

6.3 VirtualBox

Installing VirtualBox is one of the easier parts of the system since everything needed to make it run comes in one package. To do so the official repository for VirtualBox is added to the machine and then the newest version of VirtualBox is downloaded and installed.

```
echo deb [arch=amd64] https://download.virtualbox.org/virtualbox/debian bionic contrib |
  ↪ sudo tee -a /etc/apt/sources.list.d/virtualbox.list

wget -q https://www.virtualbox.org/download/oracle_vbox_2016.asc -O- | sudo apt-key add -
wget -q https://www.virtualbox.org/download/oracle_vbox.asc -O- | sudo apt-key add -
sudo apt-get update
sudo apt-get install virtualbox-6.1
```

6.3.1 Windows OS

In order to test malware, a copy of windows is needed. In this case and as recommended by Cuckoo Sandbox, Windows 7 is installed. This is simply done by taking the .iso file for the copy of Windows 7 and using it as a booting device within VirtualBox. When the installation is complete, the VM can be configured. This is done in two ways. Since two separate images are required, a Base and a Modified one, there are also two VM's to work with which each have their own configuration, the

default one and the recommended one. The difference between the default and the recommended configuration is the hardware supplied to the VM. For the default, the following settings are applied:

```
General:
    Name:                   sanbox-win7-01 (base)
    Operating System:       Windows 7 Ultimate (64-bit)
System:
    Cores:                  1
    Base Memory:            2048MB
Display:
    Video Memory:           27MB
Storage:
    Normal 40GB
```

And for the recommended one the settings are slightly changed going from 1 CPU core to 2 and from 2048MB of RAM to 4096MB.

```
General:
    Name:                   sanbox-win7-02 (modfified)
    Operating System:       Windows 7 Ultimate (64-bit)
System:
    Cores:                  2
    Base Memory:            4096MB
Display:
    Video Memory:           150MB
Storage:
    Normal 40GB
```

Now that both images have been set up in relation to their VirtualBox settings, the networking between the Guest VM's and the Host can be done.

6.4 Networking and Data Transfers

The networking aspect of the system consists of several areas. One area is the host system and the other area is inside the guest VM. Both have been set up to allow traffic and data to be exchanged between the, while keeping the Guest away from the real internet.

6.4.1 Host Networking

On the host, specific settings need to be made in order for Cuckoo and VirtualBox to be able to interact with each other over the network.

First, the networking tools are installed:

```
sudo apt install net-tools
```

Then, the network is configured for the by utilising the VBoxManager which is the tool that controls the VirtualBox application. Here a new network interface is created which is bound to VirtualBox. Using VBoxManager the configuration is done automatically and only needs a couple of commands:

```
sudo vboxmanage hostonlyif create
sudo vboxmanage hostonlyif ipconfig vboxnet0 --ip 192.168.56.1
sudo vboxmanage modifyvm sandbox-win7-01 --hostonlyadapter1 vboxnet0
sudo vboxmanage modifyvm sandbox-win7-01 --nic1 hostonly
```

In the above commands, a new interface is made called `vboxnet0` and is bound to the IP `192.168.56.1`. After that, the VM called `sandbox-win7-01` is bound to that interface.

When the interface is up and running, the routing and traffic restrictions can be applied. As mentioned earlier, we want traffic to go from the Host to the Guest and back, but we don't want the Guest to be able to access the outgoing internet from the Host. To do so, the default settings are to drop all forward requests and only allow internal traffic. The settings are configured with the tool called IPtables:

```
sudo iptables -t nat -A POSTROUTING -o eth0 -s 192.168.56.0/24 -j MASQUERADE

//Default drop.
sudo iptables -P FORWARD DROP

//Existing connections.
sudo iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT

//Disallow connections from vboxnet to the whole internet.
sudo iptables -A FORWARD -s 192.168.56.0/24 -j DROP

//Internal traffic.
sudo iptables -A FORWARD -s 192.168.56.0/24 -d 192.168.56.0/24 -j ACCEPT
```

The rules applied here won't be able to do any form of packet forwarding unless IP forwarding is enabled within the kernel on the system. To do so, there is a temporary method that resets upon reboot as well as a permanent method which saves the settings even after reboots.
For the temporary solution:

```
echo 1 | sudo tee -a /proc/sys/net/ipv4/ip_forward
sudo sysctl -w net.ipv4.ip_forward=1
```

And for the permanent solution:

```
sudo apt-get install iptables-persistent
```

After everything above has been done, the networking for the Host machine has been finished.

6.4.2 Guest Networking

After the Guest has been set up with a copy of Windows 7, the network settings can now be changed in the VM. To do so, the local interface is configured to communicate with the Host machine by routing all traffic from the VM to the Host machine. This is done as follows:

```
IP Address: 192.168.56.101
Subnet Mask: 255.255.255.0
Default Gateway: 192.168.56.1
Primary DNS: 1.1.1.1
Secondary DNS: 1.0.0.1
```

With the above settings applied, the quest can interact with any network application running on the `192.168.56.1` IP.

With the networking being applied data can now flow between the Host and the Guest without any issue.

6.4.3 Data Transfers

In order to transfer data to the VM without using VirtualBox, since that introduces artefacts, a FTP server is created on the host which the Guest can access and take data from. This is needed since we want to transfer the Cuckoo agent to the VM as well as other files such as programs and scripts to set up the Base and Modified images. To do so, an application called `vsftp` is installed:

```
sudo apt-get install vsftpd
```

After the application has been installed, it is configured to be accesed by the local machine from the Guest:

```
sudo mkdir -p /home/tarik/vmshared/pub
sudo chown -R tarik:tarik /home/tarik/vmshared
sudo chmod -R ug=rwX,o=rX /home/tarik/vmshared/
sudo chmod -R ugo=rwX /home/tarik/vmshared/pub

sudo nano /etc/vsftpd.conf
    write_enable=YES
    anon_upload_enable=YES
    anon_mkdir_write_enable=YES
    listen_address=192.168.100.1
    listen_port=2121
    anon_root=/home/tarik/vmshared
    anon_umask=000
    chown_upload_mode=0666
    pasv_enable=Yes
    pasv_min_port=10090
    pasv_max_port=10100
```

The above settings make it possible to access the newly created folder called `vmshared`. In this folder we will place the files that we want to be accessed from the Guest VM. After the files have been transferred, the FTP server can be shut down just in case no malware sample finds and accesses it.

6.5 VM Snapshots

As mentioned during Section 5.3, two different VM images are used for the malware testing and analysis. The two images differ in that one image is used as a clean base i.e. Only the essentials were installed to make it work malware testing. The other is used as a modified base where extra programs, updates and anti virtual machine tactics were used to make the modified image look more like a real system instead of a blatant virtual machine.
In this section, an overview of what and how these things were done to the two images will be explained as well as a quick test through a test script with Pafish. For both images, files were transferred to the guest VM's by using FTP.

6.5.1 Base Image

For the base image that was used for the initial testing and analysis of malware, the only changes that were made were done in order to make the Cuckoo Agent function as well as allow malware to execute. The agent comes in the form of a python script which creates a basic web service that can communicate with Cuckoo outside of the VM. The changes that were applied to the base image are as follows:

- **VC-redists**
  Visual C++ Libraries that allows many different programs to run properly. This is needed to test a variety of malware.
- **Python**
  Allows the agent to run as well as collect data from the guest virtual machine.
- **Pillow (Python Imaging Library)**
  Adds image processing capabilities to python. This is used to take screenshots of the desktop.
- **Cuckoo Agent**
  Allows the VM to send data back to the Cuckoo host for analysis.

With the above list installed on the guest VM, the base image is now ready to be used for malware analysis. It should also be note that the Cuckoo Agent was also set to run on startup in order to automate the boot process of the guest virtual machine.

6.5.2 Modified Image

For the modified image that was used for the second round of malware testing and analysis, the base image that was initially made was cloned. This was done in order to have the same foundation for both images and would avoid creating further artefacts during testing by having different configurations on the two images. The cloned image was then further edited by installing different software as well as running an anti VM detection script which changes files, registry entries and settings on the guest VM in order to make harder to detect as a virtual machine. The most important part here is the registry entries and settings part. This includes files and references to VirtualBox and anything related to the system being a virtual machine.
The changes that were applied to the modified image are as follows:

1. **Removed VirtualBox Guest Additions**
   Consist of device drivers and system applications that optimise the guest operating system for better performance and usability. These are by default a part of the OS when installed via VirtualBox.

2. **Added Programs and Updates**
   Adobe Reader (XI 11.0.01), Adobe Flash player 11.9, Java 7, Windows Update KB958830 (Remote Server Administration Tools), LibreOffice, VLC Media Player, Firefox, Google Chrome, 7Zip, paint.net.

3. **Anti VM Detection part 1**
   Changes settings, registry entries and add randomisation in the form of files with common extensions.

4. **Anti VM Detection part 2**
   On every reboot, registry entries and settings that are reset are changed again.

From the list above there are a lot of changes that impact the system being worked with which in this case is the modified image. The idea is that these changes will have an effect on how the different malware samples behave by eliminating common artefacts that are introduced by using virtual machines.
From Section 5.3 and Section 3.1 different types of artefacts and triggers were discussed and how they are easily found since they are always located in the same place. In order to change artefacts within the modified image some have been identified and either removed or edited. For the configuration used for testing in this project, the most common artefacts are identified which are those that are added by default into the system. The identified artefacts include the ones from the above list such as the VirtualBox Guest Additions which is added by VirtualBox itself where some of them can be seen on Table 3.3. Since this is the case, it is also one of the easiest to detect because they are a default setting that can be applied to a VM image. Going further than that, the anti VM detection parts also work with these artefacts by removing the artefacts that are outside of the Guest Additions and are located as default within the VM image (as discussed in Chapter 3 and Section 5.3). These include artefacts found by Pafish (Section 3.2.1) as well as what was described in places such as [19]. In terms of the second bullet point which includes the different applications that were installed, these include a combination of older applications as well as other common applications that typically installed by many machines which lie in the category of ”must have” applications for most systems [35] [17]. The idea here is that for some malware, their detection schemes include looking for different common applications such as browsers and PDF readers in order to determine if a system is in use by a real person. This also includes artefacts left by said applications such as bookmarks for browsers and PDF files for the PDF reader recent list. As far as what applications are more important than others, if entirely depends on the malware being tested which means that knowing what applications to install varies between malware samples.

To ensure that everything is installed properly, the modification process of the image was done in several steps.
First, the guest additions added by VirtualBox are removed. The guest additions add a lot of artefacts in terms of references to VirtualBox. Simply removing the installation helps with minimising artefacts but does lower performance inside the VM.

Second, basic programs and updates were installed. The programs installed, as explained in Section 5.3, are used as a trigger artefact for malware since a large variety of malware looks for or uses said applications. The chosen applications were taken based on several sources of recommended or essential applications that are installed on new user machines as mentioned earlier.
After the basic part was modified, the more advanced and crucial part was modified which is the OS aspect of the guest VM. As mentioned earlier in the report, virtual machines and their guest OS come with a lot of files, registry entries and references that refer to the virtual machine that is used. In this case VirtualBox is referenced a lot either as VirtualBox or as VBox. To change or remove those references, an open source script on GitHub called "antivmdetection" was used with slight modifications in order to work with the VM images used [40]. The script goes through everything known for a virtual machine that could be referenced within VirtualBox and edits or removes them. As an example, the script goes through the following to either remove, create or edit files, settings or registry settings based on either current or randomised values:

```
# Check dependencies
# Randomize BIOS serial
# Check other present hardware
# Create a new UUID
# Create a new system serial number
# Create a new chassi serial number
# Change disk serial
# Change disk firmware rev
# Change disk model number
# Change CD-ROM firmware number
# Change CD-ROM model number
# Change CD-ROM vendor name
# Get and write DSDT image to file
# Randomize VM MAC address, based on the host interface MAC
# Copy and set the CPU brand string
# Check the numbers of CPUs, should be 2 or more
# Check the set VM memory size. If it's less them 2GB notify user
# Check if hostonlyifs IP address is the default
# Check witch paravirtualization interface is being used
# Set custom Differentiated System Description Table (DSDT) information
# Set custom Fixed ACPI Description Table (FADT) values
# Set custom Root System Description Table (RSDT) data
# Set custom Supplemental Descriptive Data Table (SDDT) information
# Set operating system InstallDate and MachineGuid
# Depending on the operating system, set DACType and video card type
# Set the Microsoft Product ID (ProductId) and desktop background
# Check if audio support is enabled
# Check if you have the correct DevManview binary for the target architecture
# Write all data to file
# Generate random files with the following extensions: .txt, .pdf, .txt, .docx, .doc, .xls,
  ↪  .xlsx, .zip, .png, .jpg, .jpeg, .gif, .bmp, .html, .htm, .ppt, .pptx.
```

Figure 6.16: Anti VM Detection Script excerpt

The output from the script comes in the form of a PowerShell script which is then executed within the guest VM in order to apply the changes that are to be made. How the PowerShell script is made can be seen on Figure 6.17.

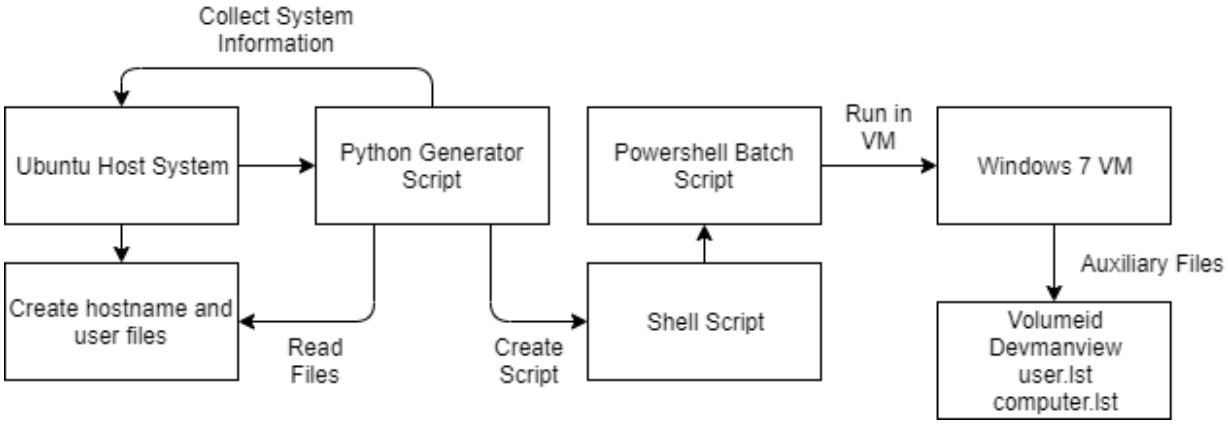
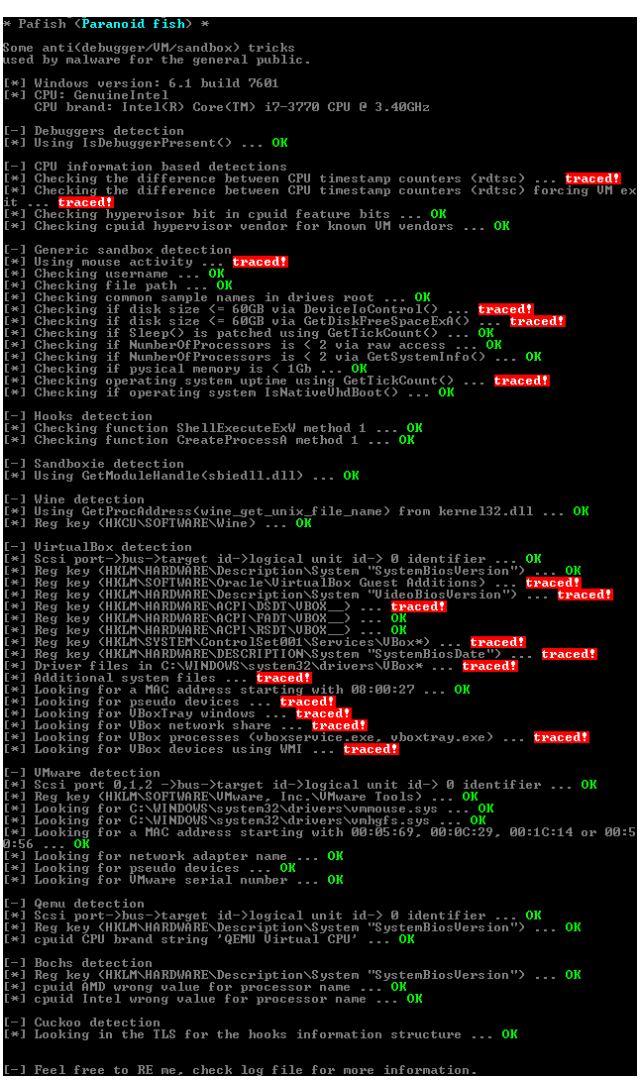


Figure 6.17: Flow of how the Anti VM Detection script is generated and run.

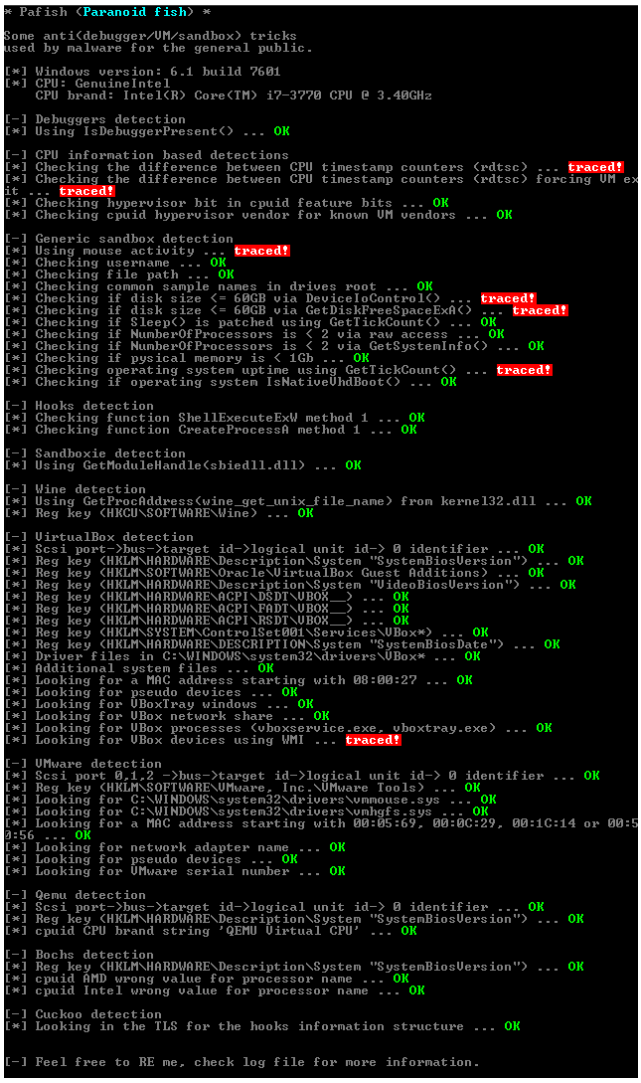
In the above figure it can be seen how the PowerShell script is made by first running a Python scripts that is collecting information from the Host system which in this case is a Ubuntu based machine. The reason it does this is because of the VirtualBox installation. The script needs to know its configuration before it can change it. When all the information is collected another script is made in the form of a Shell script. The Shell script then generates the PowerShell script which is what will actually change things inside the VM. This combined with some auxiliary files will remove and modify different artefacts within the Windows 7 VM. The auxiliary files help the script remove certain settings and devices that are made by VirtualBox and with the help of the .lst files, the script can generate random files on the system.

6.5.3 Difference Test Using Pafish

Running the two different images should in theory give two different output from Pafish since the images used have been altered in relation to each other. The base image hasn't been altered with and thus includes a lot of artefacts that reference VirtualBox. The modified image has been altered by removing as much as possible that reference VirtualBox.
As such, running Pafish on both images gives the following output on Figure 6.18a and Figure 6.18b:



(a) Pafish output from the base image.



(b) Pafish output from the modified image

Figure 6.18: Pafish output from both images used which show the differences between them.

As it can be seen from Figure 6.18, the two Pafish outputs differ in several places. The artefacts for the things that are related to VirtualBox have gone from being ”traced” to ”OK”. This means that the artefacts have been removed from the system and can’t be found anymore.
For testing malware, the malware should be tricked into thinking that it is inside a real system since it can’t find anything related to VirtualBox or other virtualisation software. It should be noted that there are still some artefacts that are present as it can be seen from the two images. Those artefacts are mainly related to the system itself which can’t be removed. One of the artefacts is the mouse movement. For the Pafish tests, no mouse movement was simulated but for the malware testing, it should be noted that mouse movement and interaction is simulated by Cuckoo. Two other artefacts can also be removed which are the detected disk size of the systems. For the Pafish testing the images were set to 40 GB each. The size of the disk can vary a lot depending on what malware considers a ”real” size. As such, having a definite answer to how big the disk size should be can’t be answered unless a very large sample size is tested where it is known what disk size is expected, if it is even being detected.

6.6 Running Everything

When everything is installed and ready to go, launching the different components is done as follows:

VirtualBox:

Starting VirtualBox is simple. The only thing that needs to be done is running the following command:

```
sudo virtualbox
```

Cuckoo Sandbox:

Since Cuckoo is installed as root, the way to launch Cuckoo is going into root mode. Besides that, after a reboot, VirtualBox also needs to attach itself to the correct IP address which will be used for communication between Cuckoo and VirtualBox. To do so the application VBoxManager is used. The following commands are used:

```
sudo su
VBoxManage hostonlyif ipconfig vboxnet0 --ip 192.168.56.1 --netmask 255.255.255.0
cuckoo
```

Cuckoo Web Interface:

To start the web interface that is used to start malware sample tests, the cuckoo web service needs to be started together with mongodb in order to save things to a database. As with normally starting Cuckoo, root must be used. The following commands are used to get the web interface up and running:

```
sudo su
cd /root/.cuckoo
sudo service mongodb start
cuckoo web runserver 0.0.0.0:8080
```

When everything is up and running, testing malware samples can begin.

7 | System Testing and Malware Analysis

In this chapter the different tests as well as examples of malware analysis will be shown. The idea with the system created was to see if changing different parameters in the form of artefacts within a closed environment would lead to a change in malware behaviour. Because of this, a system was described and implemented in Chapter 4, Chapter 5 and Chapter 6 which can be used to test and measure for malware behaviour changes. The purpose of the tests is to see if the aforementioned artefacts that were changed within the described system would change how malware behaves by using two different closed environments. The behaviour change will then be evaluated by looking at different events (file/process creation, registry edits etc.) that happened during testing and if the amount of events differed between the two VM images used. If the results from running a malware sample differ between the two VM images, a change in behaviour could possibly observed.

Before the system could be tested with live malware, the system needs to be validated in terms of running as it should as well as performance. With virtual machines, performance can vary a lot depending on the host systems hardware. As such each of the two VM images made are tested before anything else is done to them.
Besides the validation, compatibility of the malware samples is required to be checked since the system that is implemented only supports Windows based executable. These executable are also limited to what the system allows to run since not every Windows update to ever exist would be installed on the VM images. The reason for this is to have a system that isn't running on the newest version of Windows which could have patches made and implemented that would prevent some form of malware to even run. This scenario was observed in the preliminary testing where a sample that was too old, dated back to 2002, simply couldn't run because by the time Windows 7 was released, patches to stop the malware sample were already integrated into the system.
Another area in terms of system testing and performance is the how the system is presented in a realistic manner like a real world system with real world usage. Since virtual machines are limited in of themselves, making sure that the system mimics a real world system with real world scenarios would be a top priority. The reason for this is that some malware tends to look at user activity before they execute. As mentioned earlier in the report the user activity can come in many forms such as mouse movements but also in terms of what is happening in the active session. This comes in the form of applications installed and applications that are running in the active session. Though not every type of application can be installed and run, making sure the sessions cover "enough" can be done.
For this reason, the two VM images used showcase how one system, the Base image, omits having any real world sessions while the other image, the Modified one, includes a lot of different things such as randomised files, applications and user simulation.

7.1 Testing Method

In this section, the testing method used for testing and analysing live malware samples within VM's. The method for testing that was used was quite simple, since everything for the system was setup to be automatic, the only thing a user needs to add the chosen malware sample to the Cuckoo Web Interface. After that, the user can configure some basic settings in regards to the test and what VM to choose from. The rest is then handed over to Cuckoo to run the malware sample and collect all the different data required to do an analysis afterwards. In a sense the method used is black box testing since the user doesn't know what happens while the test is running. Some input is given, the malware sample, and a result is given back afterwards that you can read through.

7.1.1 Testing Criteria

As mentioned previously, there are certain things to look out for when comparing the different malware sample tests after results have been gathered. For the basic ones and what will be shown in the results is the Cuckoo ranking system. After each analysis, based on what Cuckoo and its analysis found such as anything resembling malicious behaviour, a malware sample is classified by a ranking. This ranking goes from 1 and up where 1 to 10 is the colour brackets. The ranking is based on the many different malicious things such as creating files, editing the registry and changing windows settings such as the firewall. If the malware sample does something within each category the score goes up. The scoring system that is applied by Cuckoo is however a little misleading because of the underlying methodology that it uses as per [52]. Even if a malware sample gets a low score does not mean that it is not malicious since the degree can vary and cannot always be rated in an objective way. Because of this, the ranking should be looked at as a guide and a more in depth analysis of the results are needed.
As such, the ranking is put into three different categories of danger:

- Rank: 1-4 (Green) -> Numerous Signs of Malicious Behaviour
- Rank: 4-7 (Yellow) -> Many Signs of Malicious Behaviour
- Rank: 7-10 and up (Red) -> Very Suspicious Malware Behaviour

Looking at the ranking can give a quick overview of how malicious a malware sample is as well as indicating if a malware sample changed behaviour from one test to another by having a different rank between the tests. It is however not necessary, for this project at least, to look for specific scores such as the ones that are high. What is important to look after is the change in score. So if one malware sample goes from a score of 4 to a score of 8, then it would be taken as a set of results to investigate what happened that could have caused the change in score. The ranking score can, as mentioned earlier, be misleading and could produce the same score even though something differently happened between tests. In order to trust the ranking system, a replacement as provided by [52] would be in order, otherwise a more manual approach in comparison is needed for the different tests.
Another thing for what is considered a change in malware behaviour and what is looked after when investigating the results is the proportions and distribution of events that happened during testing. As an example, an event could be a single instance of many things such as a file creation or a change in the system registry. Basically any system interaction is counted as an event and if that event is classified as malicious, it will be added to the total distribution as well as the total signatures that were detected.

7.2 Malware Sample Test Results and Comparison

With all the test being done, a total of 21 different malware samples were tested. In Appendix D, information about the different malware samples can be found. Even with this small samples size, testing all of them twice, one for each VM image, took a long time (45-60 minutes). Because of this, testing more malware sample would have been preferred but was not possible due to time constraints. Because of this limitation it can be seen that different malware samples did change their behaviour from testing in one image to another. For the results the different malware samples, their type, date and ranking for both tests are displayed. The results are as follows:

Sample No.	Malware Type	Malware Date	Cuckoo Rank Base	Cuckoo Rank Modified	Change ?	Amount
1	Trojan-Generic	2012-06-24	4.6	4.6	No	0
2	Trojan-Adware-Ransom	2013-10-26	**10.4**	**11**	**Yes**	**0.6**
3	Virus-Botnet	2004-08-04	4	4	No	0
4	TrojanWorm-Prepender-Shodi	2004-09-12	**5.6**	**6.**	**Yes**	**0.4**
5	Trojan-Dloader	2013-09-18	6.6	6.6	No	0
6	Trojan-GameModding	2018-06-19	**4.8**	**6.4**	**Yes**	**1.6**
7	Dropped-Backdoor-Prorat	2009-07-14	**11.4**	**11.8**	**Yes**	**0.4**
8	Worm-Generic-Porn	2006-03-02	**8.2**	**10.2**	**Yes**	**2**
9	Virut-Generic	2002-08-12	**5.2**	**6**	**Yes**	**0.8**
10	FileRepMalware-Worm	2007-08-30	3.8	3.8	No	0
11	Trojan-Generic	2019-04-24	4.4	4.4	No	0
12	Trojan-Backdoor-Wabot	1992-06-20	**4.8**	**5.4**	**Yes**	**0.6**
13	Trojan-Ransom-Gandcrab	2018-05-01	10	10	No	0
14	Trojan-MSIL-Bladabindi	2019-05-09	**5.2**	**5.8**	**Yes**	**0.6**
15	Trojan-Generic	2019-05-07	7.2	7.2	No	0
16	Trojan-Shipup	2013-05-02	5.2	5.2	No	0
17	Virus-Virut-Generic	2001-08-17	**3.8**	**4.2**	**Yes**	**0.4**
18	Trojan-Downloader-Injector	2010-08-30	6.4	6.4	No	0
19	Trojan-Downlaoder-Waski	2013-11-25	7.2	7.2	No	0
20	Trojan-Dropper-Generic	2013-04-19	5.2	5.2	No	0
21	Trojan-FileRepMalware	2014-02-27	5	5	No	0

Table 7.1: The results of testing the different malware samples on both VM images.

From Table 7.1 it can be seen that out of the 21 different malware samples, 9 malware samples were observed to have a different ranking and thus some change in behaviour. However, to be sure that the ranking isn't just an anomaly since some rankings are close together, comparing some of the malware samples is in order. As mentioned earlier in the use cases in Chapter 4, the two different tests need to be compared in a way that can quantify the results which can be done by measuring events that occurred during testing. To do so, one malware sample and its two different test results are picked and compared using the Cuckoo comparison tool. From the comparison, different observations in terms of system interaction are shown. These interactions are measured as events that were detected during testing by tracing the process tree of the malware sample. The event categories, which are made by Cuckoo, are as follows:

- **Registry**
  If the registry has been modified.
- **File**
  If any file was created or modified.

- **System**
  If the system settings have been altered.
- **Network**
  If the network has been used, both for links and downloads.
- **Process**
  How much processes were interacted with, both read/write but also injections.
- **Services**
  If any system services were interacted with.
- **UI**
  If any UI elements were interacted with.

For the comparison, an execution graph is made. The graph gives an abstracted overview of the execution of the malware sample. The graph represents the percentage of occurrences of events classified by each category. The bigger the block or percentage, the higher the count of events for that category out of all events that were counted which were performed by the analysed malware sample.
The different categories are labelled and coloured to make it easier to see what part of the system was inflicted the most by the malware and its behaviour. It should be noted that the comparison only shows the distribution out of 100 percent. As such, one percentage might be bigger in one test than another even though the numerical number for the category didn't change.
From Table 7.1, malware sample no. 6 was used since it differed a lot from one test to another in terms of ranking. The comparison between the two tests can be seen on Figure 7.1.

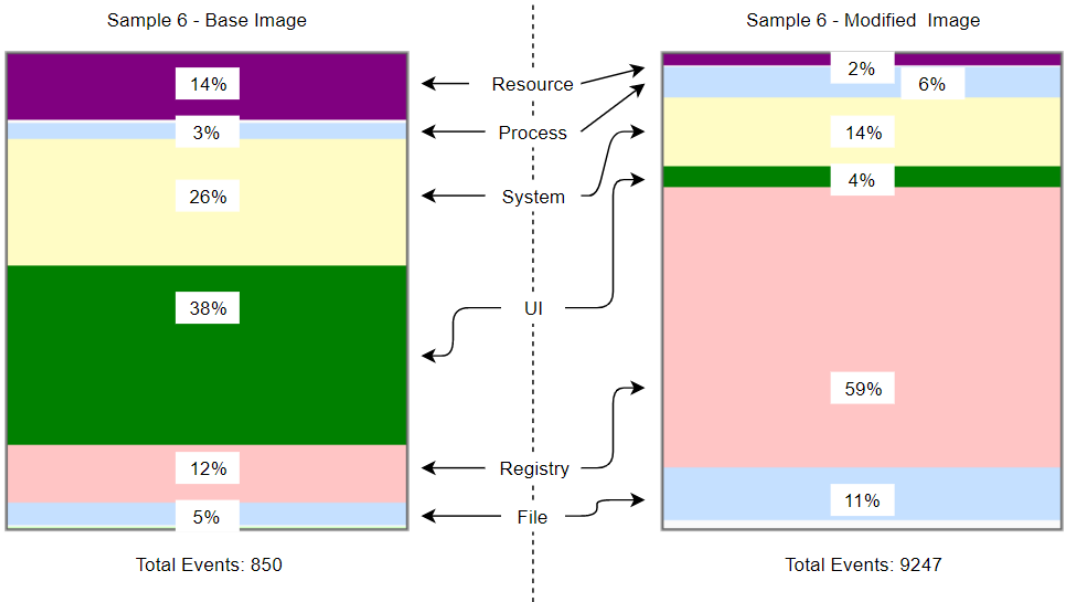


Figure 7.1: Malware sample 6 execution graph comparison

As seen on Figure 7.1, there are some difference between the two executions graphs. What is important to note from the two graphs is the "registry" and "file" category which went from 12 to 59 percent and 5 to 11 percent respectively. From the graphs it can be seen that they have grown in number of events where the "registry" category increased the most going from the base image to the modified image. From what is known about the categories it is therefore believed that the malware

sample decided to make further changes to the registry and file system based on the changes that happened between the base image and the modified images. Looking at the events that happened through the Cuckoo report, the following table shows a breakdown of the different categories:

Category	Base Image	Modified Image
Resource	121	226
Process	30	583
System	227	1381
UI	325	418
Registry	106	5586
File	41	1053
Total	850	9247

Table 7.2: Malware sample 6 events comparison for each category for both images

As it can be seen from Table 7.2, the amount of events increased a lot from the Base Image test to the Modified Image test where the Base Image had 850 events in total which was increased to 9247 events in total for the Modified Image. The number of events further proves that the malware behaved differently by doing a lot more during the test when compared to the Base Image test.
As for the different signatures that were detected during the testing, different results were produced by each test. The signatures cover everything from the malware sample unpacking itself, injecting into processes, creating files and stopping different services. For the tests with malware sample 6, the signatures that were detected for both tests can be shown on Figure 7.2.

Figure 7.2: Malware sample 6 signature comparison

As it can be seen from Figure 7.2, when testing with the modified image, more signatures are detected as well as the amount of previous detected signatures. From the test new signatures came up that indicate that the malware did several things more than when it was run on the base image test. From the figure the following signatures were detected during the modified image tests:

- One or more processes crashed (2 events)
- Searches running processes (6 events)
- Checks adapter addresses used to detect virtual network interfaces (1 event)
- Sets or modifies WPAD proxy autoconfiguration file for traffic interception (14 events)

As it can be seen from the list, what can be seen as the common denominator is that the events that happened during the modified test which didn't happen during the base test are related to potential sandbox evasion. The malware sample searched for processes which include the following processes:

- process_name: **pythonw.exe**
  process_identifier: 2524
- process_name: **taskhost.exe**
  process_identifier: 2260
- process_name: **mobsync.exe**
  process_identifier: 1736
- process_name: **SearchProtocolHost.exe**
  process_identifier: 2196
- process_name: **SearchFilterHost.exe**
  process_identifier: 1112
- process_name: **6_Mod_00aa31683f9292a003aed.exe (malware sample)**
  process_identifier: 2764

From the list above there are several processes that stand out. The first and probably the most important is the process called `pythonw.exe`. This process is a python process and is actually the process that is tied to the Cuckoo agent. If the malware was sophisticated enough, it could have tried to inject some malicious code into the process which could end up crashing the process or send malicious code back to the Cuckoo Host.

From what was found in the signatures it can be guessed that the malware went through several other sandbox evasion techniques, but because of the removed artefacts in the modified image, it continued with its payload which included looking up different processes, adapter addresses as well as changing several Web Proxy Auto-Discovery (WPAD) proxy configurations. From what is understood about WPAD, the malware most likely tried to spoof the WPAD in order to use a non-authorised server as a proxy server. This could be used to get requests from malicious domains in order to download more malicious files.

As for the other samples that were seen to have a change in behaviour, the results are similar to what was shown with minor differences in what aspect of the system were interacted with. Some malware samples such as downloader types tried to download several files which increased the network related events and some malware that created a lot of different files which were a copy of themselves or simply infected copies of other known files on the system.

7.3 Test and Analysis Conclusion

In this chapter a look into how testing was done for the malware sample testing and what thoughts lied behind it was done. Furthermore, testing and an analysis of the actual malware samples was done using the developed system which consisted of Cuckoo Sandbox and two VirtualBox VM images. This was done by collecting all the data from each of the two separate test categories and compiling them into different tables with the major outlier being described more in-depth.
As seen from the malware sample results, a lot of the samples that were randomly chosen to be used, didn't show any sign of changing its behaviour when going from one VM image to the other VM image. As discussed during the project, the reason for why a lot of the different malware samples didn't change behaviour lies in the fact that a lot of malware samples do not have evasion techniques implemented. Most malware samples simply deliver their payload and don't care if they get detected. One of the reasons for why this behaviour is common is simply because the malware sample was designed quickly and to attack as many systems as possible. If the sample is detected, the developer simply creates a new version and sends it out. This cycle repeats and is why there will always be a cat and mouse game between malware developers and anti-malware developers.
However, looking at the different tables it was seen how some malware samples did in fact change their behaviour. The behaviour observed mostly consisted of generating more malicious files, system calls and links. Further, the age of the malware samples didn't seem to matter since samples that were many years old and new ones both were observed to have a change in behaviour.
It should be noted that because of the nature of the testing where a malware sample was only tested ones per VM image, no definite conclusion can be made since more of the behaviour change could in fact simply be an anomaly with the malware samples and could be reverted back if they were tested again. More tests would be needed in order to confirm a definite change in malware sample behaviour. It is however shown that the system created during the project and the testing methodology can indeed be used for testing behaviour for different malware samples as shown by the results.

8 | Conclusion

During this project the task was to find out if creating a test setup that could be used to alter a closed environment could lead to a change in malware behaviour when testing malware samples using virtual machines. The different types of virtual machines should have altered environments where one would resemble a basic virtual machine while the other machine would have any reference of said virtual machines stripped among other changes in order to make the machine resemble a real system more. The solution designed and implemented was made at Aalborg University with a dedicated machine that was running on a isolated network in order to avoid infecting other network attached machines.

The system that was made to test malware samples was developed in different stages. The first stage was to find out what would be needed to test malware in terms of tools and software. In order to avoid inventing the wheel again, the choice was set on using established, open source tools, that could handle testing different kinds of malware samples in different virtual environments. To understand if malware samples changed behaviour between different environments, the system should support easily going from one environment to another in order to test malware samples quickly without taking an environment down in order to test in another. The solution landed on using the well established malware analysis tool called Cuckoo Sandbox together with the virtual machine manager VirtualBox. Both components supported each other which meant that little setup was required to make both work together. Using VirtualBox also allowed for switching between different virtual machines quickly.
The second stage was finding out what kind of system that would be used. At the beginning the choice was set on a powerful server, but because of the nature of Cuckoo Sandbox and VirtualBox and their need to have root access to the host machine, the server could not be used since root access was unavailable. The idea then was to take all the root requirements and put it inside a Docker Container. However, since Cuckoo Sandbox wasn't supported by Docker, that option quickly fell apart. In the end a dedicated machine was used where everything was made from the ground up to support Cuckoo Sandbox and VirtualBox.

In addition to setting up a dedicated machine for malware testing, it was also needed to implement artefacts and triggers in order to change the behaviour of the malware samples. This was achieved by creating several scripts that would strip the virtual environment for commonly found artefacts and add other artefacts that would help simulate a real system such as different types of system files and commonly used applications.

The results of running malware samples in different virtual environments showed promise since several malware samples did in fact show signs of behaviour change when comparing the result of the two different environments. However, the results were not enough to conclude anything definite

since the sample size used for testing was small at 21 different malware samples. In order to make a definite conclusion, more malware samples would need to be tested as well as run the same malware samples multiple times in order to avoid anomalies. The system developed for testing did however work and shows promise in the sense that using the testing methodology in combination with the system made can lead to a change in malware behaviour.

In conclusion, it can be said that a system which can be used for testing malware sample behaviour was made and that it fulfilled the requirement of being able to showcase that removing or changing different artefacts and triggers would impact malware behaviour which was quantified with different ”events”. The results produced by the system showed promise, but more testing would be needed in terms of more malware samples in order to prove that the artefacts and triggers changed the malware behaviour. The project therefore also contributes with the following:

- A test platform for testing malware was made.
- An overview of common artefacts for virtual environments were identified.
- A description and demonstration of hot to remove and modify artefacts within a VM image.
- A method, based on different parameters such as events, can be used to measure and quantify a change in malware behaviour.

9 | Project Outlook and Future Work

When looking at the project and what was achieved through the finished system, several aspects could have been done in a different way or improved.
The first area would be the analysis and selection of what artefacts and triggers to implement into the system. Because of the number of different things to look for as seen during the analysis part of the report, selecting something that would cover most malware samples was near impossible. In order to make the selection easier, using a different testing methodology of using multiple different VM images with different artefact implementations would improve that accuracy of what artefacts impact malware behaviour the most.

Another area that was discussed during the project but turned out to be too much work was the scalability of the project. The original idea was to create a system which could be scaled easily which could have been done using Docker containers and perhaps an orchestrating tool such as Kubernetes. This would allow for easily adding more services that could point to different VM's which could be attached via the network. Due to the sheer size of such a task, the decision was made to focus on a smaller setup in order to test the idea of having different VM images for malware behaviour testing. With more time to work on the scalability, a port of Cuckoo and VirtualBox that could work and communicate within Docker containers could be a possibility, but due to the many different components as well as the many different network configurations, making sure it all works together could turn out to be impossible if any component fails to be implemented in Docker containers.

For the testing of malware sample behaviour, things could also have been done differently in terms of how the malware samples were tested in the different environments. For the tests, a single modified VM image was used for testing. This meant that the configuration in terms of the artefacts that were implemented were limited to a one configuration. In order to investigate how and what artefacts would have the biggest impact on malware behaviour, several different configurations would have to be made where each image used would have the same base, but would differ in what artefacts would be implemented. This also ties into the results and how they were presented in the report. Due to the nature of Cuckoo, each comparison would have to be manually made in order to determine if the malware samples changed behaviour based on the criteria that were made that defined what a change in malware behaviour is. In combination with using different image configurations, a tool could have been made that would replace Cuckoos tool in order to compare different results.
As an extension of the testing, an area which was difficult to work with was the selection of malware samples. Since the archive that was used didn't have anything informative other than randomised names on the malware samples, figuring out what sample was good to use and what sample was not was impossible. A more thorough method of selecting malware samples which are known to have sandbox evasion techniques would be a major improvement to the testing aspect of the project and would also allow testing different VM image configurations where different artefacts are implemented.

List of Figures

List of Tables

Bibliography

[1] a0rtega. a0rtega/pafish: Pafish is a demonstration tool that employs several techniques to detect sandboxes and analysis environments in the same way as malware families do. https://github.com/a0rtega/pafish, Nov 2019. (Accessed on 11/07/2019).

[2] Apriorit. How to reverse engineer software (windows) in a right way. https://www.apriorit.com/dev-blog/364-how-to-reverse-engineer-software-windows-in-a-right-way, Mar 2019. (Accessed on 10/11/2019).

[3] AutoHotKey. Autohotkey. https://www.autohotkey.com/, Nov 2019. (Accessed on 11/11/2019).

[4] AutoIt. Home - autoit. https://www.autoitscript.com/site/, Nov 2019. (Accessed on 11/11/2019).

[5] AV-Test. 170807-securityreport2017_e_final.indd. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf, Oct 2019. (Accessed on 20/10/2019).

[6] AV-Test. Malware statistics & trends report | av-test. https://www.av-test.org/en/statistics/malware/, Sep 2019. (Accessed on 18/09/2019).

[7] H. Bai, C. Hu, X. Jing, N. Li, and X. Wang. Approach for malware identification using dynamic behaviour and outcome triggering. https://ieeexplore.ieee.org/document/6748548, March 2014.

[8] Blacktop. blacktop/docker-cuckoo: Cuckoo sandbox dockerfile. https://github.com/blacktop/docker-cuckoo, Dec 2019. (Accessed on 12/18/2019).

[9] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. https://www.researchgate.net/publication/226302439_Automatically_Identifying_Trigger-based_Behavior_in_Malware, 01 1970.

[10] Krzysztof Cabaj, Luca Caviglione, Wojciech Mazurczyk, Steffen Wendzel, Alan Woodward, and Sebastian Zander. The new threats of information hiding: the road ahead. https://arxiv.org/abs/1801.00694, 2018. URL http://arxiv.org/abs/1801.00694.

[11] S. Cesare, Y. Xiang, and W. Zhou. Malwise; an effective and efficient classification system for packed and polymorphic malware. https://ieeexplore.ieee.org/document/6171162, June 2013.

[12] Checkpoint. Malware evolution: Pc-based vs. mobile - check point software. https://blog.checkpoint.com/2013/01/02/malware-evolution-pc-based-vs-mobile-2/, Oct 2019. (Accessed on 21/10/2019).

[13] Catalin Cimpanu. Siemens contractor pleads guilty to planting logic bomb in company spreadsheets | zdnet. https://www.zdnet.com/article/siemens-contractor-pleads-guilty-to-planting-logic-bomb-in-company-spreadsheets/, Jul 2019. (Accessed on 21/10/2019).

[14] Cisco. Threat spotlight: Dyre/dyreza: An analysis to discover the dga - cisco blog. https://blogs.cisco.com/security/talos/threat-spotlight-dyre, Oct 2019. (Accessed on 21/10/2019).

[15] Cuckoo. Analysis results — cuckoo sandbox v2.0.6 book. https://cuckoo.readthedocs.io/en/latest/usage/results/, Nov 2019. (Accessed on 11/11/2019).

[16] Cuckoo. Cuckoo sandbox book — cuckoo sandbox v2.0.6 book. https://cuckoo.readthedocs.io/en/latest/, Dec 2019. (Accessed on 12/21/2019).

[17] Cuckoo. Requirements — cuckoo sandbox v2.0.6 book. https://cuckoo.readthedocs.io/en/latest/installation/guest/requirements/#additional-software, Dec 2019. (Accessed on 29/12/2019).

[18] Cuckoo. What is cuckoo? — cuckoo sandbox v2.0.6 book. https://cuckoo.readthedocs.io/en/latest/introduction/what/#architecture, Nov 2019. (Accessed on 11/11/2019).

[19] deepinstinct. Malware evasion techniques part 2: Anti-vm blog - deep instinct. https://www.deepinstinct.com/2019/10/29/malware-evasion-techniques-part-2-anti-vm-blog/, Oct 2019. (Accessed on 29/12/2019).

[20] Milena Dimitrova. Malware trends 2018: How is the threat landscape shaping? https://sensorstechforum.com/malware-trends-2018-threat-landscape/, Jan 2018. (Accessed on 21/10/2019).

[21] Docker. Enterprise container platform | docker. https://www.docker.com/, Dec 2019. (Accessed on 12/18/2019).

[22] J. Donahue, A. Paturi, and S. Mukkamala. Visualization techniques for efficient malware detection. https://ieeexplore.ieee.org/document/6578845, June 2013.

[23] D. Fleck, A. Tokhtabayev, A. Alarif, A. Stavrou, and T. Nykodym. Pytrigger: A system to trigger extract user-activated malware behavior. https://ieeexplore.ieee.org/document/6657230, Sep. 2013.

[24] Flowm. Github - flowm/etherkey: Emulate a conventional usb keyboard with a scriptable, network capable microcontroller. https://github.com/Flowm/etherkey, Dec 2019. (Accessed on 23/12/2019).

[25] ilyaglow. ilyaglow/remote-virtualbox: Little package to do simple things with virtualbox remotely using it's soap api. https://github.com/ilyaglow/remote-virtualbox, Dec 2019. (Accessed on 12/19/2019).

[26] Osanda Malith Jayathissa. Analyzing an autohotkey malware | blog of osanda. https://osandamalith.com/2019/05/22/analyzing-an-autohotkey-malware/, May 2019. (Accessed on 11/11/2019).

[27] Kaspersky. Kaspersky-lab-whitepaper-machine-learning.pdf. https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf, Sep 2019. (Accessed on 18/09/2019).

[28] Kaspersky. Machine learning in cybersecurity | kaspersky. https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity, Sep 2019. (Accessed on 18/09/2019).

[29] Kaspersky. Sandbox | kaspersky. https://www.kaspersky.com/enterprise-security/wiki-section/products/sandbox, Oct 2019. (Accessed on 19/10/2019).

[30] Kaspersky. What makes the legal spying software — stalkerware — dangerous? | kaspersky official blog. https://www.kaspersky.com/blog/stalkerware-spouseware/26292/, Dec 2019. (Accessed on 29/12/2019).

[31] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. http://dl.acm.org/citation.cfm?id=2671225.2671244, 2014.

[32] RIA KULSHRESTHA. Malware detection using deep learning - towards data science. https://towardsdatascience.com/malware-detection-using-deep-learning-6c95dd235432, Jul 2019. (Accessed on 10/11/2019).

[33] LastLine. Carbanak malware — ninety five percent exhibits stealthy or evasive behaviors | lastline. https://www.lastline.com/labsblog/carbanak-malware-ninety-five-percent-exhibits-stealthy-or-evasive-behaviors/, Oct 2019. (Accessed on 21/10/2019).

[34] LoadRunner. Loadrunner: Application load testing software | micro focus. https://www.microfocus.com/en-us/products/loadrunner-load-testing/overview, Nov 2019. (Accessed on 11/11/2019).

[35] makeuseof. New pc? 15 must-have windows applications you should install first. https://www.makeuseof.com/tag/getting-a-new-pc-12-must-have-applications-to-install-first/, Jan 2019. (Accessed on 29/12/2019).

[36] Malwarebytes. Machinelearning/anomalous.100% - malwarebytes labs | malwarebytes labs | detections. https://blog.malwarebytes.com/detections/machinelearning-anomalous-100/, Nov 2019. (Accessed on 10/11/2019).

[37] McAfee. Stopping malware with a fake virtual machine | mcafee blogs. https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/stopping-malware-fake-virtual-machine/, Jan 2017. (Accessed on 21/10/2019).

[38] Microsoft. Acpi system description tables - windows drivers | microsoft docs. https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/acpi-system-description-tables, Dec 2019. (Accessed on 23/12/2019).

[39] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. https://ieeexplore.ieee.org/document/7958622, May 2017.

[40] nsmfoo. nsmfoo/antivmdetection: Script to create templates to use with virtualbox to make vm detection harder. https://github.com/nsmfoo/antivmdetection, Jan 2020. (Accessed on 08/01/2020).

[41] Lindsey O'Donnell. 2019 malware trends to watch | threatpost. https://threatpost.com/2019-malware-trends-to-watch/140344/, Jan 2019. (Accessed on 21/10/2019).

[42] Luana Pascu. Limitations of machine learning algorithms in malware detection. https://businessinsights.bitdefender.com/limitations-of-machine-learning-algorithms-in-malware-detection, Nov 2019. (Accessed on 10/11/2019).

[43] Symantec Security Response. Remote linux wiper found in south korean cyberattack | symantec connect community. https://www.symantec.com/connect/blogs/remote-linux-wiper-found-south-korean-cyber-attack, Mar 2013. (Accessed on 21/10/2019).

[44] Symantec Security Response. South korean banks and broadcasting organizations suffer major damage from cyberattack | symantec connect community. https://www.symantec.com/connect/blogs/south-korean-banks-and-broadcasting-organizations-suffer-major-damage-cyber-attack, Mar 2013. (Accessed on 21/10/2019).

[45] UKIT AI ROMAN STEINBERG. 6 areas where artificial neural networks outperform humans | venturebeat. https://venturebeat.com/2017/12/08/6-areas-where-artificial-neural-networks-outperform-humans/, Dec 2017. (Accessed on 03/01/2020).

[46] Andrew Sanders. Malware statistics, trends and facts in 2019. https://www.safetydetectives.com/blog/malware-statistics/, Jul 2019. (Accessed on 21/10/2019).

[47] scmagazine. Gravityrat malware detects virtualized environments by taking infected machines' temperature | sc media. https://www.scmagazine.com/home/security-news/malware/gravityrat-malware-detects-virtualized-environments-by-taking-infected-machines-temperature/, May 2018. (Accessed on 29/12/2019).

[48] P. Su, L. Ying, and D. Feng. Exploring malware behaviors based on environment constitution. https://ieeexplore.ieee.org/document/4724666, Dec 2008.

[49] Symantec. Does malware still detect virtual machines? | symantec connect community. https://www.symantec.com/connect/blogs/does-malware-still-detect-virtual-machines, Aug 2014. (Accessed on 29/12/2019).

[50] Tripwire. The four most common evasive techniques used by malware. https://www.tripwire.com/state-of-security/security-data-protection/the-four-most-common-evasive-techniques-used-by-malware/, Oct 2019. (Accessed on 21/10/2019).

[51] VirusTotal. Virustotal. https://www.virustotal.com/, Dec 2019. (Accessed on 12/18/2019).

[52] A. Walker, M. F. Amjad, and S. Sengupta. Cuckoo's malware threat scoring and classification: Friend or foe? https://ieeexplore.ieee.org/document/8666454, Jan 2019.

[53] Wikipedia. Adware - wikipedia. https://en.wikipedia.org/wiki/Adware, Oct 2019. (Accessed on 21/10/2019).

[54] Wikipedia. Computer worm - wikipedia. https://en.wikipedia.org/wiki/Computer_worm, Oct 2019. (Accessed on 21/10/2019).

[55] Wikipedia. Computer virus - wikipedia. https://en.wikipedia.org/wiki/Computer_virus, Oct 2019. (Accessed on 21/10/2019).

[56] Wikipedia. Fileless malware - wikipedia. https://en.wikipedia.org/wiki/Fileless_malware, Oct 2019. (Accessed on 21/10/2019).

[57] Wikipedia. Malware - wikipedia. https://en.wikipedia.org/wiki/Malware#Evasion, Oct 2019. (Accessed on 21/10/2019).

[58] Wikipedia. Ransomware - wikipedia. https://en.wikipedia.org/wiki/Ransomware, Oct 2019. (Accessed on 21/10/2019).

[59] Wikipedia. Spyware - wikipedia. https://en.wikipedia.org/wiki/Spyware, Oct 2019. (Accessed on 21/10/2019).

[60] Wikipedia. Steganography - wikipedia. https://en.wikipedia.org/wiki/Steganography, Oct 2019. (Accessed on 21/10/2019).

[61] Wikipedia. Stegomalware - wikipedia. https://en.wikipedia.org/wiki/Stegomalware, Oct 2019. (Accessed on 21/10/2019).

[62] Wikipedia. Trojan horse (computing) - wikipedia. https://en.wikipedia.org/wiki/Trojan_horse_(computing), Oct 2019. (Accessed on 21/10/2019).

[63] W. Yan, Z. Zhang, and N. Ansari. Revealing packed malware. https://ieeexplore.ieee.org/document/4639028, Sep. 2008.

[64] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, and Christian Rossow. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. https://link.springer.com/chapter/10.1007/978-3-319-45719-2_8, 2016.

[65] A. Young and Moti Yung. Deniable password snatching: on the possibility of evasive electronic espionage. https://ieeexplore.ieee.org/abstract/document/601339, May 1997.

[66] ytisf. Github - ytisf/thezoo: A repository of live malwares for your own joy and pleasure. thezoo is a project created to make the possibility of malware analysis open and available to the public. https://github.com/ytisf/theZoo, Nov 2019. (Accessed on 11/11/2019).

[67] L. Ďurfina, J. Křoustek, and P. Zemek. Psybot malware: A step-by-step decompilation case study. https://ieeexplore.ieee.org/document/6671321, Oct 2013.

A | Docker and Virtual Machines

The difference between Docker containers and Virtual Machines comes in the difference between the architecture that they use. For both, the common component is that they are running some sort of application that has access to resources by the host system. The basic difference between them is that a Docker container relies on virtual isolation and doesn't rely on an actual virtual machine to execute applications, whereas a virtual machine emulates a real operating system and hardware. A virtual machine reserves resources from the host system and emulates hardware and operating system in isolation. A Docker container does away with that by having direct access to the host machines resources. Comparing the two can be seen on fig. A.1.

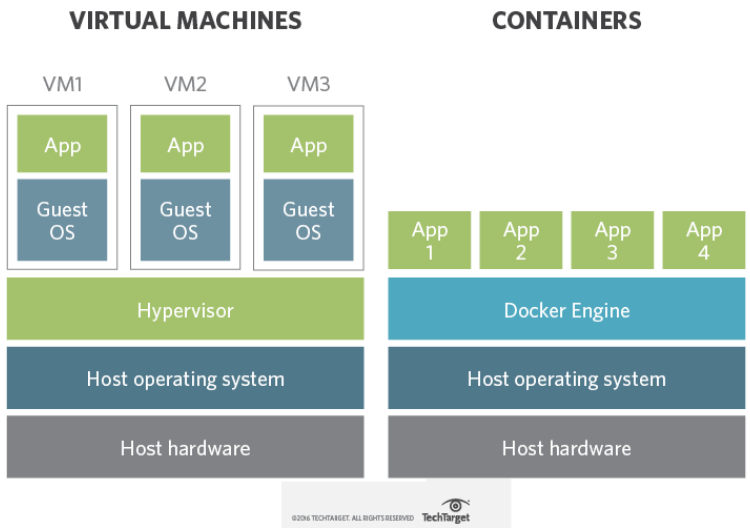


Figure A.1: Overview of the differences between a virtual machine and docker containers (credit: techtarget.com)

As seen on the figure, containers use some type of middleware which in this case is the Docker engine. The engine is used to run and maintain the different containers that are selected to run. The containers are taken and run inside their own virtual environment but still have access to the host machines resources.
For virtual machines, a middleware is also present in the form of a hypervisor. The hypervisor manages different virtual machines where each includes its own virtual environment complete with a full operating system which acts as a normal system. In comparison a container needs to be built before it is run which is achieved by building the container in different layers. This means that in order to add anything to the container such as more applications, the container needs to be rebuilt. Here the virtual machine has the advantage of not needing to be rebuilt since applications can be installed as if was a normal system.

B | Anti VM Detection Log

Output of running the anti VM script. Big chunks were removed to avoid unnecessary text.

```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

Directory: C:\Users\Nikolaj\Desktop\avmd
PS C:\Users\Nikolaj\Desktop\avmd> Set-ExecutionPolicy -ExecutionPolicy Unrestricted
PS C:\Users\Nikolaj\Desktop\avmd> .\OptiPlex7010.ps1

PSPath          : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System
PSDrive         : HKLM
PSProvider      : Microsoft.PowerShell.Core\Registry
SystemBiosVersion : DELL - v02

PSPath          : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System
PSDrive          : HKLM
PSProvider       : Microsoft.PowerShell.Core\Registry
VideoBiosVersion : v02

PSPath         : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System
PSDrive        : HKLM
PSProvider     : Microsoft.PowerShell.Core\Registry
SystemBiosDate : 09/19/12

PSPath: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\SOFTWARE\Microsoft\WindowsNT\CurrentVersion
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
InstallDate  : 1533512470

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\SOFTWARE\Microsoft\Internet Explorer\SQM
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
InstallDate  : 1533512470

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
MachineGuid  : 5c17761a-43f8-485f-aa68-5f87d13fa7c5

CodeBase            : file:///C:/Windows/assembly/GAC_64/System.Web/2.0.0.0__b03f5f7f11d50a3a/System.Web.dll
EscapedCodeBase     : file:///C:/Windows/assembly/GAC_64/System.Web/2.0.0.0__b03f5f7f11d50a3a/System.Web.dll
FullName            : System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Evidence            : {...}
ManifestModule      : System.Web.dll
ReflectionOnly      : False
Location            : C:\Windows\assembly\GAC_64\System.Web\2.0.0.0__b03f5f7f11d50a3a\System.Web.dll
ImageRuntimeVersion : v2.0.50727
GlobalAssemblyCache : True
HostContext         : 0

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
  ↪  NT\CurrentVersion
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
ProductId    : 02948-646-8799910-37269

```

```
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet
  ↪  Explorer\Registration
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
ProductId    : 02948-646-8799910-37269

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
  ↪  NT\CurrentVersion\DefaultProductKey
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
ProductId    : 02948-646-8799910-37269

Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Product key from registry cleared successfully.

PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Sysinternals
PSDrive       : HKCU
PSProvider    : Microsoft.PowerShell.Core\Registry
PSIsContainer : True
Name          : HKEY_CURRENT_USER\Software\Sysinternals

PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Sysinternals\VolumeId
PSDrive       : HKCU
PSProvider    : Microsoft.PowerShell.Core\Registry
PSIsContainer : True
Name          : HKEY_CURRENT_USER\Software\Sysinternals\VolumeId

PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Sysinternals\VolumeId
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
EulaAccepted : 1


VolumeId v2.1 - Set disk volume id
Copyright (C) 1997-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

Volume ID for drive c: updated to 7443-96c9
__GENUS          : 2
__CLASS          : __PARAMETERS
__DYNASTY        : __PARAMETERS
__PROPERTY_COUNT : 1
__DERIVATION     : {}
ReturnValue      : 0

LastWriteTime : 16-12-2019 16:25:44
Length        : 0
Name          : kummerspeck

CodeBase        : file:///C:/Windows/assembly/GAC_MSIL/System.Windows.Forms/
                  2.0.0.0__b77a5c561934e089/System.Windows.Forms.dll
EscapedCodeBase : file:///C:/Windows/assembly/GAC_MSIL/System.Windows.Forms/
                  2.0.0.0__b77a5c561934e089/System.Windows.Forms.dll
FullName            : System.Windows.Forms, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
Evidence            : {...}
ManifestModule      : System.Windows.Forms.dll
ReflectionOnly      : False
Location            : C:\Windows\assembly\GAC_MSIL\System.Windows.Forms\
                      2.0.0.0__b77a5c561934e089\System.Windows.Forms.dll
ImageRuntimeVersion : v2.0.50727
GlobalAssemblyCache : True
HostContext         : 0

OK
LastWriteTime : 16-12-2019 16:25:52
Name          : nikolaj-OptiPlex-7010.doc/ppt/jpeg/doc/jpg/xlsx/zip/html/docx/xls/pptx/bmp/gif

```

```
.divx=WMP11.AssocFile.WAV
.mkv=WMP11.AssocFile.WAV
.m4p=WMP11.AssocFile.WAV
.skype=WMP11.AssocFile.WAV
.flac=WMP11.AssocFile.WAV
.psd=WMP11.AssocFile.WAV
.torrent=WMP11.AssocFile.WAV
```

C | Scripts Used for the System

This appendix includes all the different scripts that were used for the system.

Installprereq.sh

```
#!/bin/bash
echo "Installing Updates and dependencies..."
sudo apt-get update && sudo apt-get upgrade -y && sudo apt-get autoremove -y
sudo apt-get install python python-pip python-dev libffi-dev libreadline-gplv2-dev libncursesw5-dev libssl-dev
  ↪  libsqlite3-dev -y
sudo apt-get install python-virtualenv python-setuptools -y
sudo apt-get install libjpeg-dev zlib1g-dev -y
sudo apt-get install libxml2-dev libxslt1-dev libevent-dev libpcre3 libpcre3-dev libtool libpcre++-dev g++ -y
sudo apt-get install git automake dkms unzip wget python-sqlalchemy python-bson python-dpkt python-jinja2 -y
sudo apt-get install python-magic python-mysqldb python-gridfs python-bottle python-pefile python-chardet -y

echo "Python 3"
cd /usr/src
sudo wget https://www.python.org/ftp/python/3.8.0/Python-3.8.0.tgz
sudo tar -xvf Python-3-8.0.tgz
cd Python-3-8.0
./configure
sudo make && make install

echo "Pip"
pip install --upgrade pip
sudo -H pip install pillow -y

echo "MongoDB"
sudo apt-get install mongodb -y

echo "Postgresql"
sudo apt-get install postgresql libpq-dev -y
sudo pip install psycopg2

echo "Tcpdump"
sudo apt-get install tcpdump apparmor-utils -y
sudo aa-disable /usr/sbin/tcpdump
sudo setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
getcap /usr/sbin/tcpdump

sudo -H pip install lxml cybox==2.0.1.4 maec==4.0.1.0 "Django<2"
```

Installpostreq.sh

```
#!/bin/bash
echo " - Installing -"
echo "Adding cuckoo user"
sudo adduser --disabled-login -gecos "" cuckoo
sudo usermod -G cuckoo cuckoo

echo "ssdeep"
sudo apt-get install ssdeep python-pyrex subversion libfuzzy-dev -y

echo "M2Crypto"
sudo apt-get install swig -y
```

```
sudo -H pip install m2crypto==0.31.0

echo "Volatility"
git clone https://github.com/volatilityfoundation/volatility.git
cd volatility
sudo python setup.py install
cd ..

echo "Distrom"
wget -O distormv3.4.1.tar.gz https://github.com/gdabah/distorm/archive/v3.4.1.tar.gz
tar -zxvf distormv3.4.1.tar.gz
cd distorm-3.4.1
sudo python setup.py install
sudo apt-get install libjansson-dev libmagic-dev
sudo apt-get install libtool-bin
cd ..

echo "PyCrypto"
sudo -H pip install pycrypto ansible --upgrade IPython==5.0 jupyter openpyxl ujson

echo "Yara"
wget -O yarav3.11.0.tar.gz https://github.com/VirusTotal/yara/archive/v3.11.0.tar.gz
tar -zxvf yarav3.11.0.tar.gz
cd yara-3.11.0
sudo ./bootstrap.sh
sudo ./configure --with-crypto --enable-magic -enable-cuckoo
sudo make
sudo make install
sudo -H pip install yara-python

echo "FTP Server"
sudo mkdir -p /home/tarik/vmshared/pub
sudo chown -R cuckoo:cuckoo /home/tarik/vmshared
sudo chmod -R ug=rwX,o=rX /home/tarik/vmshared/
sudo chmod -R ugo=rwX /home/tarik/vmshared/pub
sudo apt-get install vsftpd

echo "Adding stuff to /etc/vsftpd.conf"
sudo sed -i 's/#write_enable=YES/write_enable=YES/g' /etc/vsftpd.conf
sudo sed -i 's/#anon_upload_enable=YES/anon_upload_enable=YES/g' /etc/vsftpd.conf
sudo sed -i 's/#anon_mkdir_write_enable=YES/anon_mkdir_write_enable=YES/g' /etc/vsftpd.conf
sudo bash -c 'echo "listen_address=192.168.100.1" >> /etc/vsftpd.conf'
sudo bash -c 'echo "listen_port=2121" >> /etc/vsftpd.conf'
sudo bash -c 'echo "anon_root=/home/tarik/vmshared" >> /etc/vsftpd.conf'
sudo bash -c 'echo "anon_umask=000" >> /etc/vsftpd.conf'
sudo bash -c 'echo "chown_upload_mode=0666" >> /etc/vsftpd.conf'
sudo bash -c 'echo "pasv_enable=Yes" >> /etc/vsftpd.conf'
sudo bash -c 'echo "pasv_min_port=10090" >> /etc/vsftpd.conf'
sudo bash -c 'echo "pasv_max_port=10100" >> /etc/vsftpd.conf'

sudo service vsftpd restart
sudo service vsftpd status
sudo ufw allow 20/tcp
sudo ufw allow 21/tcp
sudo ufw allow 990/tcp
sudo ufw allow 10090:10100/tcp
sudo ufw enable
sudo ufw disable

echo "Cuckoo"
cat <<EO {
Do the following:
virtualenv venv
sudo apt-get remove python-dpkt
sudo su
. venv/bin/activate
sudo pip install -U pip setuptools
sudo pip install -U cuckoo
cuckoo -d
```

```
cp /root/.cuckoo/agent/agent.py /home/tarik/vmshared/agent.pyw
cd /root/.cuckoo
service mongodb start
EO
}
```

Installvirtualbox.sh

```
#!/bin/bash
echo "Installing VirtualBox"
codename=(lsb_release --codename | cut -f2)

echo deb [arch=amd64] https://download.virtualbox.org/virtualbox/debian bionic contrib | sudo tee -a
  ↪  /etc/apt/sources.list.d/virtualbox.list
wget -q https://www.virtualbox.org/download/oracle_vbox_2016.asc -O- | sudo apt-key add -
wget -q https://www.virtualbox.org/download/oracle_vbox.asc -O- | sudo apt-key add -
sudo apt-get update
sudo apt-get install virtualbox-6.1
```

Installantivmdetection.sh

```
#!/bin/bash
echo " - Installing -"
sudo apt-get install python3-pip libcdio-utils acpica-tools mesa-utils
sudo pip3 install -r requirements.txt
wget https://download.sysinternals.com/files/VolumeId.zip https://www.nirsoft.net/utils/devmanview-x64.zip
hostname > computer.lst
whoami > user.lst
sudo python3 antivmdetect.py
echo "Installation complete"
```

InstallNetworkConfiguration.sh

```
#!/bin/bash
echo "- Installing -"

sudo apt install net-tools
sudo vboxmanage hostonlyif create
sudo vboxmanage hostonlyif ipconfig vboxnet0 --ip 192.168.56.1
sudo vboxmanage modifyvm sandbox-win7-01 --hostonlyadapter1 vboxnet0
sudo vboxmanage modifyvm sandbox-win7-01 --nic1 hostonly

sudo iptables -t nat -A POSTROUTING -o eth0 -s 192.168.56.0/24 -j MASQUERADE
sudo iptables -P FORWARD DROP
sudo iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -s 192.168.56.0/24 -j DROP
sudo iptables -A FORWARD -s 192.168.56.0/24 -d 192.168.56.0/24 -j ACCEPT

echo 1 | sudo tee -a /proc/sys/net/ipv4/ip_forward
sudo sysctl -w net.ipv4.ip_forward=1

sudo apt-get install iptables-persistent
```

D | Malware Samples

This appendix includes information about the 21 different malware samples used during testing for malware behaviour.

Sample 1 - Trojan-Generic

- Authentihash: 3e765c2cfafdfd2610c0ac7f8a8def05d04d4d5cb4190b064f6df621f2b57bc4
- MD5: bf9b26669990c1a4d2f71154569e55cc
- Permalink: https://www.virustotal.com/file/00a0a5620b21090429b530dd66c6745eaaeb0f7c1be14150127e7801729ffb9a/analysis/1557449218/
- Scan_id: 00a0a5620b21090429b530dd66c6745eaaeb0f7c1be14150127e7801729ffb9a-1557449218

Sample 2 - Trojan-Adware-Ransom

- Authentihash: 076e93f566b063fed0d08006e3ac52bb72ce34f7e034b9bfd332af33b8f40bf1
- MD5: faf96aa3f8c97e18c0be2caac88762ee
- Permalink: https://www.virustotal.com/file/000a2c5a191712db64311536e6df70ed14f75a47fae37d2592817d66ba682660/analysis/1557449638/
- Scan_id: 000a2c5a191712db64311536e6df70ed14f75a47fae37d2592817d66ba682660-1557449638

Sample 3 - Virut-Botnet

- Authentihash: a4c8fdd7f1d8cfad6ba583e906980ca04476813881a7abe5a3d2fb6dcc77e970
- MD5: b76622fceabd9dcce69e19fb088843da
- Permalink: https://www.virustotal.com/file/00a3c4d66f0fe3b0f228ed07e48452254fc3246f378d8794f42d98be6a394b5e/analysis/1557448255/
- Scan_id: 00a3c4d66f0fe3b0f228ed07e48452254fc3246f378d8794f42d98be6a394b5e-1557448255

Sample 4 - TrojanWorm-Prepender-Shodi

- Authentihash: 4301221c3acddefd310257421bb396543dc67c471f2f522aa17bfacde1e372b1
- MD5: c810f46d6ed235718dc51d16e80c5501
- Permalink: https://www.virustotal.com/file/00a5d286a0cb8c41b93a52ce80a8e7d28b70724c6919080858f877406eb30cf7/analysis/1557462074/
- Scan_id: 00a5d286a0cb8c41b93a52ce80a8e7d28b70724c6919080858f877406eb30cf7-1557462074

Sample 5 - Trojan-Dloader

- Authentihash: ac22ea2a47418dec73861cbcc5ee8a765625ecdc129ec8440dbfc4fd301fc382
- MD5: 46b0950b386b3daef7ca327c84daba5b
- Permalink: https://www.virustotal.com/file/00a884d53f3cd9df407e5dc24da0f50740b80f8b543bee856251fcaea4aa7ea8/analysis/1557446876/
- Scan_id: 00a884d53f3cd9df407e5dc24da0f50740b80f8b543bee856251fcaea4aa7ea8-1557446876

Sample 6 - Trojan-GameModding

- Authentihash: 215ae1ad0df93eb90185ebd813a8d7f1ecfd4a6740c9af73b811b81995692271
- MD5: 4030413c47414961322c2b5d46be9b87
- Permalink: https://www.virustotal.com/file/00aa31683f9292a003aed7f91b4f4113113f5cf5c30075a14305269bfafff921/analysis/1557440725/
- Scan_id: 00aa31683f9292a003aed7f91b4f4113113f5cf5c30075a14305269bfafff921-1557440725

Sample 7 - Dropped-Backdoor-Prorat

- Authentihash: 5550f0a282316279eaeabeb778d05d104263d8e1b8a058f99caa5ca9539bc197
- MD5: 59402be7d5db53e5bb73ef1b1b0ba3fd
- Permalink: https://www.virustotal.com/file/00ac5624960a04fb946491f3e35c84e2d614644cd164c07b19f0651754ecc200/analysis/1557451874/
- Scan_id: 00ac5624960a04fb946491f3e35c84e2d614644cd164c07b19f0651754ecc200-1557451874

Sample 8 - Worm-Generic-Porn

- Authentihash: fec004c3fdf080001cc8e318d2d0a0c13e21a3027abfc51837f096730bc2a234
- MD5: b922d0cc57568122580b543839e42c8a
- Permalink: https://www.virustotal.com/file/00acad78bb0da3e66f751f792282a22eae89d7fb2760d61aee33f99eaa64f718/analysis/1557454492/
- Scan_id: 00acad78bb0da3e66f751f792282a22eae89d7fb2760d61aee33f99eaa64f718-1557454492

Sample 9 - Virut

- Authentihash: 6db2b419e1d13d17e35dd49318ec35e64d033f7ab7855b587f8d53d17c272610
- MD5: b78d8dd916fa3cd9e87ece5654511d88
- Permalink: https://www.virustotal.com/file/00ae87107a422d7e24e4f83870d98939a41d02a22e04d71c3dcc6b9ff183c5fb/analysis/1557463448/
- Scan_id: 00ae87107a422d7e24e4f83870d98939a41d02a22e04d71c3dcc6b9ff183c5fb-1557463448

Sample 10 - FileRepMalware-Worm

- Authentihash: 6c25efdb89b3c83316cc2d87dfaf23d4466d0ff135230469d413ea7c8ec82bec
- MD5: ba733e37ef6aee948c09db7018468010
- Permalink: https://www.virustotal.com/file/00aece6855c7787c9b4c97814ca9c8a3c97644df74cf27266360c25df63b8367/analysis/1557460415/
- Scan_id: 00aece6855c7787c9b4c97814ca9c8a3c97644df74cf27266360c25df63b8367-1557460415

Sample 11 - Trojan-Generic

- Authentihash: 9299c9523a3940d8a154ba4541f3304614978e428039366d477e80505c31d610
- MD5: a334851b61130c2e8540b5c2518d2ad1
- Permalink: https://www.virustotal.com/file/00b7c13e7370062209067a59a183741b317dd5a6cea0ac218eb4424dca3d6e0e/analysis/1557447611/
- Scan_id: 00b7c13e7370062209067a59a183741b317dd5a6cea0ac218eb4424dca3d6e0e-1557447611

Sample 12 - Trojan-Backddoor-Wabot

- Authentihash: 882bc4959fc34d00bd02c18d7fc2f5fbdc7550e89c1901c6ac02344585059ba2
- MD5: 61ed9a49a46ce07838aede8a7818e339
- Permalink: https://www.virustotal.com/file/00b9b118233ddb6f2e76af3696cd220087f2ac14aee0a15ea2564e2a85b249fb/analysis/1557449658/
- Scan_id: 00b9b118233ddb6f2e76af3696cd220087f2ac14aee0a15ea2564e2a85b249fb-1557449658

Sample 13 - Trojan-Ransom-Gandcrab

- Authentihash: bef736adbaf77090f3d8af86d921269f7d4d4107b268501ee44b451cef925143
- MD5: f69166e74d77309c5681fa8453e26f05
- Permalink: https://www.virustotal.com/file/00b78c0e37d92d64886a063373431dcad6787721b888de759ae9f22d50a3ac40/analysis/1557465616/
- Scan_id: 00b78c0e37d92d64886a063373431dcad6787721b888de759ae9f22d50a3ac40-1557465616

Sample 14 - Trojan-MSIL-Bladabindi

- Authentihash: 03fa076c73dd0718b133491a42b20eb59f9fdc56215d7a479628f82db380d525
- MD5: ca9dfe5d4f6e8d532e88d0746f544571
- Permalink: https://www.virustotal.com/file/00b95e3bee4f6a72b2b28fc8fc76a41e55a66df850d00aac4e3ca9e60b270bd8/analysis/1557452979/
- Scan_id: 00b95e3bee4f6a72b2b28fc8fc76a41e55a66df850d00aac4e3ca9e60b270bd8-1557452979

Sample 15 - Trojan-Generic

- Authentihash: 142518c63cd3a29e5e41ec6799de4d2ed349877eb14af16f70cfa6a8afed7bf5
- MD5: baaecd7bd97845476107b35c45dbf4a4
- Permalink: https://www.virustotal.com/file/00b3182e5da1141e8000292e974cac32ade84a40d556682df32b3448782961ca/analysis/1557468610/
- Scan_id: 00b3182e5da1141e8000292e974cac32ade84a40d556682df32b3448782961ca-1557468610

Sample 16 - Trojan-Shipup

- Authentihash: d51374b9f54ff9c831335c3c2445717ea97f7176cc7698603e2685ffcf7eaaaf
- MD5: a8f3031d3657fe9b46f638ba5457a9d6
- Permalink: https://www.virustotal.com/file/00bab30b5ac84c4fa50227c94d888095b8418d875cb934804b31c0f0acb77761/analysis/1557466602/
- Scan_id: 00bab30b5ac84c4fa50227c94d888095b8418d875cb934804b31c0f0acb77761-1557466602

Sample 17 - Virus-Virut-Generic

- Authentihash: 7a81ad7dc9f33ff2e7646085646d13c2209cb03aa41fd0a659425b290eee8ae0
- MD5: c52c0016b883b087ef514a428d130fc8
- Permalink: https://www.virustotal.com/file/00bb5b1220f08c994b0dbe26d4815bc8cca749dd9b28794a060f64a77de024fd/analysis/1557448288/
- Scan_id: 00bb5b1220f08c994b0dbe26d4815bc8cca749dd9b28794a060f64a77de024fd-1557448288

Sample 18 - Trojan-Downloader-Injector

- Authentihash: 4dc7943affada702460572144bdda55a2a8c2d118e80a9a207a16b656612655b
- MD5: df7bfcab29e4ab1a88fca7712556d041
- Permalink: https://www.virustotal.com/file/00c0a11036619cca9a41e512d1127166357ab300a8bb7488fbe5368a0667b27d/analysis/1557468323/
- Scan_id: 00c0a11036619cca9a41e512d1127166357ab300a8bb7488fbe5368a0667b27d-1557468323

Sample 19 - Trojan-Downlaoder-Waski

- Authentihash: 4cf010d351f6dca825c3ca812cc104907c516246649f7b84ff3c64508f0c05b8
- community_reputation: 0
- MD5: fb9f49145e1d4be72bca7987238e8af2
- Permalink: https://www.virustotal.com/file/00c1eb7bba0d6c92367452199ae58d84bda13513d0c642862af44c998a0f3fc5/analysis/1557467452/
- Scan_id: 00c1eb7bba0d6c92367452199ae58d84bda13513d0c642862af44c998a0f3fc5-1557467452

Sample 20 - Trojan-Dropper-Generic

- Authentihash: 9f86f4b23b0d43796290abd50bba38ce3117afd7cee418fd01c4c867a435310c
- MD5: 45dd58d182f906c10abaa43aa0216c5a
- Permalink: https://www.virustotal.com/file/00c7d6b5416c5b96c4757001a1a707ee227c0aa7d758ea3977d1e53a60f446f4/analysis/1557445123/
- Scan_id: 00c7d6b5416c5b96c4757001a1a707ee227c0aa7d758ea3977d1e53a60f446f4-1557445123

Sample 21 - Trojan-FileRepMalware

- Authentihash: 334e5742dc80bf29e544f525360f5e1a2eddb0d7f1aad4cbcb021100542d8c88
- MD5: 869dc9f56833bf7bd4313d535e03eaf0
- Permalink: https://www.virustotal.com/file/a388018c75b6d54d27f447c379c3b7d417a8fd9d0d8903d83376c56b0d28c05d/analysis/1557462600/
- Scan_id: a388018c75b6d54d27f447c379c3b7d417a8fd9d0d8903d83376c56b0d28c05d-1557462600