SandBox for IoT Malware analysis (Diseker)

Máster Universitario en Ciberseguridad y Privacidad

2020/2021

Autor: **Oussama El Azizi**
Dirigido por**: Carlos Hernández Gañán**
Fecha de entrega**: diciembre 2020**

Fecha del Trabajo	
Título del Trabajo:	*Sandbox For IoT Malware (Diseker)*
Nombre del Autor:	*Oussama El Azizi*
Nombre del supervisor:	*Carlos Hernández Gañán*
Fecha de entrega (mm/aa):	*12/2020*
Titulación:	*Máster Universitario en Ciberseguridad y Privacidad*
Área del trabajo final de máster:	*Seguridad en la internet of things*
Idioma del trabajo	*Inglés*
Palabras clave:	*Sandbox, malware, malware analysis, IoT*

Resum.

El marcat dels dispositius del internet de les coses ha estat en augment continu i ràpid els últims anys, afegint nou dispositius connectats a les nostres cases, hospitals i ens ajuda controlar i supervisar la nostra salut i propietat. Però l'augment dels dispositius connectats amb recursos limitats no es sempre positiu, perquè la majoria dels dispositius al mercat han segut implementat amb sistemes criptogràfic febles degut als recursos limitat del dispositiu, dispositius sense estandardització en el recursos o configuració i més gran nombre de empreses venen els dispositius sense un servei de instal·lació o gestió segura.

Tots els punts negatius anteriors han fet que la superfície de atac sigues molt gran y accessible, que al seu torn ha augmentat l'amenaça. Aquest situació va crear un gran interès en la implementació de programés maliciosos per als dispositius de les coses, el dispositius infectats són utilitzats en atacs com DDoS, enviar correus spam o s'utilitzen per minar moneda digital.

En aquest projecte se divulgarà la implementació de una Sandbox pera a programes maliciosos dedicats als dispositius IoT, aquest Sandbox proporcionarà una gran verbositat als analistes i els ajudarà entendre el comportament del programari maliciós amb l'objectiu de mitigar possibles atacs. Aquest Sandbox, nom Diseker, va ser capaç de extreure suficients detalls de una mostra dels programes maliciós i establir un patró de comportament comú.

Resumen.

El mercado de dispositivos del internet de las cosas ha estado aumentado de forma rápido en los últimos años, añadiendo nuevos dispositivos conectados a nuestras casas, hospitales y nos ayudan también a control nuestro estado de salud. Pero el aumento de estos dispositivos conectados con recursos limitados no es siempre positivo, ya que la industria opta a la producción y acabó generando dispositivos con sistemas criptográficos pobres debido a lo recursos limitados, ha creado muchos dispositivos con una falta de estandarización y también optó a la desplegar dispositivos sin un servicio de instalación o gestión seguro.

Todos los puntos negativos anteriores han hecho que la superficie de ataque a esos dispositivos sea muy grande lo que aumentó la amenazas que provocó también un gran auge de programas maliciosos dedicados dispositivos con recursos limitados, los dispositivos infectados acaban siendo utilizados en ataques como DDoS, enviar correo no deseado o minar criptomonedas.

En este proyecto se divulgará la implementación de una Sandbox para programas maliciosos dedicados a dispositivos IoT, esta Sandbox va a proporcionar una gran verbosidad a los analistas y les ayudará a entender el comportamiento del código malicioso. Esta Sandbox con nombre Diseker, fue capaz de dar suficiente detalle de los códigos maliciosos en este proyecto y se pudo establecer un patrón de comportamiento de la muestra utilizada, lo que demuestra el potencial de una herramienta parecida.

Abstract.

The market of IoT devices has been increasing rapidly in the last few years, adding new devices and tools to homes, adding new tools that can be managed remotely to hospitals and allowing us to monitor our health and security very closely by using wearables and installing cameras in our houses, but the fast and rapid increase of those limited resource devices made the industry start developing new devices without standardization, using weak cryptography systems that can be easily broken due to the limited resources or by deploying devices without the proper services to install them in houses or hospitals (such as cameras and monitorization devices).

The lack of standardization, weak security configurations and outdated systems used by the IoT devices in the market, has made the IoT devices an easy target to threat actors which in turn increased the presence of IoT malware on the internet. Those threat actors take advantage of the presence of such security weak devices and use them for attacks such DDoS, mining or spamming.

In this project I will be discussing a readapted sandbox for IoT devices that will help security analysts tun malicious code in it and understand it behaviour which will help them extract IOCs and create signatures to protect network and devices from being used maliciously. This sandbox with the name Diseker, was successful of analysing multiple malware instances as well as helped established a pattern performed by most of the malware in the dataset.

Index

1 INTRODUCTION 1
1.1 STATE OF THE ART 2
1.2 OBJECTIVES 3
1.3 METHODOLOGY 3
1.4 EXPECTED TASKS 4
1.5 TIME PLAN 4

2 PROJECT REQUIREMENTS 6

3 DESIGN 7
3.1 CHALLENGES TO CONSIDER 7
3.1.1 Diversity 7
3.1.2 Library linking 7
3.2 ARCHITECTURE 8
3.3 DATA STRUCTURES 12
3.4 DATABASE DESIGN 14
3.5 SANDBOX COMPONENTS 15
3.5.1 Virtual machine 16
3.5.2 Virtual IoT devices 16
3.5.3 Database 17

4 IMPLEMENTATION 18
4.1 MAIN SCRIPT (DISEKER) 18
4.2 ORCHESTRATION 19
4.3 STATIC ANALYSER 20
4.4 DYNAMIC ANALYSER 21
4.4.1 Preparing the environment 21
4.4.2 Starting the emulation 30
4.4.3 Extracting the logs 31
4.5 PARSING 32
4.5.1 Syscall parser 32
4.5.2 Network parser 33
4.6 REPORT GENERATION 34
4.7 NETWORK CONFIGURATION 36

5 TESTING 38
5.1 TESTING METHODOLOGY 39
5.2 TESTING USER-SANDBOX INTERACTION 39
5.3 TESTING SANDBOX ANALYSIS 41
5.3.1 Analysis of a tested sample 43

6 CONCLUSIONS AND FUTURE WORK 48

7 REFERENCES 49

Tables Index

TABLE 1 USER-SANDBOX INTERACTION TESTS 40
TABLE 2 SAMPLES TESTED IN THE SANDBOX 41
TABLE 3 RESULTS OF TESTING THE SAMPLES 42

Figures Index

FIGURE 1 DATA FLOW OF THE ANALYSIS PROCESS 3
FIGURE 2 TIMELINE OF THE PROJECT 5
FIGURE 3 FLOW DIAGRAM OF THE SANDBOX 8
FIGURE 4 ARCHITECTURE AND INTERACTION OF THE COMPONENTS OF THE STATIC AND DYNAMIC ANALYSERS 9
FIGURE 5 CLASS DIAGRAM FOR THE STATIC AND DYNAMIC ANALYSERS 11
FIGURE 6 INTERACTIVE SESSION BEHAVIOUR DIAGRAM 19
FIGURE 7 BUILDROOT MENU 22
FIGURE 8 FILESYSTEM CONFIGURATION IN BUILDROOT 22
FIGURE 9 TOOLCHAIN CONFIGURATION 23
FIGURE 10 SYSTEM CONFIGURATION 23
FIGURE 11 BUILDROOT TARGET PACKAGES 24
FIGURE 12 DIRECTORY TREE OF THE LOGS 29
FIGURE 13. HTML TEMPLATE OF THE REPORT 35
FIGURE 14. SAMPLES DISTRIBUTION IN THE DATASET 39
FIGURE 15 SAMPLE DETAILS PROVIDED BY THE SANDBOX 43
FIGURE 16 ENTROPY GRAPH OF THE SAMPLE 44
FIGURE 17 STRINGS SAMPLES FROM THE STATIC ANALYSER 44
FIGURE 18 NAMES OF BROWSER AGENTS AND ADDRESSES FOUND BY THE STATIC ANALYSER 45
FIGURE 19 SAMPLE OF FUNCTION NAMES EXTRACTED BY THE STATIC ANALYSER 45
FIGURE 20 PROCESS TREE OF THE MALWARE 46
FIGURE 21 STATE OF PROCESS EXECUTING IN THE DEVICE BEFORE AND AFTER THE MALWARE WAS EXECUTED 46
FIGURE 22 SAMPLE OF SYSTEM CALLS OF PID 268 47
FIGURE 23 SYSTEM CALLS OF PID 270 47
FIGURE 24 SYSTEM CALLS OF PID 271 47

Code Index

Code 1 System call events Json schema 13
Code 2 Network events Json schema 14
Code 3 Database schema 15
Code 4 Script used to apply Audit rules to the operating system 27
Code 5 Script used to execute the malicious file, start network monitoring and search audit logs 28
Code 6 Recursive script used to get logs from child processes 29
Code 7 Script used to extract logs from IoT filesystem 31

1 Introduction

Nowadays there is huge increase in the use of IoT devices in out day to day life, from smartphones, smartwatches, cameras to lightbulbs, fridges, and microwaves, and they can be classified in two types:

- General-purpose IoT devices: devices that help perform are day to day tasks, and everyone uses them for simple household functions. These types can go from a small sensor to heating, ventilation to complex air conditioning systems.
- Special-purpose IoT devices: These devices are mostly used by professionals to help them achieve a goal, they are checked regularly and used very often compared to the previous ones, these devices range from medical devices installed in hospitals, to smartwatches to monitor heart rates, etc [12].

All those devices are connected to some network of some sort, all of those added elements to the internet is not an issue, the issue lies on the fact that the increase use of those devices pushed the industry to create new products without having security standards, producing poorly developed devices, poorly tested devices or just the lack of resources in the devices which led to the usage of lightweight cryptosystems or validation tools, to all of the previous reasons we need to add misconfiguration and use of default configuration by the user which generally lead to exploitation.

The lack of security standards or just the misconfiguration of those devices made their attack surface big enough to capture the attention of multiple threat actors, which lead to the exploitation and use of those devices in malicious ways, those malicious uses go from spaying, data theft to big attacks controlled by botnets such as Mirai, Hide and seek (HNS) or Zollard, those botnets then use the devices for [12]:

- DDoS attacks
- Email spam campaigns
- Identity theft
- Cryptocurrency mining
- Click-fraud

It is common to consider that all the infections to IoT devices are related to botnets, that was proven to be wrong since researchers from the cyber security firm Pen Test Partners were able to successfully perform a ransomware attack against an IoT device and simulated it in DefCon hacking conference [19], so the range of possible infections is increasing with the technology and use of IoT devices. This idea of new and improved threats created a huge interest in developing new tools and environments to analyse IoT malware through conventional and reinvented malware analysis techniques.

The big growth in IoT devices use and creation, and the growth in exploitation and use by threat actors motivated me to get involved in developing a new easy to use system that can analyse automatically different malware for IoT devices as well as generate reports that can help security analyst extract indices of compromise and use them to improve their monitoring systems and harden their devices from possible attacks. The tool I am interested in developing is a sandbox for IoT malware, the sandbox will be called Diseker.

1.1 State of the art

IoT sandboxes are something newly created, and it is difficult to set up an environment that can emulate every IoT device in the market due to the high number of variety of devices and configuration, which resulted in that most of the sandboxes implemented are either specific to one type of IoT device or sandboxes that do not give the users enough details about the malware being executed. For some IoT devices is almost impossible to emulate because no emulation system supports their hardware yet.

Most of the sandboxes I found in my research were either emulators or automated emulators or were sandboxes, but they did not deliver enough information to the user to have a clear idea about the behaviour of the binary being analyser. Some of the most relevant sandboxes are:

- Detux: is a sandbox developed to do traffic analysis of Linux malware and capture IOC[1]s, it uses the QEMU hypervisor to emulate Linux (Debian) for various CPU architectures.
- LiSa (Linux Sandbox): an automated Linux malware analysis tool that supports various CPU architectures, it does perform static and dynamic analysis, also analyses network traffic to capture IOCs.
- IoT Sandbox to analyse IoT malware Zollard: the developers claim that it can be used to analyse and emulate any type of IoT malware, but it has been only used to analyse Zollard malware, it gave a great insight to the botnet and helped capture multiple IOCs.
- V-Sandbox: sandbox specialized on dynamic analysis for IoT Botnets, the main goal of this sandbox is creating an environment where the malware can run all its functionalities by providing a simulated C&C [2]server and capture system calls and system health.

And there are many more, each sandbox has its own advantages and drawbacks, in the case of Detux is very useful to grab network IOCs from the network capture and then complement that information from the simple static analysis performed, for LiSa it captures most of the behaviour intended in the design of our sandbox but creating and deploying new images and architectures can be a problem since it uses SystemTap to capture system calls which can be a problematic if the probing needs to be changed, IoT sandbox for Zollard seem to be intended only for one type of botnet and finally V-Sandbox which is the most

[1] IOC or Indexes of compromise is a term used in cybersecurity to refer to all the information that can identify a malicious behaviour in a device. The information could name of files, IP addresses accessed by a process or certain modifications to files.

[2] C&C, CnC, C2 are synonyms to command-and-control server, which are nodes in a botnet network in charge of management and control of the different infected hosts.

complete one in the list but it does emulate the behaviour of the C&C server which is something that can be easily avoidable by the malware developers in the future.

1.2 Objectives

The objectives of this project can be extended as much as needed, but I have narrowed them to the main points that follow the malware analysis methodology:

- Create a secure environment where malware can be run without jeopardizing the network or other devices.
- The environments must perform dynamic analysis
  - The environment must mimic an IoT device as much as possible.
  - The environment must be flexible with enough tools to allow the malware to exhibit all its functionalities.
  - The environment must log network traffic and system calls.
- The environment must perform static analysis.
- The environment must be able correlate the data obtained and help classify the malware.

1.3 Methodology

The methodology used is based on the common malware analysis steps by analysing the malware statically and getting all the details from the file and then grabbing the behaviour of the malware by executing it and capturing the malware interaction with the system. To simplify the idea, we can assume that the process will be following the diagram below.

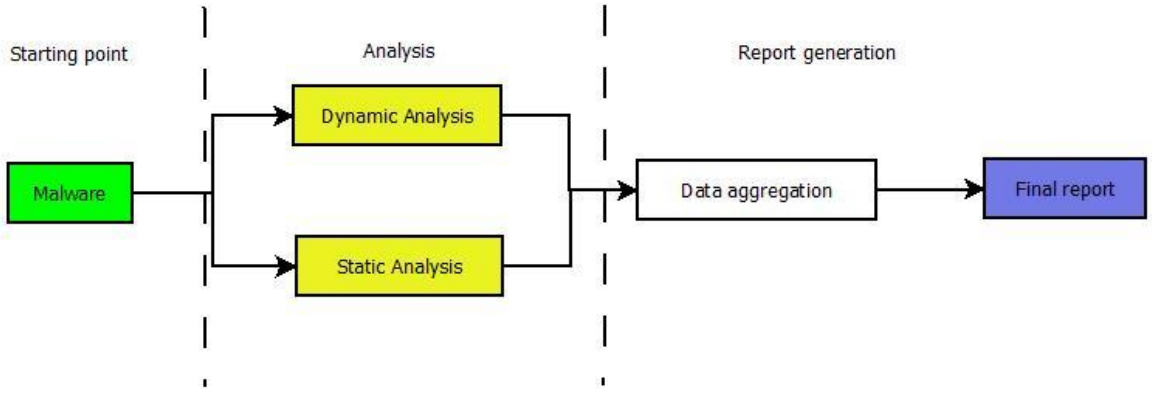


Figure 1 Data flow of the analysis process

The line of thought to achieve this project is simple learning about the different malware that can attack an IoT device, use the most generic IoT oriented emulation of a device, set up an automated environment with multiple logging tools to extract the needed information and finally create a representation format for the data extracted.

1.4 Expected tasks

The tasks expected in this project are flexible enough in case a change required and sufficient for this project to be complete:

1. Research and state of the art: perform a previous research and determent de tools available and all the previous implemented Sandboxes if any.
2. Search for malware samples.
3. Environment preparation
    a. Decide which CPU architectures to emulate
    b. Decide which emulator to use for the IoT devices
    c. Decide which application should the emulated IoT device contain
    d. Decide the logging tools
4. Implementation of the sandbox
5. Test of the malware samples
    a. Results study and future work
6. Results and summary
7. Theses writing
8. Video presentation
9. Theses defence

1.5 Time plan

For all the tasks listed above, the task dependency and the time required to finish the project is expected to be as follow:

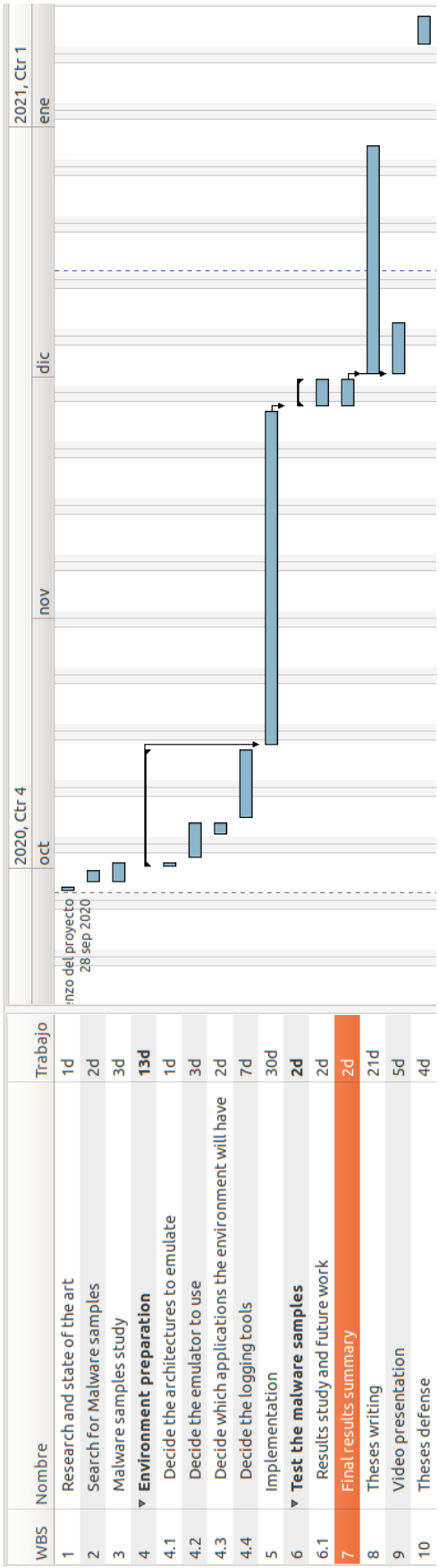


Figure 2 Timeline of the project

2 Project Requirements

This project can be extended in different directions, the main two ways you could orient this project are by data presentation and accessibility of the tool or by deeper analysis of the malware. In this implementation I focused more on system that can give a more comprehensive analysis of the malware rather than how to use such tool in the market. To fulfil such goal and the ones enumerated and the previous section, I have set the following requirements for the sandbox:

R1. The Sandbox should be able to emulate multiple architectures.

R2. Each architecture image should contain multiple audit and monitoring tools.

R3. The Sandbox should be as fast as possible.

R4. The sandbox emulation phase should be able to capture system calls made by the malware executed in it.

R6. The Sandbox emulation phase must monitor network and capture the traffic, most importantly the traffic of the protocols Telnet, ICMP, DNS, RDP, NTP, FTP, IRC and HTTP.

R7. Attach signatures to the traffic captured from the Sandbox (either by IDS or IPS) that was recognized as malicious.

R8. The Sandbox should be able to remove HTTPS security and transform it to HTTP either by using a proxy or other tool.

R9. The user most be allowed to choose between using emulated internet or connect the emulator to the real internet.

R10. The user should be able to choose between using a proxy to decode HTTPS traffic or not.

R11. Network logs and system calls logs need to be aggregated and correlated to create a complete vision on the behaviour of the malware.

R11. The sandbox should be able to conduct static analysis to the binaries, that being extracting strings, detecting binaries type, getting the malware hash, etc.

R12. The sandbox should be able to generate a report of the malware analysed based on the data extracted from it.

R13. All the analysed malware needs to be stored in a database so futures analyses of the same files can be extracted directly from the database instead of repeating the analysis unless the user wants to forcefully emulate the binary.

R14. The sandbox should be able to handle more than one simulation at a time.

3 Design

3.1 Challenges to consider

The diversity of the target is one of the main issues for a project based on emulation, since it needs to consider as much variety as possible. The challenges related to the diversity for the emulator do not rely only on the different architectures that the IoT malware targets, but also the type of operating system, the type of libraries being loaded, and other libraries linked to it. So, to continue with the design of the sandbox first we need to list the challenges to face.

3.1.1 Diversity

The diversity is not only in terms of architectures or hardware that needs to be consider when emulating a IoT system (which is an important part in this project), but it is also important to consider that all the IoT devices use Linux based systems that provides high variety and the executables generated, in this project we will be mainly focusing on the ELF type files, but event for those there are different options depending on the platform that uses the file:

- The ELF header is different for different platforms such us, Android, Linux or BSD.
- The executables can be dynamically linked or statically linked, which requires the OS to have certain interpreters to execute the file.
- ELF file loaders can change from one implementation to another.

And the diversity of possibilities will only grow, since the IoT world is just starting to evolve.

3.1.2 Library linking

Libraries can be linked to an executable using one of two techniques, static linking or dynamic linking, the first allows the executable to be self-contained since all the binaries required are compiled with the resulting binary. The second type of linking takes, which is dynamic linking, it uses a shared library in the target system to perform certain operations or system calls, the dynamic linking generates smaller executables than the statically linked ones.

The challenge in this situation is to prepare the system to consider the two options of execution, as well as the types of libraries being used, in the Linux operating systems there are three main libraries to consider uClibc, musl and Glibc. The other challenge is related to the statically linked binaries, as in [21] they detailed that the result binary does not rely on a higher liver API that performs the system call, such as *libc*, instead those binaries interact directly with the system calls which may lead for them to crash in runtime if the ABI of the kernel is different from the one expected.

3.2 Architecture

The sandbox architecture consists of two main steps that will help increase the speed of the results, the first step when analysing a file is checking if the file has been already analysed before and retrieve the results from database, if not execute the sandbox and update the database with the new values, the flow diagram of the process is as follow:

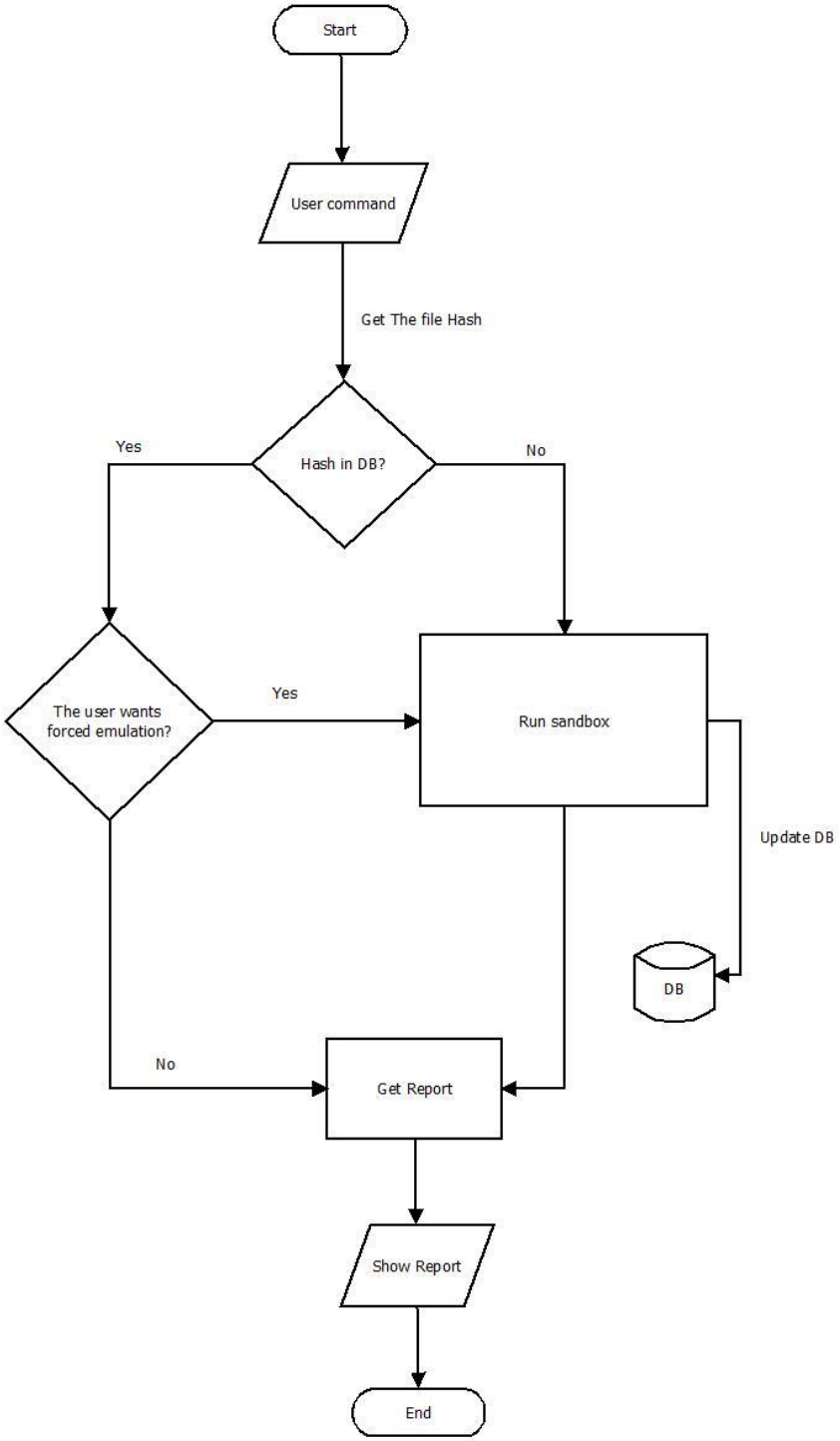


Figure 3 Flow diagram of the sandbox

The second step is performing the emulation of an IoT device and running the malware inside the sandbox. The sandbox can be splitted in two main sections based on the type of

analysis performed over a binary file, which are the static and dynamic analysis. For the static analysis, the sandbox performs simple operations over the file and sends the data to report generator, while the dynamic analysis is responsible of executing the binary, monitor its execution and retrieve the logs, the simplified vision of the architecture looks like the following:

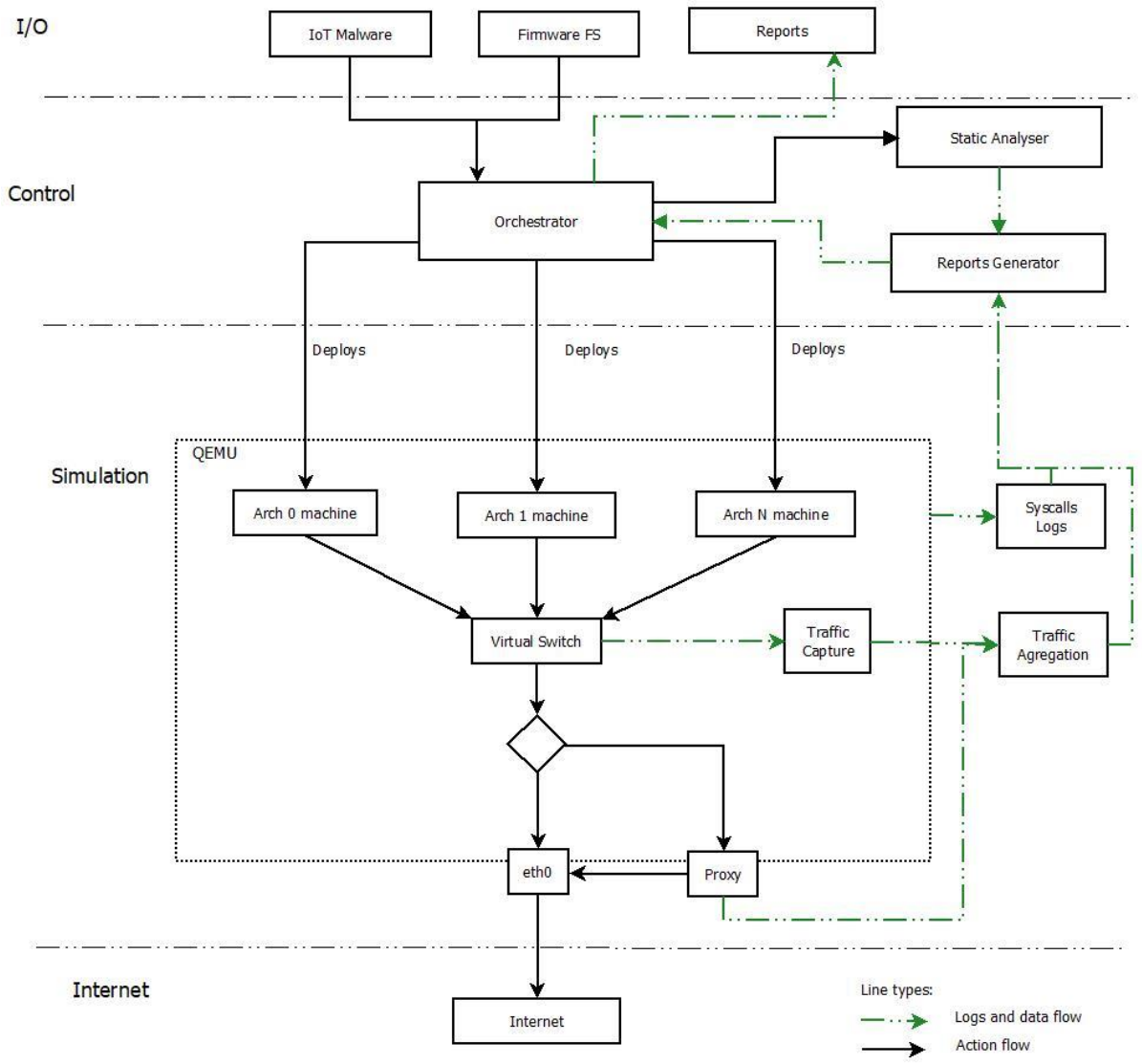


Figure 4 Architecture and interaction of the components of the static and dynamic analysers

As you can see, there are three main parts *I/O*, *Control*, *Simulation*, *Internet*, the first part is simple is the input/output section where the user can input the commands desire,d the malware path and the path to the filesystem that will be emulated (this parameter is optional) and after processing the users request the sandbox returns a report. The second part is *Control* that is responsible of managing the different emulations and making sure that the logs have been extracted, the report generator has been called and as well as the static analyser, it also makes sure that all the processes related to a job have finished and been compiled into a

report, *Control* is also responsible of giving results to the user. The third part is *Simulation*, this part contains all the tools, scripts and configuration needed so the emulation works. Finally, *Internet*¸ this part makes sure that the sandbox has access to the internet either emulated or real.

The structure of the code that is going to manage the execution and data flow is represented by class diagram at Figure (5). The classes *Orchestrator* and *ReportGenerator* belong to the components in the *Control* section in Figure (4), which are responsible of managing the execution and obtaining the logs and establishing connection with the database by using *MongoConnection* singleton; each *ReportGenerator* object has one or multiple parser that are made to parser logs that contain system calls or network traffic. There is also the *Unit* class which is the object that the *Orchestrator* class uses to manage the analysis process by calling the abstract class *Analyser*, which by polymorphism can execute either the static or dynamic analyser by using *StaticAnalyser* or *DynamicAnalyser* respectively. Due to the nature of the dynamic analysis, the class *DynamicAnalyser* will oversee and control the emulation process by communicating with the virtual environment, sending scripts and malicious files, and retrieving raw logs. All the classes involved with *Unit* are in the *Simulation* section in Figure (4).

ReportGenerator is an associative class that can only be instantiated when the relation between *Orchestrator* and *Unit* is fulfilled, this way the traceability of the reports and logs is insured. The other important thing is that the singleton patter on *MongoConnection* insures that along the entire program there is only one connection established in the database, which allows concurrency and avoid corrupting data.

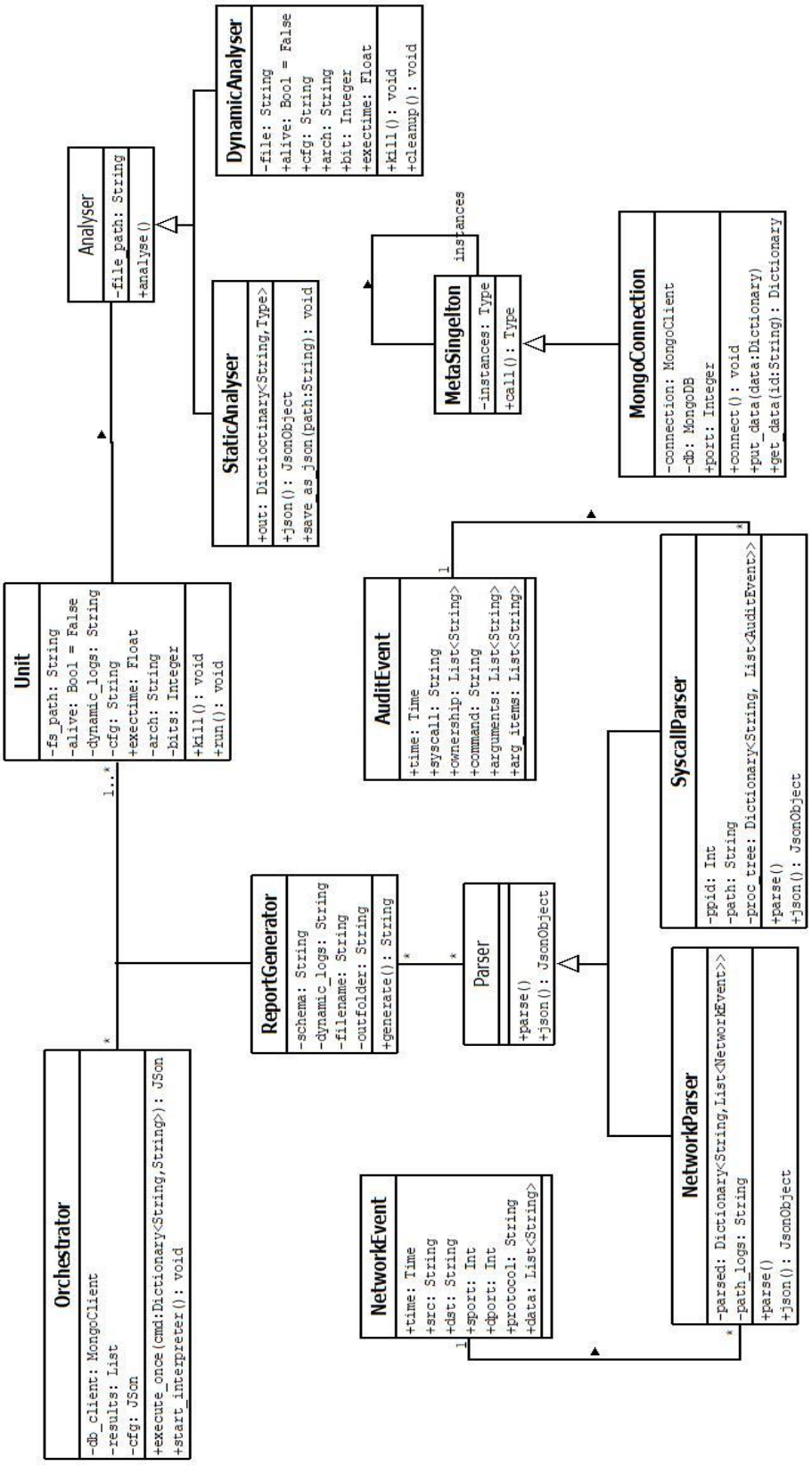


Figure 5 Class diagram for the static and dynamic analysers

3.3 Data structures

All the data obtained from the emulation needs to be parsed and grouped in a way that maintains the timeline of events. The logs in this case will be grouped in two types, the system call (syscall) logs and network logs; the logs are split this way because the parsing of each type of log is different.

The system call logs are very important since they give detailed information about what the malware was trying to execute in the system, to ensure that all events related to the execution of the malware are being extracted we need to get all the logs and assign every syscall event to the process ID (PID) that executed it and reference the parent process ID (PPID) to maintain the traceability. It also important to have fast access to the individual PIDs in the data structure in case we need to extract the syscalls of o single PID. All the required information of a syscall events is stored in the class *AuditEvent* in Figure (5), to have fast access to the different PID a hash table will be used that uses the process PID as key and a tuple of PPID and a list of syscall events sorted by time. This data structure is not readable enough and will not help analyse the logs if it is required to be done manually, it is also difficult to store and recover or send to other process to visualize the data such us frontends or similar tools, to achieve that a transformation to JSON format following the scheme is used:

```
{
  "PID": {
        EXEV: "Command"
        SYSCALLS: [
                  { time: "time"
                    name: "name"
                    ownership: [uid, suid, euid]
                    arg: [a0, a1, a2, a3]
                    ítems: […]},
                        .
                        .
                        .
                  ],
        children: [
            {
               "PID": {
                        EXEV: "Command"
                        SYSCALLS: [
                                 { time: "time"
                                   name: "name"
                                    ownership: [uid, suid, euid]
                                    arg: [a0, a1, a2, a3]
                                    items: […]},
                                    .
                                    .
                                    .
                                    ],
                       children:[…]
            },
            .
            .
            .]
}
```

Code 1 System call events Json schema

The scheme helps establish the execution tree as well as contain all the syscalls per process.

In the case of network logs, same hash table is used but using the name of protocol as key and a list of *NetworkEvents* sorted by time, this way is possible retrieve the network events for each protocol individually and fast. The *NetworkEvent* class encapsulates all the relevant information we need from the network logs, such us time, source address, destination address, source port, destination port, protocol and payload or data. Network logs also have a Json format that can be used, the scheme of the Json to be used for the network logs is the following:

```
{
  PROTOCOL: [
             {
                Time: time,
                Src: source_address,
                Dst: destination_address,
                Sport: source_port,
                Dport: destination_port,
                Prot: protocol,
                Data: payload
              },
                 .
                 .
                 .
             ]

}
```

Code 2 Network events Json schema

3.4 Database design

The database used in the project is NoSql because we do not have intricate relations between entities, and we need fast access and relation between the ones to use. NoSql databases also provides great flexibility between relations which can help in developing new ways of associating the different malware by using graphs or other structures.

The database will contain the hash or signature value of the file analysed, the file name, a scan count that must be increased every time a forced scan is conducted on the file, it contains tags that classifies the files analysed, the path to the raw or json format logs and finally the path to the generated report.

The scheme of the database is as follows:

DB name: analysed_malware

```
{
  "_id": objectID,
  "filename": String,
  "scan_count": Int,
  "tags":[
       {
         "tag_id":objectID,
         "name": String,
         "description": Text
       }
     ],
   "report_path": String,
   "logs_path": String
}
```

Code 3 Database schema

3.5 Sandbox components

The sandbox requires multiple components and tools to work properly and fast; most of tools to be used are monitoring systems that will allow investigate the behaviour of the binary being executed and know what exactly it does.

The main components of the sandbox are:

- Virtual machine that will host all the project.
- Virtual IoT devices units that will run the malware.
- Database that will contain the repots and logs of the analysed malware.
- Scripts to automatize the execution.
- Custom operating system for the emulated IoT devices with all the tools required to monitor the execution of the malware.

3.5.1 Virtual machine

The virtual machine will work as a host of the sandbox; we are using a VM because we need to change the network configuration, the firewall rules and other elements of the host operating system so the sandbox can work properly. In the future the sandbox will be hosted in a container that has all the configuration needed and the tools as well, but in the meantime, we are using a virtual machine.

The VM host is a Linux based operating system, this will allow create the virtual switch in *Simulation* section of Figure (4) by using TAP interfaces and a bridge, this operating system will also help perform some pre-parsing such us splitting the network PCAP files in multiple files and filtering by protocol, this way is easier for the parser to extract the information. The VM will also host the proxy server that is going to be used to transform HTTPS and FTPS traffic to HTTP and FTP traffic, as well as the simulated internet network (SIneT). The advantage of this configuration is that the emulated devices will access the proxy server and the SIneT using the bridge interface used by the virtual switch as in Figure (4). The only disadvantage is that the parser needs to be aware of the previous traffic of each IoT emulator so requests from other emulators working simultaneously can be ignored.

Finally, the VM will host the database of the information about all previous analysed malware, as well as the logs and reports.

3.5.2 Virtual IoT devices

Qemu will be used to emulate the hardware or parts of the machines that host the IoT device, this emulator is easy to use, there is extensive documentation and tutorials on how to use it to emulate multiple devices and it is possible to emulate all the devices this project will cover.

The main architectures to be emulated through Qemu are:

- ARM
- aarch64
- I386
- PowerPc

For this emulation to work, we need a Linux based operating system (OS) compiled for each architecture, the diversity of files format for each architecture will also be considered by compiling the operating systems for the two my C libraries, glibc and uClibc. The OS will contain multiple monitoring tools that will allow set up the environment for the analysis. The main tools to consider in the section are the following:

- Linux Kernel Audit Subsystem or Auditd: is going to be the tool used to monitor system calls, this subsystem provides a secure logging framework that allows capturing and recording security relevant events, this is achieved by components which generates audit records based on system activity, the logging is performed by a userspace daemon which logs these records to a local file or remote aggregation server. The system activity to monitor es defined by rules that the user needs to design to get comprehensive information from the system activities as well as reduce the overheat of the logging.

- For network monitoring TCPdump will be used, it is a simple and powerful tool that captures all the traffic being sent and received through a network interface and it stores result in a PCAP file or prints it in the standard output, in this case it will be saving the captured data into a PCAP file.

3.5.3 Database

The database is going to be used in this project is MongoDB, due to the extensive community and documentation in the web, as well as examples that help develop the application needed. One of the main advantages in the fact that this database uses JSON syntax to create database, establish schemes and save or retrieve data, which is suitable when interacting with the database because most of the programming languages support Json.

Other interesting features of this database is the fact that it can support saving and loading big files without compromising their integrity as well as the possibility of deploying the database to be used as a service easily.

4 Implementation

The implementation of this project is extensive and required multiple tools to achieve the objectives set previously, to simplify the details of the implementation I will explain it in the following key points:

- Main Script (Diseker)
- Orchestration
- Static Analyser
- Dynamic Analyser
- Parsing
- Report generation
- Network configuration
- Database configuration

This project is implemented in Python3, due to the simplicity of implementation and the tools available. It would be interesting to implement this project in other fast languages in the future to see the impact of it on the execution speed.

4.1 Main Script (Diseker)

The main script is the one that the users will call to start the sandbox, this main script will manage the two operation modes of the sandbox, single execution, or interactive mode. The single execution is when the user introduces the command to execute the sandbox only once for one file and wait for the sandbox to return a result, while the interactive mode allows the user to send files to emulate in parallel if there is a free thread, this process is managed by the Orchestrator.

The main script also parses the parameters that the sandbox accepts, which are:

- -i : parameter to activate the interactive mode.
- --in : file to be executed
- --out: folder where the report will be saved
- -f, --force: forces the emulation of a file even if that file was found in the database.
- -c, --config: configuration file to be used.
- -a, --arch: CPU architecture that the binary runs on, at this moment only one of these options can be chosen, aarch64, ppc64, x86_64, x86 or arm.
- -b, --bits: bits of the file, either 32 or 64.
- -t, --time: execution time of the emulation, the default value is 120 seconds
- -e : exit, only works in interactive mode.

To run the interactive console, it is required to run the command with the options -i and -c or –config which are the main components that will not change along the interactive session.

4.2 Orchestration

The orchestration is supported by the class *Orchestrator* which manages the execution of the different emulations and controls the results of each one. It also runs the backend of the interactive console of the sandbox.

For a single run without the interactive session, the orchestrator grabs the command from the main script and starts the sandbox process and only returns when there is a result or an error, but for the interactive console it runs an event loop and waits for users input using the options above. To achieve this behaviour the class *Orchestrator* implements the interactive functionality using threads, where a main thread reads the commands introduced by the user in the interactive session and creates threads for each command, no more than a certain parallel emulations can be run at the same time, this limit is controlled by the *Job thread reference queue*, all the commands created by the user are added to a queue that is consumed by the *Jobs executer thread* as in Figure (6), once the threads that run a job finish they store the results in a queue that is consumed by the interactive session that shows the results to the user.

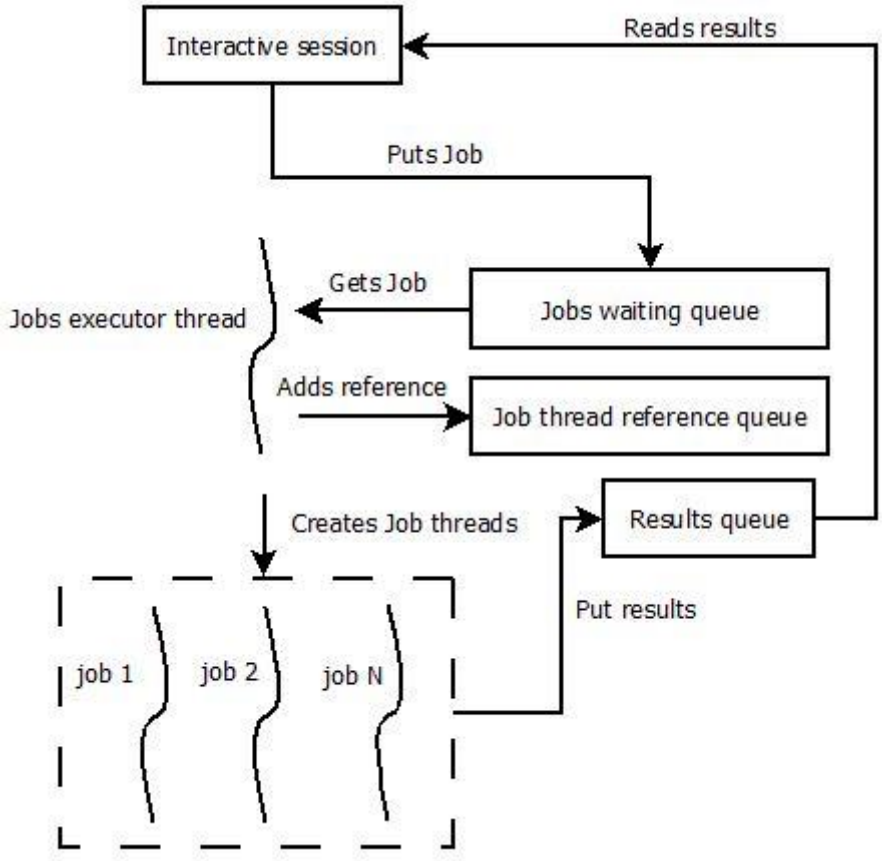


Figure 6 Interactive session behaviour diagram

This behaviour insures the interactive session and the delivery of the results in a consistent manner.

The orchestration also makes sure that the connection to the database has been established correctly and that the configuration file is the same for each emulation attempt. The configuration file contains all the information about the CPU architectures supported by the sandbox, the information that holds is the OS path (kernel and filesystem), location of configuration files to start a Qemu instance correctly and the commands to run per OS supported. The configuration file also contains the path to the dynamic analyser logs and report schema or report template. The file itself is in YAML format, due to the ease of manipulation, is human readable and the speed of loading it, and example of configuration files is:

```
ppcp:
  64:
    path: /path/to/ppc64_pseries/
    kernel: /path/to/ppc64_pseries/kernel
    image: /path/to/ppc64_pseries/filesystem.ext2
    cmd: qemu-system-ppc and the rest of the command

tmp: /path/to/logs
schema_path: /path/to/report/schema/
```

It is important to maintain the same configuration file for an entire emulation session with the interactive mode because the information of that file will establish how the network is going to be used and how many concurrent tasks can be run at the same time.

4.3 Static analyser

The static analyser performs in depth data extraction from the file to get all the information needed to identify the signature of the file, indices of compromise related to it or understand if the executable has encrypted data or not. In the sandbox the static analyser performs the following operations on an executable:

- Extracts all information from the header, such as the type of file, the CPU architecture and the bits used. It also determents the size of the file and hash or signature of the file.
- Extracts all the sections of the file, the size of each section and where does the section start in the file. This can help give an idea about how many hardcoded data is stored.
- Extracts all function names in the file, only functions that has a non-generic or autogenerated name, it also extracts the size and the start address of each function. This information used with syscalls logs can help the analyst guess with high certainty parts of the behaviour of the executable.
- Extracts all the strings from the file, the strings extracted are bigger than 10 characters, it also checks for special strings that contain URLs, Domains, IPv4 and IPv6, those strings can be used as indices of compromise (IOC) for future detections, it will also help link the file with known threats.

- The static analyser also measures the entropy of the file for each 1024 bytes of data, this measure determents the randomness of information which helps put in context all the information above, higher randomness implies high number of mixed strings and functions with unconventional names, which means that the section is either compressed, obfuscated or encrypted. For the entropy, the analyser uses Shannon entropy, which is a continuous function based on the prediction of future appearance of an element of a set in a subset. Shannon entropy formula is:

Equation 1 Shannon entropy

$$\mathrm{H}(X) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log \mathrm{P}(x_i)$$

To perform the analyses above I used Radare2, which is a complete framework for reverse-engineering and analysing binaries, it supports multiple CPU architectures, including the ones used in this project. To use Radare2 in python r2pipe was required, which is a library implemented by Radare2 that emulates the interactive terminal of Radare2 using python functions to use commands in Radare2.

4.4 Dynamic analyser

The dynamic analyser is much more complex than the static one, because the process requires preparing the environment, starting the emulated environment, and executing the binary, and finally extracting the data to a known location. For that to work multiple key components had to be built in advanced such us the OS images and automation scripts.

4.4.1 Preparing the environment

This step consists of detailed explanation of the steps taken to achieve the OS and how the tools were set to log the execution of the binary inside the emulation.

4.4.1.1 OS Building

The operating systems used in this sandbox are Linux based, compiled for different architectures but containing the same tools. To build these images I used Buildroot, which is a tool that applies patches and simplifies the process of building Linux environment for embedded systems, it also uses cross-compilation hence the different architectures supported.

The images built are based upon Linux and they use Busybox, which is a common software suit between IoT devices since it contains a compact version of all the common tools in a Linux environment, it has been used in Android devices and still being used in most out Routers in the market.

To configure an OS, I started with a predefined configuration of an architecture I wanted to use, let's take as an example the X86 architecture, first we use that architecture as follow in the folder of Buildroot:

```
$ make qemu_x86_defconfig
```

That will populate the configuration file of Buildroot with all the information needed to build the image with default configuration. Before starting to edit anything, we need to set the max size of the file system to be used, this file system size needs to be slightly bigger than the size of the tools that contains, this way we can add more documents and files without crashing the system, to set the size of the filesystem we first need to start the configuration menu:

```
$ make menuconfig
```

Which will start the following menu:

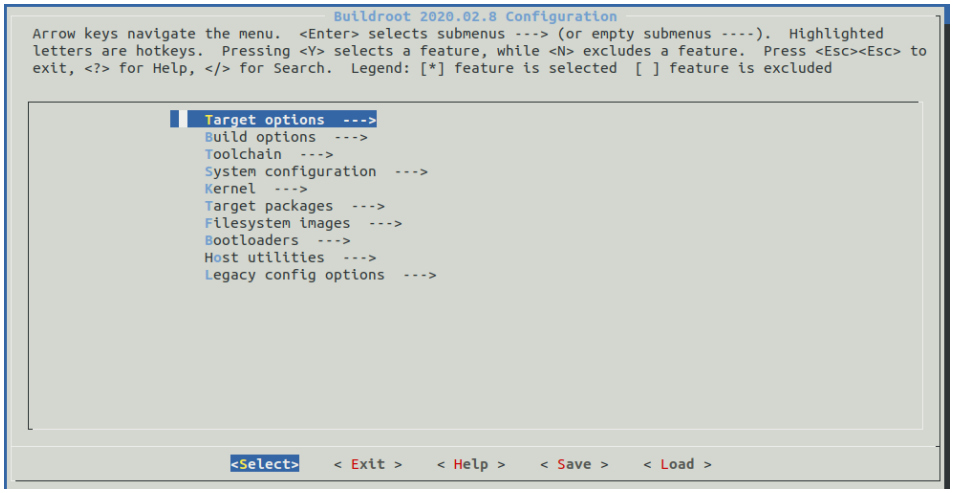


Figure 7 Buildroot menu

In the option “Filesystem images” we put the following configuration, selection the the “ext” filesystem with the exact size of 1024M bytes.

```
[ ] cpio the root filesystem (for use as an initial RAM filesystem)
[ ] cramfs root filesystem
[*] ext2/3/4 root filesystem
      ext2/3/4 variant (ext4)  --->
()    filesystem label
(1024M) exact size
(0)   exact number of inodes (leave at 0 for auto calculation)
(5)   reserved blocks percentage
(-O ^64bit) additional mke2fs options
```

Figure 8 Filesystem configuration in Buildroot

Then we need to set the toolchain[3] to use to compile all the tools and packages, for that you need to select the “Toolchain” option from Figure (7), then change the tool chain type to “External Toolchains”, this will download the latest tool chain for the architecture you

[3] A suite of tools used in a serial manner, used for developing software application and operating systems

want to compile for with all the tool needed, you can set from where you want to download the toolchain from in the option “Toolchain Origin”, finally you need to verify what “Bootlin toolchain variant” to use, this option gives the user the possibility to compile the system with Glibc, uClibc or musl libraries, in this case we will compile the same architecture with Glibc and uClibc.

```
        Toolchain type (External toolchain)  --->
        *** Toolchain External Options ***
        Toolchain (Bootlin toolchains)  --->
        Toolchain origin (Toolchain to be downloaded and installed)  --->
        Bootlin toolchain variant (x86-i686 uclibc stable 2020.08-1)  --->
    [ ] Copy gdb server to the Target
        *** Host GDB Options ***
    [ ] Build cross gdb for the host
        *** Toolchain Generic Options ***
    ()  Extra toolchain libraries to be copied to target
    ()  Target Optimizations
    ()  Target linker options
    [ ] Register toolchain within Eclipse Buildroot plug-in
```

Figure 9 Toolchain configuration

The next step is to set the system configuration with the host name, password, banners, as well as the device tables to use to create the drivers in the Linux, the file system, the Init system, passwords and so one. The configuration chosen is the following:

```
    Root FS skeleton (default target skeleton)  --->
(Diseker) System hostname
(Welcome to Diseker) System banner
    Passwords encoding (sha-256)  --->
    Init system (BusyBox)  --->
    /dev management (Dynamic using devtmpfs only)  --->
(system/device_table.txt) Path to the permission tables
[ ] support extended attributes in device tables
[*] Use symlinks to /usr for /bin, /sbin and /lib
[*] Enable root login with password
(veryCoolPassword) Root password
    /bin/sh (busybox' default shell)  --->
[*] Run a getty (login prompt) after boot  --->
[*] remount root filesystem read-write during boot
(eth0) Network interface to configure through DHCP
(/bin:/sbin:/usr/bin:/usr/sbin) Set the system's default PATH
[*] Purge unwanted locales
(C en_US) Locales to keep
[ ] Enable Native Language Support (NLS)
[ ] Install timezone info
()  Path to the users tables
()  Root filesystem overlay directories
(board/qemu/x86/post-build.sh) Custom scripts to run before creating filesystem images
()  Custom scripts to run inside the fakeroot environment
(board/qemu/post-image.sh) Custom scripts to run after creating filesystem images
($(BR2_DEFCONFIG)) Extra arguments passed to custom scripts
```

Figure 10 System Configuration

The next option to configure and generate the operating system is “Target Packages” at Figure (7), this option allows the user to choose which tools, libraries and controllers to compile so they can be used one the system and running. It is important to note that not all the tools will be available for all the architectures, since some tools may not have been written to be compatible with every CPU that exists, which means that the tools that are going to be used are as broad as possible as well as compliant with the objectives of the project.

Buildroot classifies the tools in the following types:

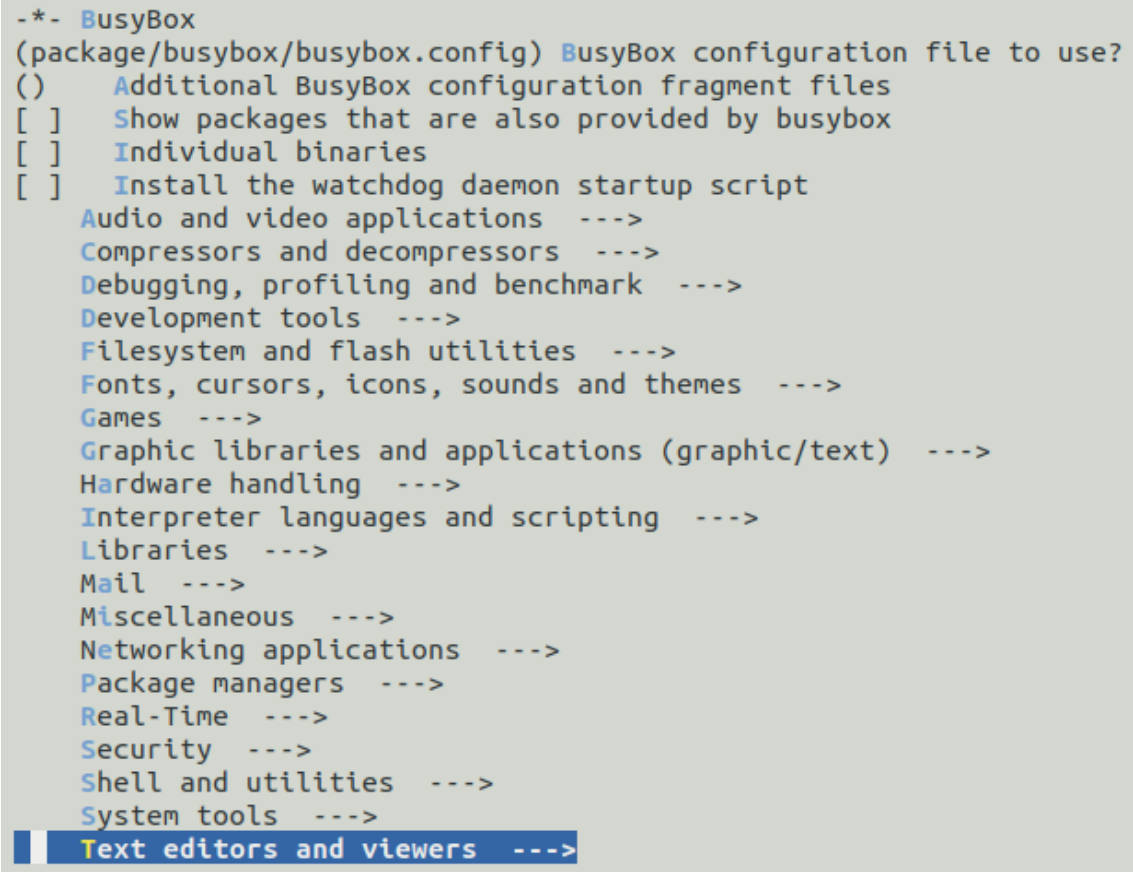


Figure 11 Buildroot target packages

As you can see at Figure (11), all the tools going to be installed in the target are going to be based on Busybox, which the toolbox that most of the IoT devices use, the same toolbox was chosen to be the Init system at Figure (10). The busybox of the system will contain the following tools based on the name provided by Buildroot at Figure (11).

- Compressors and Decompressors: The added compressors and decompressors are broadly used, and they could be required by some malware to extract downloaded packages, this is just a speculation is not based on any other findings
  - Lzip
  - Zip
- Debugging, profiling and benchmark: The following tools are mainly used for deeper analyses of the malware in case the information extracted is not enough. The first one is a debugger, and the second ones are program tracers[4] to extract the function calls made by the process.
  - gdb / gdbserver
  - ltrace
  - strace

[4] A tracer is a debugging program that is used to trace function calls made by a process, the trace is different for statically or dynamically linked binaries, which means that different tools are used, *ltrace* is used to debug dynamically linked binaries and *strace* is used for the statically linked ones.

- Development tools: this are installed just in case a malware have them a dependency due the vast use of them.
    - Flex
    - Libtool
- Interpreter languages and scripting: multiple malwares use the following scripting languages to modify the system, as well as these scripting languages are commonly used in Linux based systems as well.
    - Luajit
    - Python
- Network applications: The network applications will allow the proper configuration and manipulation of the stat of the network interfaces, the also contain logging tools and other interesting ones that the malware may need to contact with a command-and-control server or just to establish a secure connection.
    - Bind
    - Dhcpcd
    - Dnsmasq
    - Dropbear
    - Enthtool
    - Ifupdown scripts
    - Iproute2
    - Iptables
    - Lftp
    - Pppd
    - Rpcbind
    - Tcpdump
    - Traceroute
    - Wpa_supplicant
    - Wpan-tools
- Shell and utilities: these tools will be used for troubleshooting and testing the malware in different contexts.
    - File
    - Sudo
- System tools: system tools will contain the auditing tools and complementary tools that will allow good management of services running in the background.
    - Audit
    - Daemon
- Text editors and viewers
    - nano

Busybox toolbox contains other tools as well that are more common in Linux systems and they are not listed above. Once all the options are set correctly, the operating system needs to be built, so first need to save the configuration and get out of the configuration menu and then type:

```
$make -j4
```

The command starts the building process of the operating system with 4 threads to increase speed by using concurrency. In this phase Buildroot will download all the packages needed and compile the tools and the kernel using the Toolchain that was set before.

The process above will have to be performed for all the other CPU architectures the same way. This will generate all the images and operating systems needed for the sandbox to cover the diversity of target.

4.4.1.2 Setting up the logging tools

You may have noticed that one of the required tools in the OS is *Audit*, which is a tool developed by Redhat that is used to monitor the system behaviour based on a set of rules, those rules can be set to capture system calls, reading, writing and access to files and folders, and network traffic from the kernel space. In this project the set of rules that are going to be used are designed to detect the execution of different commands, the creation of chilled processes, reading and writing files, installation and removal of kernel modules and attempts of connection through network traffic. The set of rules is:

```
# Processes rules
-a always,exit -F arch=b32 -S kill -S execve -S fork -S getpid -S clone
-S execveat -F key=precess-interaction
-a always,exit -F arch=b64 -S kill -S execve -S fork -S getpid -S clone
-S execveat -F key=precess-interaction


# Pipe stuff
-a always,exit -F arch=b32 -S pipe -S tee -F key=pipe-creation
-a always,exit -F arch=b64 -S pipe -S tee -F key=pipe-creation


# Files rules
-a always,exit -F arch=b32 -S open -S creat -S link -S unlink -S symlink
-S mknod -S openat -S linkat -S unlinkat -S mknodat -F key=file-creation
-a always,exit -F arch=b32 -S rename -S renameat -F key=file-move
-a always,exit -F arch=b32 -S dup -F key=file-pipe


-a always,exit -F arch=b64 -S open -S creat -S link -S unlink -S symlink
-S mknod -S openat -S linkat -S unlinkat -S mknodat -F key=file-creation
-a always,exit -F arch=b64 -S rename -S renameat -F key=file-move
-a always,exit -F arch=b64 -S dup -F key=file-pipe




# Kernel modification rules
# The following syscalls does not exist in b32 kexec_load and
kexec_file_load
-a always,exit -F arch=b32 -S init_module -S delete_module -S add_key -
S request_key -S finit_module -F key=kernel-module
-a always,exit -F arch=b64 -S init_module -S delete_module -S add_key -
S request_key -S kexec_load -S finit_module -S kexec_file_load -F
key=kernel-module
```

```
#Events and signals
-a always,exit -F arch=b32 -S signalfd -S eventfd -F key=events-fd
-a always,exit -F arch=b64 -S signalfd -S eventfd -F key=events-fd

# Networking and sockets
-a always,exit -F arch=b32 -S socket -F a0=2 -F key=connections
-a always,exit -F arch=b64 -S socket -F a0=2 -F key=connections
-a always,exit -F arch=b32 -S socket -F a0=10 -F key=connections
-a always,exit -F arch=b64 -S socket -F a0=10 -F key=connections

# accept syscall is not in arch 32
-a always,exit -F arch=b32 -S setdomainname -S connect -F key=connections
-a always,exit -F arch=b64 -S accept -S setdomainname -S connect -F
key=connections
```

To apply these rules when in the environment once the machine is running, I use the following script with name "*apply_rule.sh"*.

```
#!/bin/bash
input="test-rules.rules"
while IFS= read line
do
      if [[ $line != "#"* && $line != "" ]]
      then
            auditctl $line
      fi
done <"$input"
```

Code 4 Script used to apply Audit rules to the operating system

The script above reads the rules from a file and applies them one by one, this is better than having the rules load through audit automatically because this way there is no need to implement different rules for different architectures *auditctl*[5] will ignore any rule that is not applicable for the system running instead of crashing, which is what happens when loading the rules by the *audit* daemon.

[5] Tool provided by Auditd to control the behaviour, get the status and add or remove rules from the audit system

Once the rules are set, the script *stater.sh* needs to be executed which will run the binary and start the logging of the network traffic, filter and extract the data from the audit logs as well as creating the directory tree of the logs to be extracted. This script also saves the state of the machine before executing and after executing the malicious code by saving the processes executing in it, this script can extract all suspicious processes dispatched after the execution of the malicious code and then get all the system calls performed by those processes. A process is suspicious if its PID is bigger than *starter.sh* and is not a child of neither *starter.sh* nor the malicious code.

```
#! /bin/sh
BASE=$(pwd)
NAME=$1
SYSCALL_PATH=logs/syscalls
NET_PATH=logs/network
mkdir -p $NET_PATH
mkdir -p $SYSCALL_PATH
PRE_EXEC=logs/pre.state
POST_EXEC=logs/post.state
ps -o ppid,pid,user,comm,vsz,stat | awk '$4 != 0 && $5 !~ "Z"' > $PRE_EXEC
cat $PRE_EXEC | grep "starter.sh" -A 100 | grep -v "ps" | grep -v "grep"
> logs/dummy.state
tcpdump -nn -s0 -w $NET_PATH/net_logs.pcap &
TCPDUMP_PID="$!"
./$NAME >> logs/out.txt 2>> logs/out.txt &
APP_PID="$!"
echo "The pid of the process is $APP_PID"
sleep 1m
echo "First Analysis"
ausearch -i -p "$APP_PID" -m SYSCALL > $SYSCALL_PATH/$APP_PID.logs
./follower.sh $APP_PID $BASE/$SYSCALL_PATH
kill -15 $TCPDUMP_PID
# Get the IRC traffic
tcpdump -nn -vA -r $NET_PATH/net_logs.pcap "port 6667" -w
$NET_PATH/irc_net.pcap
#Get NTP traffic
tcpdump -nn -vA -r $NET_PATH/net_logs.pcap port 123 -w
$NET_PATH/ntp_net.pcap
#Get ICMP traffic
tcpdump -nn -r $NET_PATH/net_logs.pcap icmp -w $NET_PATH/icmp_net.pcap
#Get DNS traffic
tcpdump -nn -r $NET_PATH/net_logs.pcap "port 53" -w
$NET_PATH/dns_net.pcap
#Get Telnet traffic
tcpdump -nn -r $NET_PATH/net_logs.pcap "port 23" -w
$NET_PATH/telnet_net.pcap
#Get HTTP/s traffic
tcpdump -nn -r $NET_PATH/net_logs.pcap port 80 or port 443 -w
$NET_PATH/http_net.pcap
ps -o ppid,pid,user,comm,vsz,stat | awk '$4 != 0 && $5 !~ "Z"' >
$POST_EXEC
```

Code 5 Script used to execute the malicious file, start network monitoring and search audit logs

The script *starter.sh* also splits the logged network file in "net_logs.pcap" into the different protocols supported which will help afterwards with parsing. The script *starter.sh* also runs another script called *follower.sh* which is the one that extracts the logs of the children processes logged in the audit logs recursively and save the results in the logs directory tree, the script in question is:

```
#!/bin/bash
if [ "$1" == "" ]
then
      exit 0
fi
for pid in $(ausearch --ppid $1 -m SYSCALL | grep SYSCALL | cut -d ' '
-f 14 | cut -d '=' -f 2 | sort | uniq -d | tr '\r\n' ' ' )
do
      ./follower.sh $pid $2
      ausearch -i -p "$pid" -m SYSCALL < /var/log/audit/audit.log >
$2/"$pid"."$1".logs
done
```

Code 6 Recursive script used to get logs from child processes

The directory tree that holds the logs is basically the following:

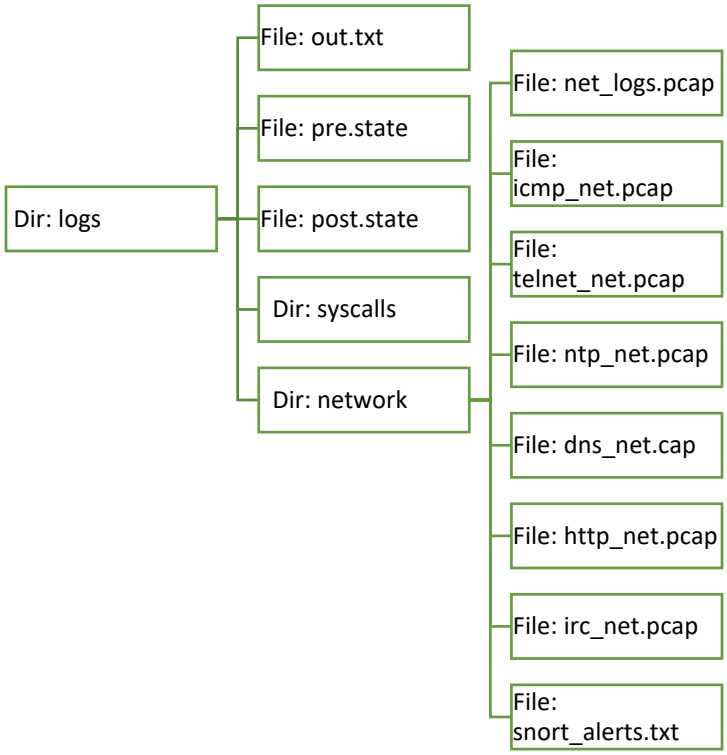


Figure 12 Directory tree of the logs

The “logs” folder will contain the file with the output (standard and error output) of the malware as well as the before and after states of the machine in files “pre.state” and “post.state” respectively; the “syscalls” folder will contain audit log files with PID of the processes audited as names and the sequence of PID.PPID1.PPID2.PPIDn.logs if it’s a chilled process. While in “network” folder will always contain the same files.

The scripts above are copied into the filesystem of each OS compiled using e2tools with execution permissions, that will allow to run them directly once the emulation starts, the command used for that is:

```
$e2cp file_to_copy filesystem.ext2:/path/to/folder/file
```

One of the logging tools used as well is Snort NIDS/NIPS (Network intrusion detection system/ Network intrusion prevention system), which will analyse the captured traffic and detected any possible alerts triggering, this tool is going to be used mainly to detected scanning activities and common attacks to known vulnerabilities in the network. As you may have already known, Snort is an engine that analyses traffic and uses a set of rules that describe a certain malicious or noncompliant behaviour, a rule contains a header and a body, the rule header contains the information about the packets that defines its type, the source and destination as well as what action to take once the body of the rule matches a packet or behaviour. There are 3 default actions in Snort, *alert* (generate an alert and log the packet), *log* (log the packet without alerting) and *pass* (ignore the packet).

This tool is used due to the availability of rule sets provided by the community and ease of implementation of those. In this case we have used the community rules form Snort website plus some minor rules to detect different types of stealthy network scanning such as Xmas, Scan Null, Scan fin, etc and general port scan, sample of the rules for scanning are:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN XMAS";
flow:stateless; flags:SRAFPU,12; reference:arachnids,144; classtype:attempted-recon;
sid:625; rev:7;)

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN nmap XMAS";
flow:stateless; flags:FPU,12; reference:arachnids,30; classtype:attempted-recon;
sid:1228; rev:7;)

alert tcp any any -> $HOME_NET any (msg: “TCP Port Scanning”;
detection_filter:trackby_src, count 400, seconds 60; sid:100006; rev:2;)
```

Once the network traffic has been extracted from the emulated environment, it will be run through Snort to see if any alert triggers and then we save the alerts for posterior treatment.

4.4.2 Starting the emulation

Once the type of architecture has been determined either by the user or the static analyser and the system confirms that it can be emulated, then the dynamic analyser copies the filesystem of the OS to be used into the destination folder of the logs, which has been specified in the configuration file, then it copies the executable into the root folder of the filesystem of the OS using e2tools, the copied folder will have the name *tobe_executed*, next

it loads the audit rules by executing the script *apply_rules.sh*, and executes the *starter.sh* file with the executable as a parameter and finally waits as much as the execution time allows in seconds. Once the timeout is reached, the copied filesystem is removed after retrieving the logs and parsing process starts.

The interaction between Python and Qemu is done through Pexpect, which is an automation tool for interactive applications, it is used to introduce the password for the user once the image is loaded and it is also used for executing all the commands and files listed in the other sections inside the Qemu emulation.

4.4.3 Extracting the logs

Once the emulation has finished a script is executed to get all the logs from the filesystem used by Qemu before removing it, the script is

```
#!/bin/bash
if [ $# -lt 2 ]
then
  echo "Usage: extract.sh filesystem dest_folder"
  exit 1
fi
SYSCALL=logs/syscalls
NET=logs/network
mkdir -p $2/$SYSCALL
mkdir -p $2/$NET
syscall_logs=$(e2ls $1:/root/$SYSCALL)
for fl in $syscall_logs
do
  e2cp $1:/root/$SYSCALL/$fl $2/$SYSCALL
done
net_logs=$(e2ls $1:/root/$NET)

for fl in $net_logs
do
  e2cp $1:/root/$NET/$fl $2/$NET
done
e2cp $1:/root/logs/out.txt $2/logs/out.txt
e2cp $1:/root/logs/pre.state $2/logs/pre.state
e2cp $1:/root/logs/post.state $2/logs/post.state
exit 0
```

Code 7 Script used to extract logs from IoT filesystem

The script will create a copy of the logs directory with all the files in the logs folder of the host machine for posterior analysis. The entire extraction process is controlled by the dynamic analyser and the logs path is the result of the dynamic analysis.

4.5 Parsing

The objective of the parsing is extracting all the useful fields from the logs and group them so they can be read and interpreted easily. The parsing as in the class diagram is done for the network logs and the system logs separately; only the dynamic analyser logs get parsed because the static analyser logs are getting parsed from the moment the binary is analysed.

4.5.1 Syscall parser

This parser uses the information from the log path at "logs/syscalls/" and extracts all the system calls that have been logged by the audit rules.

Audit logs have different types of entries for each event logged by a rule, there is an extensive list of types that are being stored, but I was only interested in the following.

- EXCVE: triggered to record arguments of the execve(2) system call which helps determent any commands that the binary has executed in the host.
- SOCKADDR: triggered to record socket address, it also contains the type of address is being used and what address and port that the process is trying to connect to.
- SYSCALL: triggered to record a call to the kernel. This will help get all the information we need about the system calls being logged.
- PATH: triggered to record full path and file name that are being accessed by a process, which helps get information about written and read files.

I found that by using the above types is more than enough to describe the behaviour of the executable.

The parsing process starts with reading chunks of the log files, which look like the following:

```
----

type=PROCTITLE msg=audit(06/11/20 02:12:01.046:3750717) : proctitle=/bin/bash
./starter.sh ./Tests/correlation_test.sh

type=EXECVE msg=audit(06/11/20 02:12:01.046:3750715) : argc=2 a0=/bin/bash
a1=././Tests/correlation_test.sh

type=PATH msg=audit(06/11/20 02:12:01.046:3750717) : item=0 name=/lib/x86_64-
linux-gnu/libtinfo.so.5 inode=392485 dev=08:01 mode=file,644 ouid=root ogid=root
rdev=00:00 nametype=NORMAL cap_fp=none cap_fi=none cap_fe=0 cap_fver=0

type=CWD msg=audit(06/11/20 02:12:01.046:3750717) : cwd=/home/user/TFM/Diseker

type=SYSCALL msg=audit(06/11/20 02:12:01.046:3750717) : arch=x86_64
syscall=openat success=yes exit=3 a0=0xffffff9c a1=0x7f4cb5529dd0
a2=O_RDONLY|O_CLOEXEC a3=0x0 items=1 ppid=14650 pid=14654 auid=unset uid=root
gid=root euid=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts1
ses=unset comm=correlation_tes exe=/bin/bash key=file-creation

----

type=PROCTITLE msg=audit(06/11/20 02:12:01.046:3750718) : proctitle=/bin/bash
./starter.sh ./Tests/correlation_test.sh

type=PATH msg=audit(06/11/20 02:12:01.046:3750718) : item=0 name=/lib/x86_64-
linux-gnu/libdl.so.2 inode=393055 dev=08:01 mode=file,644 ouid=root ogid=root
rdev=00:00 nametype=NORMAL cap_fp=none cap_fi=none cap_fe=0 cap_fver=0
```

```
type=CWD msg=audit(06/11/20 02:12:01.046:3750718) : cwd=/home/user/TFM/Diseker

type=SYSCALL msg=audit(06/11/20 02:12:01.046:3750718) : arch=x86_64
syscall=openat success=yes exit=3 a0=0xffffff9c a1=0x7f4cb550c4d0
a2=O_RDONLY|O_CLOEXEC a3=0x0 items=1 ppid=14650 pid=14654 auid=unset uid=root
gid=root euid=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts1
ses=unset comm=correlation_tes exe=/bin/bash key=file-creation
```

Where each event is separated with the line "----" and each event contains all the record types that have triggered for the event. The goal of the parser is getting the events one by one by using the separator and get the information only from the interesting record type.

All the data extracted from the logs need to be parsed to a data structure aligned with the Json format designed previously, to achieve that all the information from the different events is compiled and stored in the *AuditEvent* class which later can be transformed easily to a Json format. The fields that are going to be extracted from the events are "audit" that contains the timestamp and then the following fields in correspondence to the record type:

- From EXCVE the number of arguments of the command and reconstruct the command going through the arguments in order.
- From SOCKADDR the "saddr" field that contains the addresses.
- From SYSCALL the fields "syscall", "success", the four arguments (a0, a1, a2, a3) and finally the fields "uid", "gid" and "suid" that get stored in ownership attribute in the class *AuditEvent*.
- From PATH only the field "name" is extracted.

Parsing also maintains the relation between parent and child process by using the naming of the log files extracted. That is important because that information is used to create the process execution tree form the PID of the first execution to the last child created.

4.5.2 Network parser

The parsing process of the network logs starts with the logs in the network folder, and since the network logs can be quite big, I decided to parse the logs using multithreading, where each thread creates a list of *NetworkEvents* objects with all the information of a certain protocol which when finished it gets added to a hash table where the key is the protocol name. Once all the PCAP files were extracted the Snort alerts get parsed as well extracting the name of the alerts generated in the file *snort_alerts.txt*.

To dissect the packets in the PCAP files I used the python library PyPacker, it is simple to use and supports most of the protocols that this project intended to analyse, it also allows the implementation of new protocols if needed.

The information extracted from the network packets is, the time stamp, the source and destination address, the source and destination port and finally the payload. The payload format depends on what protocol being parsed, for the different protocol the following information will be parser:

- DNS: request and responses, including the type of request made.
- ICMP: only captures the ICMP code.
- NTP: it counts the number of NTP requests done in the spam of the emulation
- TELNET: we store the raw telnet data, with all the commands and results.
- IRC: we extract all the IRC commands the users, their destination, etc.
- HTTP: we get the http header with all its information.

4.6 Report generation

The reports generated will be based on a HTML template (Figure 13), the template was developed based on Bootstrap framework, which give users dynamic control over different components of page using multiple JavaScripts scripts that work like a framework for web developers.

Report generation starts when the raw data from the dynamic analyser is obtained and the data from the static analyser as well due to the relation defined in the class diagram in the previous sections. The report is divided in four main parts:

1. Summary: this section contains all the details about the file, such us the file hashes, the type of the file, the CPU architecture and finally the size of the file and name of it.
2. Static Analyser: this section holds all the analysis don in the static analyser section by representing the entropy with a graph and listing all the strings, sections and functions in tables that can be hided.
3. Dynamic Analyser: this section contains some simple statistics about the files read and written, and the amount of network traffic; the process tree and all the system calls captured by the rules and grouped by process child, if any, that has triggered them.
4. Network behaviour: this section will contain all the protocols being analysed and the most significant information from their payload. It will also contain Snort alerts that have triggered if any.

To be able to modify the template and add all the information listed above I used BS4 or beautiful Soup library which is normally used to pull data out from HTML and XML files, but in this project, it was used to extract key tags with certain ID that were used to add more tags and information to the template file and generate the report. Another tool that has been used is Matplotlib plots to generate the entropy graph and save it in the report destination folder.

The report is saved in a destination folder that the user specifies or just the project folder.

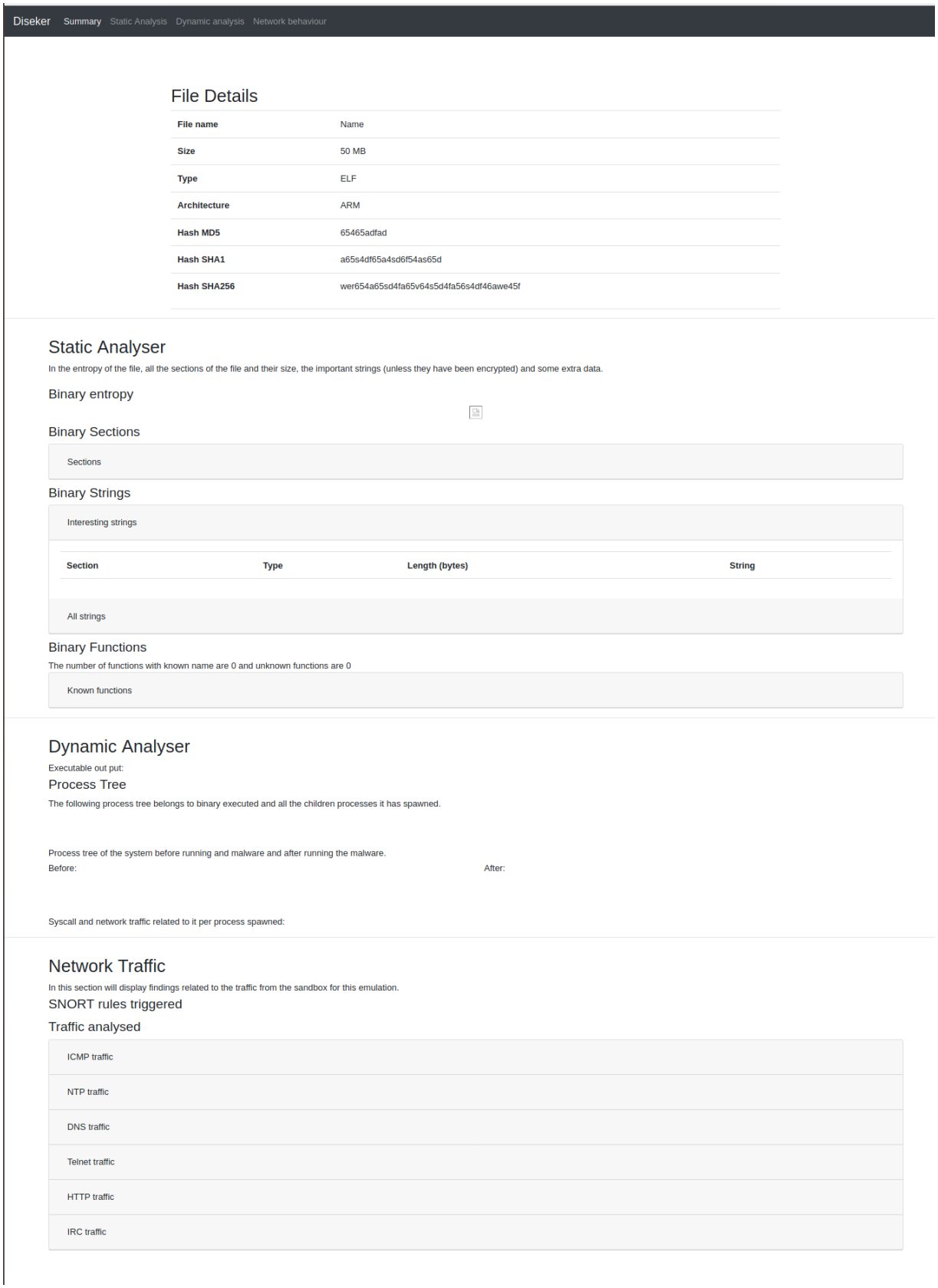


Figure 13. HTML template of the report

4.7 Network configuration

The network configuration is based on the creation of a bridge interface and a group of tap interfaces in the host which will be used by the Qemu instances when running to access the network. For this project I will create four TAP interfaces that will be used by the different parallel Qemu instances and the bridge will work as a switch between the TAP interfaces and the network interface called eth0.

The tools needed to create this bridge are:

- dhclient: dhcpd client that requests an IP from the server
- sysctl: tool used to configure kernel parameters at runtime
- tun module: a module that allows the creation of TUN/TAP interfaces

First make sure to have the tools available in the host:

```
$ sudo apt-get update
$ modprobe tun
$ sudo apt-get install dhclient
```

To create a bridge interface with name br0 you need to use the command:

```
$ sudo ip link add br0 type bridge
```

br0 must be the master of all interfaces in the Qemu network, which means that we need to make br0 master of eth0.

```
$ sudo ip link set eth0 master br0
```

To create a TAP interface and adding it to the bridge, the following command needs to be executed:

```
$ sudo tuntap add dev tap0 mode tap
$ sudo ip link set tap0 master br0
```

This can be done as much as TAP interfaces needed, the only thing to change is the name of the TAP interface. Once all the TAP interfaces have created and added br0 as a master interface, the interface br0 needs to be bought up and then assigned an IP address.

```
$ sudo link set dev br0 up #bringing up the interface
$ sudo dhclient br0 # requesting IP address from dhcp server
```

After that, an IP address needs to be assigned to the TAP interfaces under certain network, which also implies that the host needs to be able to forward packets and translate addresses since the TAP network will be different than the network of the host:

```
$ ip addr add 10.10.1.2/24 dev tap0 # Add the address to the TAP device

$ sudo sysctl -w net.ipv4.ip_forward=1 # Allow forwarding

$ sudo iptables -t -A POSTROUTING -s 10.10.1.0/24 -j MASQUERADE # Allow packet
# translation
```

5 Testing

The testing dataset to be used in the testing is from three main sources that combined we have obtained 5747 samples of malware of different types and architectures. The sources used to obtain the samples are:

- IoTPot honeypot dataset: the data obtained is a random sample from malware capture by the honeypot in between 2016 and 2017. All those malwares were obtained by the IoT honeypot set by the Research Center for Information and Physical security in Yokohama National University in Japan working together with Saarland University in Germany. The honeypot consists of a backend that contains different emulation systems of simple IoT devices of different architectures (MIPS, ARM, PPC, etc) and a vulnerable frontend that al the emulated systems share; this allows the attackers to take advantage of the vulnerable frontend and then the malware gets trapped and analysed by the honeypot.
- MalwareBazaar: It is a project operated by abuse.ch with the purpose of collecting and sharing malware samples.
- Stratosphere Lab dataset: Stratosphere Lab was created to fill the security need of civil organizations and NGOs with low budget and cannot afford spending money on sophisticated security systems, to achieve that the lab started the project Stratosphere which uses the latest academic achievement in the fields of machine learning and security to create an opensource machine learning based IPS[6] that can be used to detect malicious traffic. In the process of this project, the lab has released multiple samples of malware (including IoT malware) and information about APT attacks that helped researchers and the security community.

The dataset contains malware for most of the architectures that this project implements, as well as other architectures that will not be tested because they were not contemplated. But it is interesting to check the distribution of malware in the dataset (Figure 14), most samples in the public databases are mainly for ARM, MIPS or X86 architectures, this distribution is directly related to the architectures mostly used in the market, being X86 mostly for PCs, ARM for smart devices for user interaction like Android or for demanding hardware like high-speed routers, and finally MIPS which is the CPU architecture that most of home routers or small smart devices use. The other CPU architectures are also used for IoT devices or mostly devices that require low energy consumption and limited functionality.

[6] Intrusion Prevention System

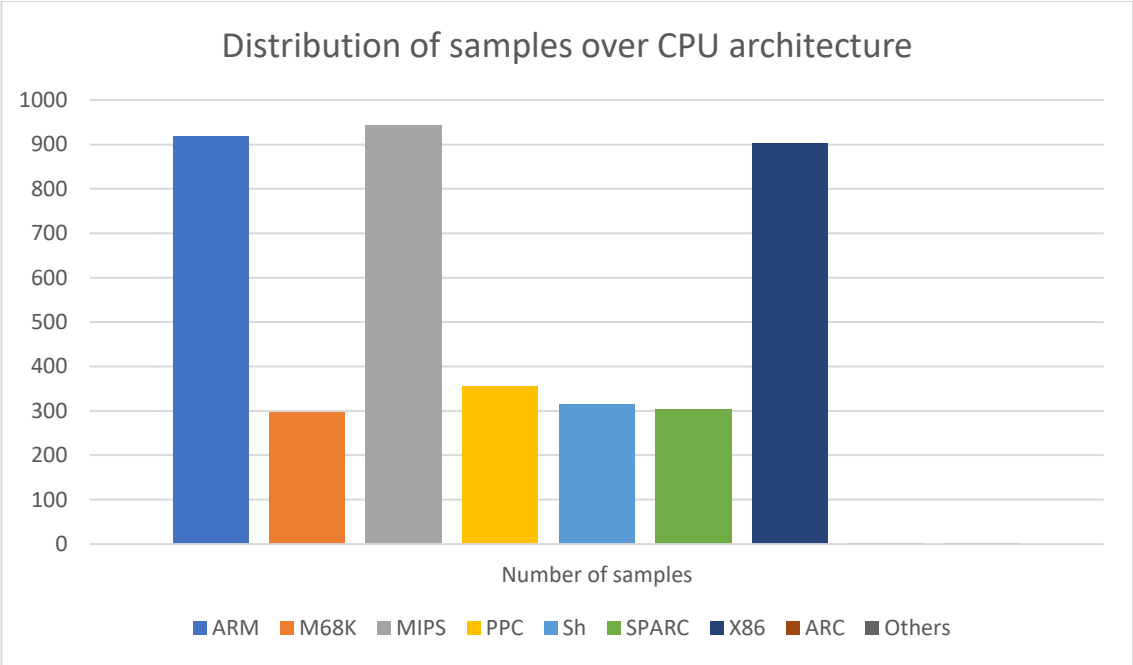


Figure 14. Samples distribution in the dataset

5.1 Testing methodology

The sandbox must be robust enough to determine the state of analysis and give a proper response to any unexpected situation. To evaluate that, I will be testing the user-sandbox interaction and then sandbox analysis phase individually, the first one will help understand if the sandbox is able to cover all possible input by the user and if it returns a report clear enough to give the user a clear idea about the process of emulation and execution of the malware. The second testing will consist of executing as many varieties of malware as possible (considering the dataset described above) and attempt to understand what has been achieved and what are the next steps to follow in future updates.

The testing process will not consider code and implementation, but rather logic, since it is considered that the process of verification of the good implementation was performed while coding.

5.2 Testing user-sandbox interaction

This test is done to make sure that the sandbox can evaluate the correctness of user input and the correctness of the report generated. The do the testing I have chosen a one sample of malware by CPU architecture supported and tested the following points:

Test	Expected	Result	Success
Unique execution with 32 bits, arch X86 and not forced	The malware executed and the results were shown in the report, no interpreter session was opened	The results the same as expected, the malware did not generate much data to show in the report	Yes
Unique execution with 32 bits, arch arm and not forced	The malware executed, the reports was generated, and the results were saved in the DB and	The output is the same as expected,	Yes

Unique execution with 32 bits, arch X86 and not forced	Read from the results from the DB since the execution is not forced	The sandbox used the hash of the file to read from the DB and did not execute the malware	Yes
Unique execution with 32 bits, arch X86 and forced	The file was found in the DB, but all the sandbox was run anyway because the user forced the emulation	The emulation was performed, even though the malware was found int the DB, because the user forced the execution	Yes
Unique execution with 32 bits, arch X86 and not forced with a malware that does not generate dynamic logs	The Sandbox should generate a report with an error message explaining to the user the reason why no details of the dynamic analyser are being shown	The result report does not detail the reason why the sandbox did not generate logs for the dynamic analyser, it only displays an error message	Partial
Unique execution with 32 bits, arch MIPS	The sandbox should show an error with the architecture supported for this implementation.	The sandbox shows an error and a usage message with the options of architecture that this implementation supports	Yes
Unique execution with 64 bits, X86 and not forced	The sandbox executed the malware and returns results	The sandbox executed and returned results	Yes
Unique execution with 16 bits, X86 and not forced	The sandbox shows and error message with the possible options for architecture Bits that can be used	The sandbox shows an error, a usage message and the possible bits that can be emulated.	Yes
Start an interactive session, with configuration file	The session starts and waits for the user to introduce commands	The session starts and waits for the user to introduce commands	Yes
Start an interactive session without a configuration file	The session should not start, and an error message must be shown to the user to ask him about the configuration file	The session does not start, and an error message is shown to the user asking for the configuration file	Yes
Start an interactive session and submit a command for 32 bits, X86 not forced for 200 seconds of execution time	The session loads the command executes it when a thread is freed, it goes through the same process as in the unique execution and returns the results when found.	The session loads the command executes it when a thread is freed, it goes through the same process as in the unique execution and returns the results when found.	Yes
Introducing a wrong command in an interactive session	The session shows an error with a help message. Does not execute or run any malware	The session shows an error with a help message. Does not execute or run any malware	Yes
Closing an interactive session	Before closing it will print all the analysis performed in the session, waits for running jobs to finish and then closes	It shows all the previous analysis done in that session after waiting for the running jobs to finish and then it closes.	Yes
Running an interactive session and introducing too many commands	The system will store all the commands in the waiting queue informing the user about it. The commands get executed every time thread is freed	The commands get added to the waiting queue and they get consumed when a thread is freed.	Partial
Run a command with files analysis was stored in DB without forcing	The system should return the result without emulation.	The system returns the results without emulation	Yes

Table 1 user-sandbox interaction tests

5.3 Testing sandbox analysis

To perform this test, we have chosen a random limited number of malwares for the different architectures that can be emulated. The number of samples will be low since I need to verify that the sandbox was able to extract all the behaviour successfully. For a test to be successful all the analysers and the traffic must capture the behaviour of the malware, the test is partial if only one analyser return results and finally the test will be considered failure in case of a lack of a tool or the inability of analysing the malware.

The samples to be used in the testing process are:

#	MD5	CPU arch	Linkage	C Library
1	dda3c47921bba43b4f33bf0ab27faa13	X86–32bits	Dynamic	uClibc
2	3ee5f6b919203c48f4512ae26a7dfc3f	ARM–32bits le	Dynamic	uClibc
3	76b40918b492402a696f9c4ac760df31	ARM-32bits le	Dynamic	uClibc
4	b2b0c9f6cd2a5c9c7d367667739a2744	ARM-32bits le	Dynamic	uClibc
5	3951bc82b1e4487a85eaa3986b829c80	X86–32bits	Dynamic	Glibc
6	11166712561c5b463c08f49d5213c1e0	X86–32bits	Static	Glibc
7	62379511e6848cbd920c40dc4495b0ce	X86-32bits	Dynamic	Glibc
8	868788ed5f594fb1de6dac82ae70a700	ARM-32bits le	Static	Glibc
9	b0efff0dafe3b234c177b28012c1161d	ARM-32bits le	Static	Glibc
10	88a9ed5408f20300ea79dd9c9b219379	ARM-32bits le	Static	Glibc
11	7d07e6669ae4f63d08a34a2b3edcd72f	X86-64bits	Static	Glibc

Table 2 Samples tested in the sandbox

All the files in the table above gave a result in the static analyser, which means that the analyser was able to extract the entropy, extract strings, sections and functions from the binary file. In the following table there is a small description of the behaviour captured by the dynamic analyser for each sample of Table (2).

#	MD5	Success	Observation
1	dda3c47921bba43b4f33bf0ab27faa13	Yes	The execution went correctly, but the malware seems to have not been successful with establishing connection with the CnC. No network traffic was captured.
2	3ee5f6b919203c48f4512ae26a7dfc3f	Partial	It did not execute the dynamic analyser because a library was not found in the image.

3	76b40918b492402a696f9c4ac760df31	Yes	The malware executed correctly and was able to spawn a child that tried to connect to the CnC at 173.212.226.176, but it was not able to establish connection with it.
4	b2b0c9f6cd2a5c9c7d367667739a2744	Yes	The malware executed correctly and was able to spawn a child that tried to connect to the CnC at 193.169.135.179, but it was not able to establish connection with it.
5	3951bc82b1e4487a85eaa3986b829c80	Yes	The execution went correctly, but the malware seems to have not been successful with establishing connection with the CnC. No network traffic was captured.
6	11166712561c5b463c08f49d5213c1e0	Partial	The file contained a very changing levels of entry and the extracted stings were not always readable. The sections of the file suggested that the malware was written with GoLang which was confirmed with the output from the malware. From the output I found that the malware created an HTTP server with a random name for a node and stated to do discovery. It is important to note that the IRC file of the traffic was not found within the logs which means that the malware stopped TCPDUMP from capturing the traffic.
7	62379511e6848cbd920c40dc4495b0ce	Yes	The execution went correctly, but the malware seems to have not been successful with establishing connection with the CnC 173.212.226.176:1664. No network traffic was captured.
8	868788ed5f594fb1de6dac82ae70a700	Partial	The static analyser was able to read all the strings from the file and extract the functions names, but the dynamic analyser was not able to capture the behaviour of the malware. Further manual analysis over the file showed that it does not allows its analysis by using ltrace and strace either.
9	b0efff0dafe3b234c177b28012c1161d	Partial	The malware executed and halted, without executing. It is possible that the malware is sleeping so it can be stealthy and avoid being run in sandboxes, this hypothesis could be true since the function “sleep” is present in the “sym” section.
10	88a9ed5408f20300ea79dd9c9b219379	Yes	The execution went correctly, but the malware seems to have not been successful with establishing connection with the CnC 89.34.97.132:48. No network traffic was captured.
11	7d07e6669ae4f63d08a34a2b3edcd72f	Yes	The execution went correctly, but the malware seems to have not been successful with establishing connection with the CnC 50.115.165.132:13174. No network traffic was captured.

Table 3 Results of testing the samples

The dynamic analyser in the other hand was not always successful. When testing the sandbox (with the samples above and other not listed) I found that there are other binaries that are able to spawn a malicious code in the memory from which *Auditd* was unable to capture the system calls (similar behaviour to the one in sample #8 at Table 13), this issue was mostly found when testing statically linked libraries. This gives the impression that the malware can modify the audit configuration and avoid being analysed, because similar statically linked binaries had their system calls correctly captured, another sample that was successful with avoidance technique it was the sample #6, this sample did not allow TCPDUMP from extracting IRC traffic, which may suggest that the malware itself uses that protocol to get in contact with other malicious entities in internet which could be either a Botnet or just an individual CnC.

All the previous analysis was done to ARM an X86 (32 bits and 64 bits), the other CPU architectures that this sandbox was meant to support presented some issues, for PPC (PowerPC) the auditing tool *auditd* crashed with segmentation fault for every combination of Linux Kernel, toolchain or other configuration which means that I was not able to dynamically analyse any of the malware executed in that instance, for the other hand for Aarch64 architecture I was not able to find any malware sample that I could use to test it with. But in general, the sandbox was able to disclose big part of the malware behaviour as well as provide OSINT to detected future possible infections.

5.3.1 Analysis of a tested sample

This section consists of a step-by-step process of interpretation of the analysis results of the sample 1 from Table (2) with MD5 hash dda3c47921bba43b4f33bf0ab27faa13. The sample has the following details:

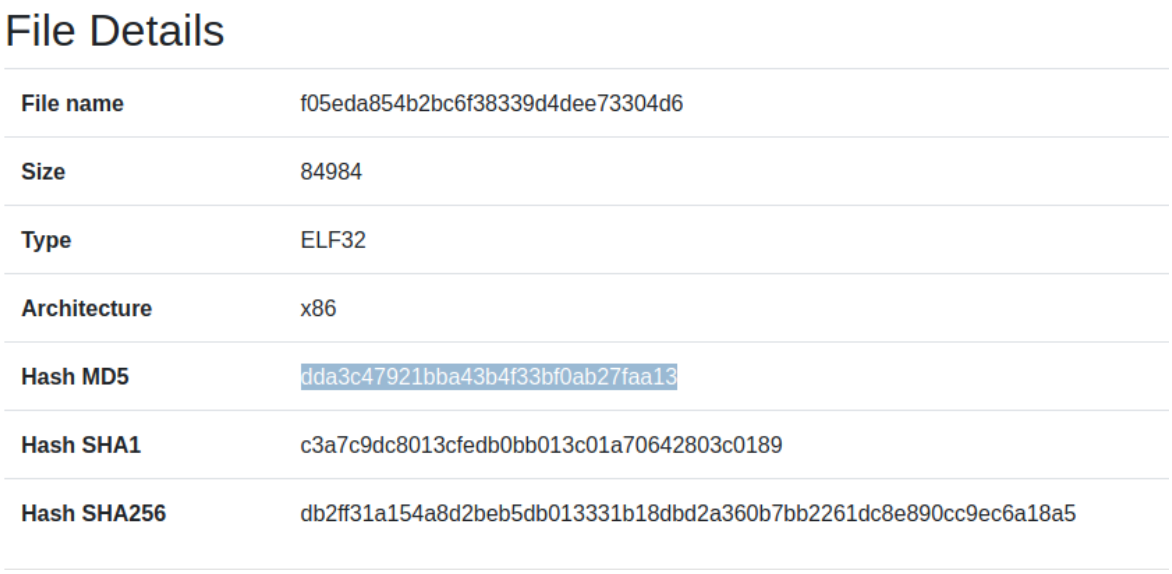


Figure 15 Sample details provided by the sandbox

All the details of the binary were exported successfully as in Figure (15), the other important information that has been obtained as well is the entropy of the file, which seems to be low, with section with lower amount uncertainty than others, this information can help make sense of the strings extracted from the binary, in this case there is a dip of entropy in around

“.eh_fram, init_array and fini_array” sections, which means that that area is filled with the same value.

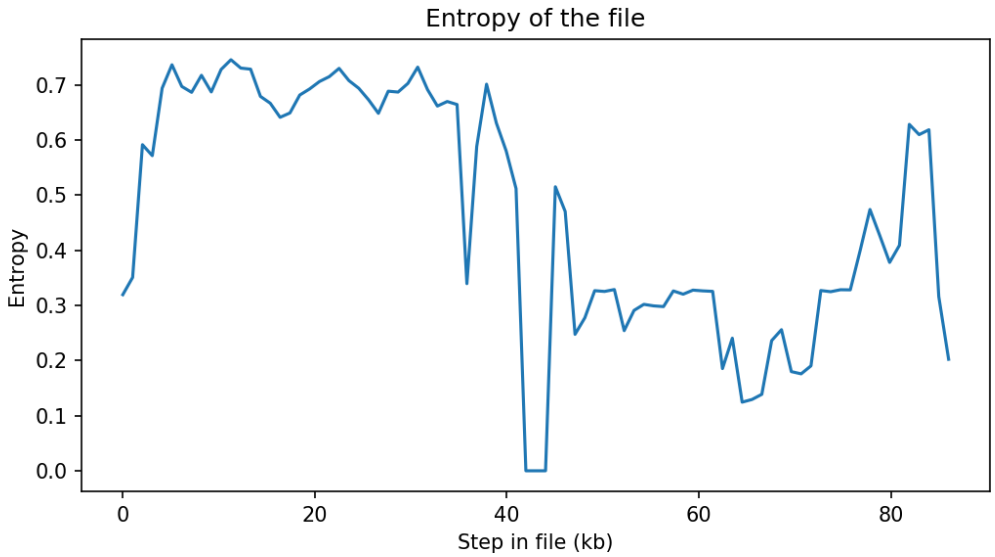


Figure 16 Entropy graph of the sample

The static analyser was also able to extract all the string from the binary successfully for this sample, the list of strings is big since I opted on extracting all the possible strings from the binary. For this sample, multiple strings

.rodata	ascii	15	REPORT %s:%s:%s
.rodata	ascii	28	INFECTION SUCCESS - %s:%s:%s
.rodata	ascii	20	Telnet'd %s|%s|%s|23
.rodata	ascii	27	FAILED TO INFECT - %s:%s:%s
.rodata	ascii	26	Failed opening raw socket.
.rodata	ascii	32	Failed setting raw headers mode.
.rodata	ascii	17	Invalid flag "%s"
.rodata	ascii	12	wget -s -U "
.rodata	ascii	34	wget -O /tmp/yuagwduiagwdhg/a -U "
.rodata	ascii	29	KILLSUB <sub version to kill>
.rodata	ascii	42	not killing myself cuz im not that version
.rodata	ascii	24	SCAN <threads> <timeout>

Figure 17 Strings samples from the static analyser

The strings also contained information about the remote addresses that the device will attempt to connect to or the browser agents that will be using:

.rodata	ascii	20	193.169.135.179:1665
.rodata	ascii	74	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/20100101 Firefox/13.0.1
.rodata	ascii	106	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.56 Safari/536.5
.rodata	ascii	108	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.11 (KHTML, like Gecko) Chrome/20.0.1132.47 Safari/536.11
.rodata	ascii	117	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/534.57.2 (KHTML, like Gecko) Version/5.1.7 Safari/534.57.2
.rodata	ascii	67	Mozilla/5.0 (Windows NT 5.1; rv:13.0) Gecko/20100101 Firefox/13.0.1
.rodata	ascii	119	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.11 (KHTML, like Gecko) Chrome/20.0.1132.47 Safari/536.11
.rodata	ascii	67	Mozilla/5.0 (Windows NT 6.1; rv:13.0) Gecko/20100101 Firefox/13.0.1

Figure 18 Names of browser agents and addresses found by the static analyser

And more text related information, which is a very good OSINT source. The static analyser was also able to extract some function names implemented in the file, which can give a general idea about the possible interaction that the malware has with the device and the internet. For this sample, the malware seems to be able to use HTTP, can send data to a command and control sere as well as get the public IP of the device.

134526604	sym.socket_connect	267
134526871	sym.echoLoader	220
134527091	sym.StartTheLelz	5674
134532765	sym.sendSTD	243
134533008	sym.sendUDP	1219
134534227	sym.spoofTest	690
134534917	sym.sendTCP	1193
134536110	sym.sendHTTP	504
134536614	sym.sendHTTP2	526
134537140	sym.sendCNC	165
134537305	sym.processCmd	4893
134542198	sym.initConnection	289
134542487	sym.getOurIP	479
134542966	sym.getBuild	10

Figure 19 Sample of function names extracted by the static analyser

The dynamic analysis performed by the sandbox was successful for this malware, it was able to extract all the behaviour possible, which is simple the installation of the malware in the memory of the device and then attempt to contact with command-and-control server. The malware generated two processes and one printed “BUILD GetWrecked” to the standard output as in Figure (20).

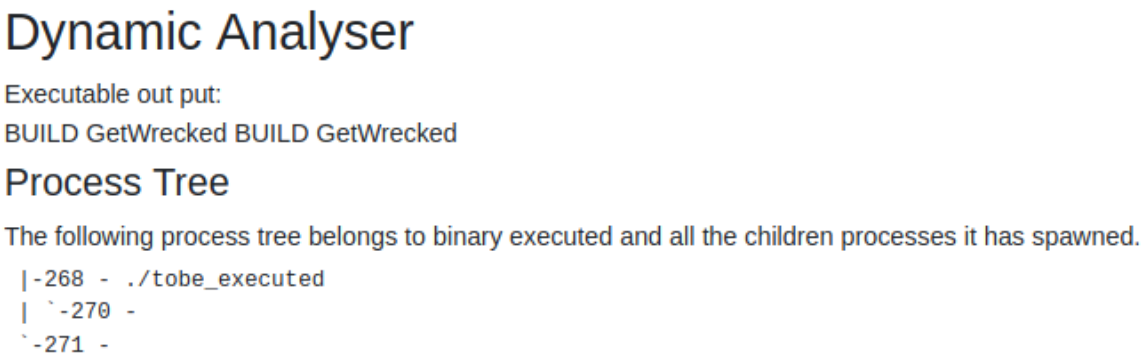


Figure 20 Process tree of the malware

In Figure (20) we also observe that the malware executed with PID 268, generated a child process with PID 270 and both seem to have finished, but the sandbox was able to capture one suspicious process that has spawned after the malware has been executed on PID 271, the suspicious PID can be seen added to the processes in execution after running the malware in Figure (21).

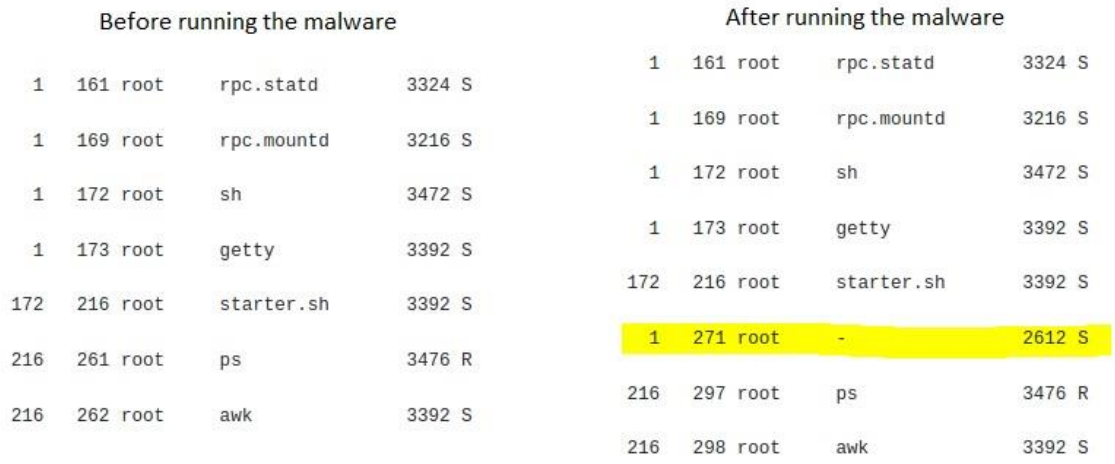


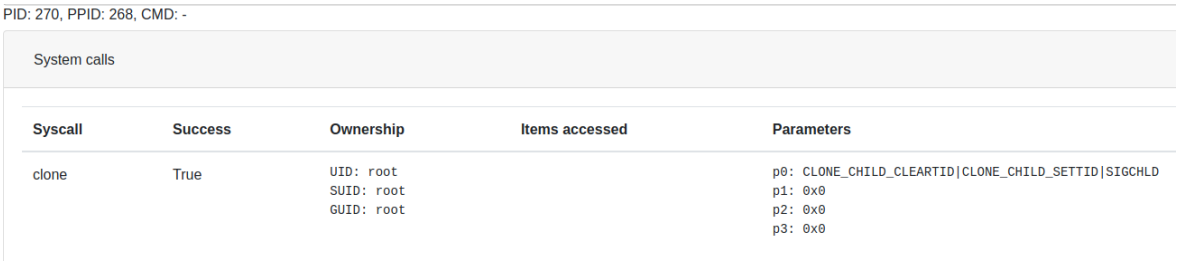
Figure 21 State of process executing in the device before and after the malware was executed

Auditd was able to capture the system calls in the rule set that the processes detected performed, which for the process with PID 268, the interesting system calls were getting its PID, reading the files “/etc/rc.d/rc.local” and “ /etc/rc.conf”, establishing a connection with the IP 8.8.8.8 over port 443, reading the file “/proc/net/route” then cloning itself.

getpid	True	UID: root SUID: root GUID: root		p0: 0xbff172e0 p1: 0x5fe7da08 p2: 0xb7f183a4 p3: 0x5fe7da08
openat	False	UID: root SUID: root GUID: root	- /etc/rc.d/rc.local	p0: 0xffffff9c p1: 0x8051720 p2: O_RDONLY p3: 0x0
openat	False	UID: root SUID: root GUID: root	- /etc/rc.conf	p0: 0xffffff9c p1: 0x8051735 p2: O_RDONLY p3: 0x0
socket	True	UID: root SUID: root GUID: root		p0: inet p1: SOCK_DGRAM p2: ip p3: 0x5fe7da08
connect	True	UID: root SUID: root GUID: root	- inet - 8.8.8.8 - 443	p0: 0x3 p1: 0xbff15d1c p2: 0x10 p3: 0xb7f10000
openat	True	UID: root SUID: root GUID: root	- /proc/net/route	p0: 0xffffff9c p1: 0x80516f0 p2: O_RDONLY p3: 0x0
clone	True	UID: root SUID: root GUID: root		p0: CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD p1: 0x0 p2: 0x0 p3: 0x0

Figure 22 Sample of system calls of PID 268

The last clone dispatched the child process at PID 270 which created another process with PID 270.



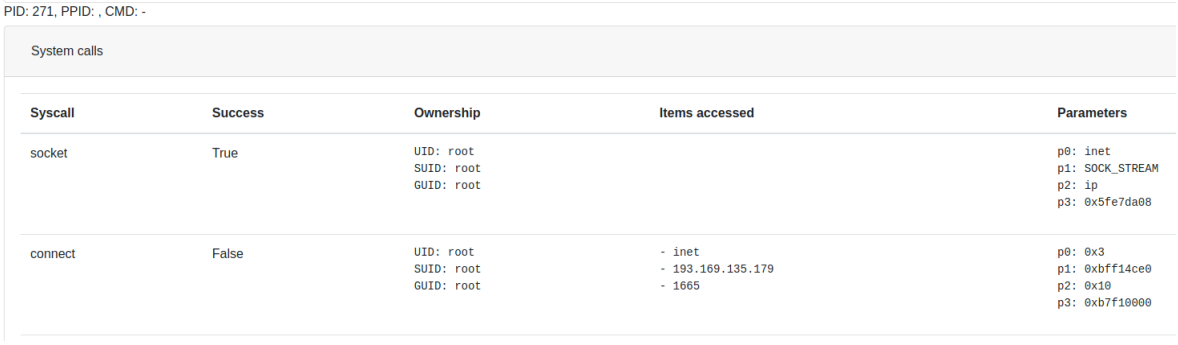
PID: 270, PPID: 268, CMD: -

System calls

Syscall	Success	Ownership	Items accessed	Parameters
clone	True	UID: root SUID: root GUID: root		p0: CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD p1: 0x0 p2: 0x0 p3: 0x0

Figure 23 System calls of PID 270

The final child, which is the suspicious process that was detected by the sandbox, was attempting to establish a connection with the address “139.169.135.179” over port 1665 and no attempt was successful.



PID: 271, PPID: , CMD: -

System calls

Syscall	Success	Ownership	Items accessed	Parameters
socket	True	UID: root SUID: root GUID: root		p0: inet p1: SOCK_STREAM p2: ip p3: 0x5fe7da08
connect	False	UID: root SUID: root GUID: root	- inet - 193.169.135.179 - 1665	p0: 0x3 p1: 0xbff14ce0 p2: 0x10 p3: 0xb7f10000

Figure 24 System calls of PID 271

6 Conclusions and future work

The field of malware analysis is never ending, due to the emerging new technologies with wide attack surface or the sophisticated techniques that threat actors use to infect and take advantage of victims. In this project I have discussed a readapted sandbox for IoT devices that gives more verbosity to the analysts and supports multiple CPU architectures. This sandbox (Diseker) can capture system calls, files accessed or modified as well as attempts to establish traffic connections with CnC servers. When testing the sandbox, I found that it was not infallible and there are some considerations that could be taken to improve its efficiency, such us adding a tracing tool like *ltrace* or *strace* to the analysis pipeline, add a network traffic analysis system based on deep packed analysis to detect the type of traffic being sent and received without counting on TCP/UDP ports, which could be misleading, add a flow based analyser to detect the involvement in DDoS attacks or similar attacks, and finally support way more platforms and CPU architectures. It may be noted that this implementation of the sandbox did not comply with all the requirements established for it, to be specific these three R8, R9, R10 and some protocols that I was not able to implement the parser for, either ways Diseker is a working tool that can be used to analyse most of the IoT malware in the market at this moment.

Even though the sandbox can be improved a lot, it did help determine a common pattern used by most of the malware tested in it, the common pattern detected can be simplified in 4 steps which are:

1. Reconnaissance: the malware tests the network by connecting to a known service, for example requesting connection to 8.8.8.8:443 or 8.8.8.8:53, to verify the internet connect; the process of reconnaissance also includes the attempt to opening system files such as “/etc/rc.conf | /etc/rc.d/rc.local” to gain persistence or read system information from locations like “/proc/net/route”.
2. Despatch: after the malware has established the capabilities of the system it either continues executing with the same first process or dispatches a child process with different PPID to be running fileless in the system memory.
3. CnC beaconing: once the malware has well established its presence in the device it starts beaconing traffic towards the CnC server confirming its infection to a device.
4. Command-and-control: at this point the malware has established first connection with the CnC and now it is performing the tasks asked to do.

The previous steps were inferred from the information provided by the static analyser and the dynamic analyser as well, as well as the out put of the malware and the processes executing in the system after running the malware.

The process of developing this sandbox was thrilling and this is only the first version, it will be interesting to implement all the possible features listed above, as well as compare the effectiveness of it with other implementations, as well as compare different monitoring tools provided for each CPU architecture. I did not know what to expect when I have started the project, but I do know now that the amount of work and research needed for such tool is way more than what I was expecting, the research is extensive either due to the diversity in the IoT world or the diversity in the malware to analyse, in either cases it is must to consider all possibilities.

7 References

1. **Sikorski, Michael and Andrew Honig.** *Practical Malware Analysis.* San Francisco : No Starch Press, Inc, 2012. 1-59327-290-1.

2. **V.N. Parasram, Shiva.** *Digital Forensics with Kali Linux (Second Edition).* BIRMINGHAM : Packt Publishing Ltd, 2020. 978-1-83864-080-4.

3. **G. Conrads, Jessica.** *DDoS Attack Fingerprint Extraction Tool: Making a Flow-based Approach as Precise as a Packet-based.* Enschede, The Netherlands : University of Twente, 2019.

4. *IoTCandyJar: Towards an Intelligent-Interaction Honeypot for IoT Devices.* **Luo, Tongbo, et al.** Palo Alto : Palo Alto Networks Inc., 2017.

5. *IoTPOT: A Novel Honeypot for REvealing current IoT Threats.* **Yin Minn, Pa Pa, et al.** 3, s.l. : Journal of Information Processing, 2016, Vol. 24.

6. **detuxsanbox.** Detux: The Multiplatform Linux Sandbox. 2018.

7. **Uhrıcek, Daniel.** LiSa – Multiplatform Linux Sandbox for AnalyzingIoT Malware. [Online] 2019. http://excel.fit.vutbr.cz/submissions/2019/058/58.pdf.

8. *V-Sandbox for Dynamic Analysis IoT Botnet.* **HAI-VIET, LE and Guoc-Dung, Ngo.** s.l. : IEEE Access, 2020, Vol. 8. 2169-3536.

9. **Buildroot.** Buildroot. *The Buildroot user manual.* [Online] https://buildroot.org/downloads/manual/manual.html.

10. **Qemu.** Qemu. *QEMU System Emulation User's Guide.* [Online] QEMU. https://www.qemu.org/docs/master/system/index.html.

11. **Hjelmvik, Erik.** NETRESEC. *Installing a Fake Internet with INetSim and PolarProxy.* [Online] 09 December 2019. https://www.netresec.com/?page=Blog&year=2019.

12. **Kambourakis, Georgios, et al.** *BOTNES: Architectures,Countermeasures, and Challenges.* New York : CRC Press, 2020. 978-0-367-19154-2.

13. **Winward, Ron.** *IoT Attack Handbook: A Field Guide To Understanding IoT Attacks from the Mirai Botnet to its Modern Variants.* s.l. : Radware, 2018.

14. *IoT sandbox: to analysis IoT malware Zollard.* **Chang, Kai-Chi , Tso, Raylin and Tsai, Min-Chun.** 4, Cambridge, United Kingdom : Association for Computing MachineryNew YorkNYUnited States, March 2017. 978-1-4503-4774-7.

15. *Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.* **Daming, D. Chen, et al.** San Diego : s.n., 2017, NDSS Symposium 2017. 1-891562-41-X.

16. **OpenWRT.** OpenWRT. *OpenWRT in QEMU.* [Online] 09 11 2020. https://openwrt.org/docs/guide-user/virtualization/qemu.

17. **Python Software Foundation.** The Python Standard Library. [Online] 05 December 2020. https://docs.python.org/3/library/index.html.

18. **Stahn, Michael.** GitLab. *Pypacker.* [Online] 2018. https://gitlab.com/mike01/pypacker.

19. **PenTestParteners.** PenTestPartners: Security Consulting and Testing Services. *Thermostat Ransomware: a lesson in IoT security.* [Online]

https://www.pentestpartners.com/security-blog/thermostat-ransomware-a-lesson-in-iot-security/.

20. **Dickson, Ben.** IoT Security Foundation. *The IoT ransomware threat is more serious than you think.* [Online] https://www.iotsecurityfoundation.org/the-iot-ransomware-threat-is-more-serious-than-you-think/.

21. *Understanding Linux Malware.* **Cozzi, Emanuele, et al.** San Francisco : 2018 IEEE Symposium on Security and Privacy (SP), 2018.