Architecture for analyzing Potentially Unwanted Applications

Alberto Geniola

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 10.10.2016

Thesis supervisor:

Prof. Tuomas Aura

Thesis advisor:

M.Sc. Markku Antikainen

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Alberto Geniola

Title: Architecture for analyzing Potentially Unwanted Applications

Date: 10.10.2016 Language: English Number of pages: 8+149

Department of Computer Science

Professorship: Information security

Supervisor: Prof. Tuomas Aura

Advisor: M.Sc. Markku Antikainen

The spread of potentially unwanted programs (PUP) and its supporting pay par install (PPI) business model have become relevant issues in the IT security area. While PUPs may not be explicitly malicious, they still represent a security hazard. Boosted by PPI companies, PUP software evolves rapidly.
Although manual analysis represents the best approach for distinguishing cleanware from PUPs, it is inapplicable to the large amount of PUP installers appearing each day. To challenge this fast evolving phenomenon, automatic analysis tools are required. However, current automated malware analisyis techniques suffer from a number of limitations, such as the inability to click through PUP installation processes. Moreover, many malware analysis automated sandboxes (MSASs) can be detected, by taking advantage of artifacts affecting their virtualization engine.
In order to overcome those limitations, we present an architectural design for implementing a MSAS mainly targeting PUP analysis. We also provide a cross-platform implementation of the MSAS, capable of running PUP analysis in both virtual and bare metal environments. The developed prototype has proved to be working and was able to automatically analyze more that 480 freeware installers, collected by the three top most ranked freeware websites, such as cnet.com, filehippo.com and softonic.com. Eventually, we briefly analyze collected data and propose a first strategy for detecting PUPs by inspecting intercepted HTTP traffic.

Keywords: PUP, PPI, MSAS, PUA, Potentially Unwanted Application, Potentially Unwanted Programs

Preface

This thesis work constitutes the final step of my master education in Computer Engineering. It has been written during my exchange year, spent at the Aalto University, in Helsinki.

I wish to thank professor Tuomas Aura, who gave me the opportunity of working in the research area of the information security deparment. I had the opportunity to expand my education background, by working with IT security research experts, immerse in an intellectually challenging environment. Special thanks go to Markku Antikainen, whose valuable advices guided me while writing the thesis. I must also offer a very special word of thanks to all my researchers colleagues of the security department, especially to Blomqvist Tuomas, who demonstrated genuine interest about my work. Great thanks go to professor Fulvio Risso, for his constant (and valuable) availability and support.

I would like to thank my two flatmates, Luigi and Raul, whose sympathy and cheerfulness have been a relief during hard times. Furthermore, I would like to thank all my closest friends Marina, Marta, Valentina, Valeria D. for their constant emotional support. Thanks to my girlfriend Elena, who always believed in me and supported my ideas. Special rewards go to Valeria Z., who inspired me with her *IT-difficulties*.

I must spend special word of thanks for Francesco and Riccardo, two of my closest friends who shared hard times during these two years of master studies.

Lastly, I would like to thank all family that has always stood by my side. Great thanks to Maria, whose generosity helped me in finishing my studies.

Turin, Sep 26, 2016

Alberto Geniola

This work was supported by TEKES as part of the Cyber Trust program of DIGILE (the Finnish Strategic Center for Science, Technology and Innovation in the field of ICT and digital business).

Contents

Symbols and abbreviations

Abbreviations

API	Application Programming Interface
CTPH	Context Triggered Piecewise Hashing
DNS	Domain Name Service
EULA	End User License Agreement
FS	File System
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol over Secure Socket Layer
IO	Input Output
IT	Information Technology
JSON	JavaScript Object Notation
MAC	Media Access Control
MITM	Man In The Middle
MSAS	Malware Sandbox Analysis System
MSI	Microsoft Installation Package
NIC	Network interface controller
OCR	Normalized Character Recognition
OEM	Original equipment manufacturer
OS	Operating System
PE	Portable Executable
PPI	Pay Per Install
PUA	Potentially Unwanted Application
PUP	Potentially Unwanted Program
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

The growth of Internet in terms of availability and bandwidth has changed the way software is distributed. New software monetization techniques arose, such as advertising or user's data collection. While those techniques enable free software to spread all over the web, they still introduce new security challenges, mainly affecting users' privacy. In particular, *spyware* and *adware* represent two *malware* classes aimed at enabling software monetization.

Recently a new kind of software monetization technique came to light. Its business model is based on the distribution of legitimate bening software bundled within unknown, third party software, which generally offers poor or none functionality [63]. In order to gain legal right of collecting user's data or installing third party software on the user's system, their installers display complex end user agreements [17, 33], that users tend to skip by habituation [19, 54]. Furthermore, those programs tend to present tricky installation procedures, aimed at stealing the user's consent for performing questionable operations on his system. Therefore, users not paying sufficient attention during the installation process tend to install programs which they are not aware of. Those programs are usually undesired and unexpected by users, who may realize their presence because of behavioral changes of their system, brought by such *unwanted programs*. Classic examples are invasive *toolbars* which also change the browsing experience, by altering the default homepage and search engine results. Fake antimalware and registry cleaners products represent another class of unwanted applications, generally addressed as *rogue software* or *grayware* [63]. Current antimalware vendors include all those software classes in a wider definition, such as *Potentially Unwanted Programs* (PUP) or *Potentially Unwanted Applications* (PUA) [8, 34, 47, 78].

While PUPs might not be malicious by their nature, they may still be undesired by users. Usually, those kind of applications come with no warranty, nor they comply with any security standard. As a consequence, the installing users end up with third party software they are not aware of, which certainly enlarge software's surface of attack. This phenomenon possibly introduces vulnerabilities on the affected systems. Therefore, it is reasonable to consider the PUP class as dangerous, especially when suspicious behaviors are observed on the system.

While spyware and malware are considered malicious, the case of PUP is pretty different. The main difference between malware and PUPs stands in the user's consent of installing the software. While malware behaves secretly and silently, without asking for user's permission, PUPs extort user's consent to operate with his permission [19]. This

explicit *uninformed consent* enables the application to operate in a *legal gray zone*. As a consequence, antimalware products are not entirely entitled to automatically remove that software [27, 28]. This legal issue is aggravated by the business model standing behind the PUP world, generally addresses as *Pay Per Install*. Recent studies confirmed that such business model is particularly valuable [86] and widely adopted over the Internet. Thus, companies implementing the PPI model can count on a solid economic base, then have economic resources to legally fight against antimalware products affecting their products [27].

Current antimalware products recognized the threat of PUP as a relevant issue. For this reason, major antimalware vendors started to adopt some client-side countermeasures in order to mitigate the spread of PUPs. However, considering the legal issues they need to face, they tend to adopt conservative detection methods, based on user's feedback, blacklisting and signature-based detection. Those methods represent classic tools for fighting malware threats, but are unable to address the new ones introduced by PUPs. On the other hand, PUP detection and classification is still a open research problem. Current antimalware products rely on a mix of indicators and manual inspection of software in order to classify it as PUP. For instance, they check whether toolbars are installed or default search engine is altered. However, those indicators might led to misclassification errors, causing legal issues we discussed previously. For this reason, antimalware vendors offer public submission forms for requesting manual checks and whitelisting of false positive detections [41].

Due to the solid business model supporting PUP spread, PUP installers are continuously evolving, refining their behaviors and avoiding classic detection systems. As a consequence, antimalware products need to evolve in turn in order to catch up the gap between them. Clearly, manual inspection of PUPs is not fast enough to react against new PUPs. Therefore, semi-automatic analysis mechanisms can deliver better results in the battle against PUP spread. For this reason, we introduce an architecture for a full automated sandbox for analyzing unknown software installers. Our goal is to adapt classic malware analysis techniques for analyzing PUP installers, solving a number of limitation affecting current automated analysis tools. In particular we focus in describing a distributed, multiplatform architecture, capable of handling unmanned software installation and collecting low-level information about software execution. By merging static and dynamic analysis techniques, together with general user interface automation, we implemented a prototype of the architecture. Our prototype demonstrated its scaling capabilities and provided detailed behavioral information about analyzed installers. Moreover we tested the goodness of collected data and we identified a first simple indicator for detecting PUPs by looking at their HTTP downloads.

1.2 Research goals

The main objective of this work is to define a system architecture able to automate static and dynamic analysis for PUP installers, capable of overcoming limitations affecting current automated sandboxes. In particular we focus on automating user interface interaction offered by PUP installers, taking advantage of computer vision and text recognition technologies. At the same time our system implements an isolated sandbox, capable of intercepting and logging low level access to system's resources.

The major contributions brought by our PUP analysis sandbox include the followings:

- providing a cross-platform, distributed architecture with acceptable scaling capabilities, able to automatically analyze thousands of binaries per day.
- delivering automatic human-like interaction mechanism for clicking through installing procedures, directly from within the sandbox environment
- collecting behavioral information about the analyzed binary, by implementing dynamic and static analysis capabilities
- providing both virtual and bare-metal support, so that virtual machine aware software is identified by running the test on both environments
- providing aggregated and easy-to-query data, representing a truth ground on which to study PUP behaviors or to define new PUP detection techniques.

1.3 Structure

This thesis is divided in eight chapters. Chapter 1 presented the central topics of this work, alongside the our research goals. In the second chapter we provide an overview of PUP, discussing its definition and describing its classification problems. We also describe why the PUP problem affects Windows operating systems and what are the challenges related to those operating systems. Moreover, second chapter also describes static and dynamic analysis technologies, on which we based part of our work. Chapter 3 describes other research works that are related to automated sandbox analysis, PUP detection and user interface automation. In chapter 4, we discuss problems affecting solutions considered in the previous chapter, which are later on addressed in the directly in following chapters. Chapter 5 presents the architecture of the analysis system we developed, from high level perspective. It provides a wide overview of the nodes composing the analysis systems, pointing out objectives and goals for each component of the system. Chapter 6 describes some relevant technical aspects regarding the implementation of some core components of the architecture. It enlighten how we applied DLL injection and API hooking in order to implement the sandboxing system. In chapter 7 we describe test configurations and present obtained results. Furthermore, in the same chapter we briefly analyze collected data, and identify a possible way for identifying certain classes of PUPs. The document is concluded with our final remarks, exposed in chapter 8.

Chapter 2

Background

In this chapter we will introduce the fundamental concepts needed to comprehend our work. At first, a general description of PUP is given. It follows a short analysis of the problem on Windows operating systems. Later on, common analysis techniques are described. Afterwards, a brief overview of the Window's user interface system is provided. In the same section some user interface automation frameworks will be presented. We conclude this chapter explaining which are the limitations of current technologies.

2.1 Potentially Unwanted Programs

Our work focuses on automating PUP installation in order to enable more reactive PUP detection systems. Thus, it is crucial to familiarize with the PUP technology before taking into account further technical aspects. For this reason, we will discuss fundamental concepts behind PUPs, giving to the reader a general overview of this software type.

2.1.1 Definition

It is difficult to define Potentially unwanted programs (PUP), although some definitions exist. ESET identifies *Potentially Unwanted Program* (PUP), also known as *Potentially Unwanted Application* (PUA)[1] as a particular type of software with its set of associated behaviors. Not necessarily malicious, this class of programs is able of installing additional unwanted software, thus changing the behavior of the hosting digital device [34]. In the scope of this work, we will embrace this definition. Being general and wide, the given definition includes genuine software as well as malware. Indeed, a program may be classified as PUP even when it is totally unharmful, but behaves in an unexpected or suspicious manner. On the contrary, trojan and spyware are considered subsets of PUPs [33] and classified as malicious software. Among all the possible suspicious behaviors that a software may expose, there are some common patterns that PUPs tend to include [34, 47], such as:

- Implicit browser plugin or toolbars installation
- Implicit ADWare installation
- Installation of software following the pay-per-install business model
- Changes to the web browser settings (search engine, homepage)

[1] or *Potentially Unwanted Software* (PUS) or simply *rogue software* [63]

- Changes to the network system settings (DNS/Proxy)
- Installation of vulnerable software that can be exploited
- Leak of adequate uninstall procedures

Although some of these are not easy to notice by non tech-savvies (e.g. network settings change), usually the side effects of PUP are invasive and evident to the end user [74]. That is the case of some browser plugins, shipped with legitimate software, which present invasive advertisements and pop-ups, heavily impacting on the user experience.

Hidden bitcoin miners (a.k.a. Potentially Unwanted Miners) present another case of PUP: usually they do not gather personal user information, neither affect the Internet browsing sessions of the user explicitly; instead they consume system CPU impacting on the system usability. Malwarebytes LABS reported the case of `yourfreeproxy.net`, a website that used to distribute a free proxy application, installing a bitcoin miner on the user's system [11]. In that case, the end user license agreement (EULA) reported the necessity of mathematical calculation to improve the system security. However, the CPU power was used to mine bitcoins, without sharing any of the earnings with the user. At the time of writing `yourfreeproxy.net` has been closed. Moreover, its PUPs are now considered malicious by major antimalware products [91].

2.1.2 Potentially-unwanted ambiguity

Beside a formal definition has been provided, the concept *potentially unwanted* still offers many sources of ambiguity. It is non-trivial to classify which software is desired and which one is unwanted. The difficulty of this classification lays on the fact that many programs may legitimately use third party libraries or services. These third party components may be necessary to the original software in order to provide some core functionality, nevertheless the user may classify those as unwanted.

Another important aspect to consider is the knowledge gap among computer users. It's reasonable to assume that basic computer users are not familiar with the technicalities, and they tend to look at a program like a monolithic entity. Tech-savvies, instead, are aware of software modularity and may be able to detect what is really a desired feature and what is unwanted. This knowledge disparity is the basement on which the adverb "potentially" lays on. While for expert users there would be little or no uncertainty for classifying unwanted software, that is not the case of end-users. For this reason some antispyware and antivirus companies try to help non IT users by performing a classification of software as "potentially unwanted", warning them that the program they are installing may not be desired [33].

Potentially-Unwanted Programs and Potentially-Unwanted Data

Another source of ambiguity regards the relationship between software and data. Although both the terms "program" and "software" are usually used as synonyms, there is a difference between them. According to *IEEE Standard Glossary of Software Engineering Terminology*, a **program** is defined as *a set of computer instructions and data definitions*, while a **software** is a *set of computer programs and associated data* [10]. The latter definition better covers some PUP border line cases, when the unexpected software behavior is limited to some file-drops on the user's system. As an example, we might consider a legitimate software that, alongside its basic features, also installs an *unwanted* Root Certificate Authority (CA). In this case, data is unwanted, rather than software. The installer

software itself may now be categorized as PUP, because it installed a Root CA with little or no notification to the user.

2.1.3 PUP Classification

Malware is malicious software, that causes damage to users, computers or network [75]. On the other hand, legitimate software is sometime referred as *cleanware*. Given the definition of PUP, it is hard to establish a hierarchical relationship between malware, cleanware and PUP. In fact, all the three software classes lay on the same logic layer.

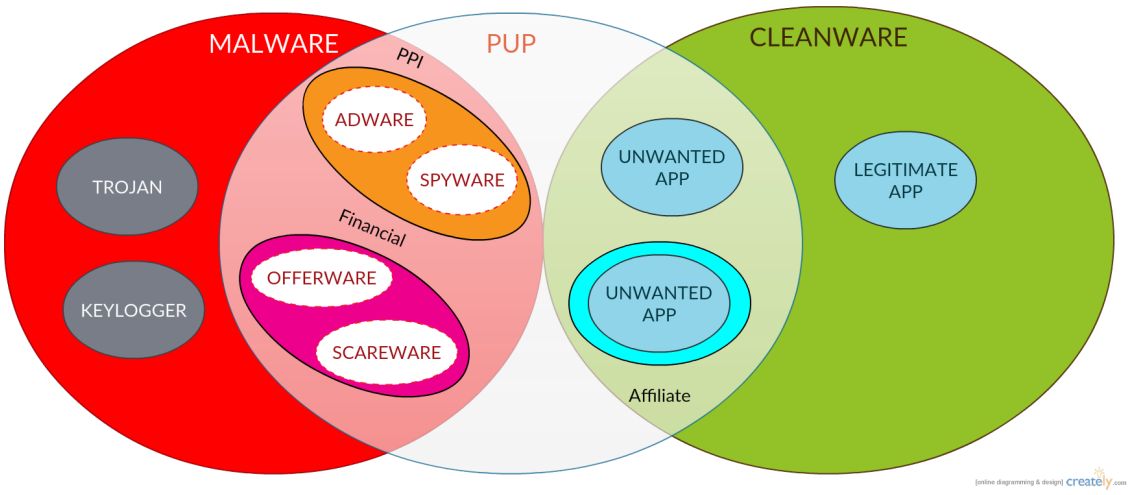


Figure 2.1: Relationship among PUP, malware and cleanware

A PUP may be benign, i.e. part of the cleanware class, yet not desired on the user's system. On the contrary, a large number of malicious software is classified as PUP, falling as well into the malware category. For instance, *spyware* programs are considered a type of PUP, aiming at stealing user's personal information [51, 74]. Figure 2.1 points out other classes of malicious software, categorized as PUP. Among those we find *adwares* and *offerwares*, which mainly impact on system usability [74] and user privacy. On the other hand, it exists a class of benign software that bundles third party software without explicit reason, causing the user to become suspicious about that.

An example of legitimate and potentially unwanted application is given by the Avast Antivirus installer, which - at the time of writing - installs the Google Chrome browser and will make it the default browser. Although it is possible to opt-out both these options, both of them are pre-selected. Beside Avast Antivirus might have legitimate reasons to install the Google Chrome browser, it still classifies itself as PUP installer, being the browser functionality not strictly needed in order to operate the protection services offered by the antivirus. A possible explanation would be the ability of logging SSL session keys not offered by Internet Explorer at the time of writing. Google Chrome offers the possibility of extracting the SSL private key when surfing the web, so that antimalware products may decrypt SSL traffic to spot dangerous websites. However, Avast Antivirus installs a root certificate on the system in order to intercept HTTPS traffic; so the key-export functionality is no strictly needed.

PUP Taxonomy

In the past years some research works have been done over the PUP phenomenon, providing an early attempt to classify the different kinds of PUPs according to their business model and general functionality. The result of these studies led to the following classification [63], also shown in Figure 2.1:

- Affiliate PUP
- Pay-Per-Install (PPI) PUP
- Financial PUP

PUPs are classified as *affiliate* when they offer a very minimal set of functionality, requiring the user to complete a registration through the vendor website, in order to activate the core features of the software. Moreover, those utilities are distributed through a network of affiliate partners, in which the installer party receives a relevant part of the sales as revenue. The second category regards the Pay-Per-Install (PPI) PUP. In general, PPI PUP are software installers which contain hidden malicious payloads, installed silently on the user's system. In this case, the installer distributor receives payments by the malware coder, based on the number and geographical location of performed installations. *Spywares* and *adwares* usually belong to this category. Finally, Financial PUPs represent a class of malicious software that offer zero functionality and directly interacts with the user, in order to steal credit card information. *Scarewares* and *offerwares* are part of this category.

2.1.4 Severity

We have already mentioned that identifying a PUP is not a trivial task due to the narrow limit that separates a benign software from a potentially unwanted application. So, in general, it is inconvenient and wrong to describe all the PUPs as threats for the users. However, based on the classification presented in 2.1.3, there is no doubt that a substantial part of PUPs means a threat for inexpert users. Financial PUPs and PPI PUPs are the most relevant threats, since they implement or bundle malicious software [63], such as spywares, adwares, offerwares and scareware. Those threats mainly target the user's privacy and financial data (user's asset), exposing the victim to potential frauds, as well as privacy loss.

An important consideration also regards the case of affiliate PUPs and benign unwanted applications. Let's assume an user downloads a benign utility that installs an unwanted antimalware on the system. Let this PUP be benign, but extremely poor in performance and threat detection. The program itself would be considered benign because it shows no suspicious behavior, however an important side effect is also obtained: the user may consider her system secure, while it is not. That will expose the user and his system to many threats. Another possible goal of benign PUPs is to press the user to buy the full-version of the software, claiming that only the professional version can solve a certain problem, even when it is obviously impossible. That pattern has been already discovered and pointed out by previous researches in this area, demonstrating how those PUP vendor may trick the user into frauds [63].

2.1.5 Profitability

The spread of PUPs is mainly justified by the existence of a business model that makes PUP distribution profitable. This model is called Pay-Per-Install (PPI), and in the last years has been used by malware distributors [8, 23].

The PPI distribution model is based on a three entity architecture: *clients*, *PPI service providers* and *distributors* (aka affiliates). Clients are software vendors (or malware coders) willing to install their products (payloads) on a large number of hosts. They pay a commission to the PPI service provider, that is proportional to the number of successful installations. On the other hand, PPI service providers represent the central orchestration points of this distribution model. They charges clients for every successful software installation and rely on affiliates to perform those installations. Moreover, PPI providers implement affiliate recruitment through advertisements, so that their network reaches a larger number of affiliates. Lastly, affiliates impersonate the distributors of the clients' software. By installing clients' programs on different hosts, affiliates earn revenues from the the PPI service providers.

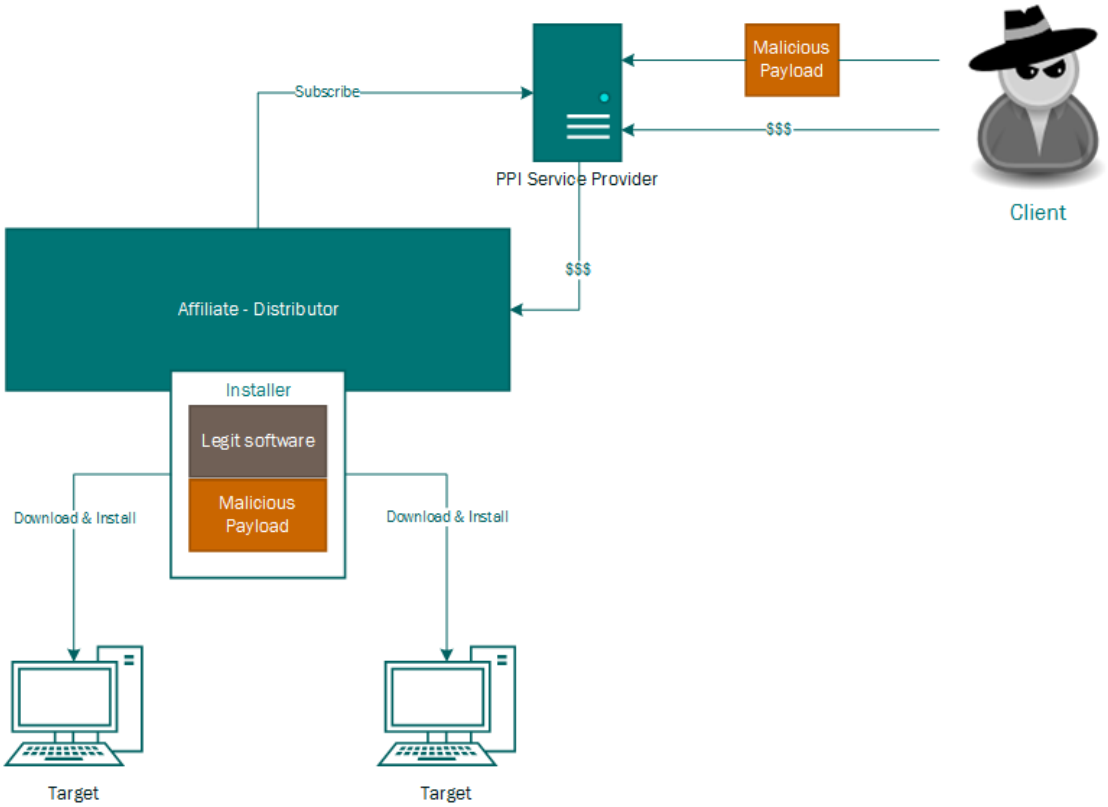


Figure 2.2: PPI business model

There are different possible ways for affiliates to install software into target hosts. One common approach is to bundle the payload along side a well-known and legitimate software, as shown in Figure 2.2. Once bundled into an installer, this package is usually distributed by *freeware* or *shareware* download websites. This technique allows the aggregator to get higher ranking and visibility on search engines, thus the possibilities of PUP spread increase. After the PUP gets downloaded, the installer may either obtain the user (not

informed) consent, by displaying a complex EULA, or may install itself silently. In the first case, the payloads may be categorized as spyware or PUP. In the second case, those payloads are generally malware, bundled through software binders and installed silently.

2.1.6 The EULA trap

Pay-per-install business moves a substantial amount of money. Thus, it is reasonable to assume PPI business runners may resort to legal courts whenever their products are categorized as malicious [20]. That is one of the reasons why antimalware products introduced the expression *potentially unwanted programs* [51].

Afraid of possible legal problems, many vendors carefully decide to whitelist most of *grayware*, although current antispyware technology is capable of detecting many different PUPs. In fact, some of those suspicious software (primarily adwares) are not explicitly prohibited by current laws, provided that the user is informed [27]. In these cases, the PUP installers present EULAs, in which the suspicious behavior is somehow justified. By accepting the EULA, the user fully authorize the software vendor to act as stated in that agreement. Once the user's consent is obtained, the software has the legal authorization to act as described in the agreement.

2.1.7 User informed-consent to dicriminate PUP

Some PUP vendors display EULAs, at installation time, in order to give the impression of installing fully benign software [28]. However, an EULA does not imply trustness by itself. In fact, some malicious PUP installers will rely on long (6000 words or more) and complex EULAs, in order to discourage the users from reading them. Nevertheless, malicious PUPs present EULAs written using complicated and specific legal terms, usually obscure to the final user [17].

In order to take an accurate decision, the user must read the entire EULA and must be able to fully understand it. When both of these circumstances are verified, then the user consent is valid. This situation is generally defined as *user informed consent* [19]. If the software obtains the user's informed consent, then it does not belong to the PUP class. On the contrary, when an application is installed without the informed consent, then it may be classified as PUP, or even malware. In fact, spyware and PUP try to get the user's not-informed consent [20]. This attempt, if successful, puts the spyware vendor in a legitimate position, legally protecting the software merchant.

It is worth noticing that previous researches have shown it is possible to detect classes of spyware (i.e. subset of PUP) simply analysing the EULA, displayed at installation time [21].

2.1.8 Habituation and blind approach

Being a legal contract, the EULA should be read very carefully by the user. On the contrary, some previous research works show that the general trend is subjected to habituation and heuristic shortcuts [19]. This phenomenon also evolves: when new user interfaces (UI) appear, users start learning new shortcuts to get their prime task done (i.e. installation), ignoring the possible implications that their blind interaction would produce.

In some cases EULAs are used as a mechanism to demonstrate false legitimacy of software installers. In this case, long and uncomprehensive text is shown at installation time [33]. Disheartened by the long reading, the users tend to blindly accept the EULA,

ignoring possible privacy consequences. This phenomenon is much more evident when the software being installed is somehow known to the user. However, when legitimate software is bundled as part of PPI installers, this approach is catastrophic: accepting EULA blindly basically turns into installing any software bundled within the PPI installer.

Beside the problem of the EULA has its fundamental roots in the user mis-information and habituation, previous researches have proposed a number of countermeasure for mitigating this problem [20]. All the proposed approaches, however, did not find any practical implementation at the time of writing.

2.2 Problem: PUP on Windows

The PUP phenomenon is not limited to a single operating system architecture. However, the spread of PUP is a major problem on Windows operating systems. For years Microsoft Windows has been the most used desktop operating system when surfing the web, nevertheless it still is the most popular at the time of writing (78.6 %) [92]. Therefore, PUP threat mainly targets Windows hosts, because they offer the most wide market for PPI business. For this reason, in this work we will focus on Microsoft Windows operating systems.

In order to better understand how the PUP phenomenon affects the Windows OS, it is necessary to consider some technical aspects of OS family. Thus, in this section, we will introduce some of the main challenges arising when dealing with PUP on a Windows OS.

2.2.1 Windows installers

Given an application, its installer is a software aimed at automating the deploying phase in a software engineering process. An installer generally takes care of copying executables files, data and configuration entities on the target operating system, in accordance with the OS version and architecture. Moreover, installers are also in charge of checking OS compatibility and fixing missing software dependencies. In the most general case, an installer is just a program, of which main task is to persist and configure the bundled application.

Microsoft Windows operating systems do not enforce any standard mechanism for installing a software. In fact, in the simplest case, an application is installed on the OS by copying its executable files on the host file system, sometimes providing commodity shortcuts on the user's desktop. Such a simple task can be achieved by a scripting file (e.g. batch files), or done manually by the user. When more advanced features are required, such as autostart on boot, the installer generally needs to deal with the Windows's registry. However, Windows registry has evolved during the time, preserving part of its legacy architecture for compatibility reasons [66]. This registry evolution has introduced confusion for software developers who want to target many different Windows versions. Moreover, being part of a common process (software production), it makes sense for most companies to automate, wherever possible, the way software is bundled. For these reasons, various free and commercial frameworks arose, such as InstallShield, Advanced Installer, Wix and Microsoft Installer. Their goal is to facilitate the bundling task for software developers, implementing a systematic deployment process. Therefore, these frameworks usually automate graphic user interface (GUI) generation, software removal procedures and software upgrade tasks. Each installer framework may operate in a different way: different frameworks may use various registry keys to achieve the same goal, and generally offer different installer GUI.

At the end of 90s, Microsoft introduced *Windows Installer*[93], a technology that defines standard Microsoft Installer file type (MSI) alongside *Windows Installer Service*[65]. The former defines a database-like file type, based on tables, used to describe the installation process of a software. This file type documents most of the common tasks needed to deploy an application, such as file copying, software dependencies, installer GUI, etc. Those MSI files are then processed by a Windows service, called Windows Installer, preinstalled on the operating system.

The goodness of MSI

Since the first release of Windows Installer, Microsoft has been encouraging developers to adopt this standard[2].

There are several reasons that justify this choice. In first instance, the MSI file format provides a clear organization of information regarding the deployment process. By implementing a database-like file structure, MSI packages contain tables grouping data and metadata, describing the most important phases of the software installation. By relying on such organized package structure, it is possible to analyze software components and features beforehand. This means that more sophisticated analysis techniques may be applied to detect potential anomalies, without running any code. Nevertheless, it is worth noticing that MSI packages are not executable by themselves. To install a certain MSI package, the operating system invokes the Windows Installer Service (MSIExec). This service is in charge of parsing, validating and installing data and programs contained into the MSI file. MSIExec provides then a simple and unified interface to manage operating system dependent matters, freeing the developer from complex version checks.

A relevant aspect of MSI data structure concerns the software removal process. Since MSI describes which files and registry keys are being installed, the MSIExec is able to implement automatic rollback operations. In other words the software uninstaller can be derived directly looking at the MSI package. In fact, the software developer is generally no longer required to implement custom software removal procedures, because most of it is handled by MSIExec service.

Custom executables

As already mentioned, Windows operating systems do not enforce any standard for application installation. Thus, any executable file is potentially an installer. This approach introduces many problems. First of all, the developer is in charge of handling direct deployment operations and has to deal directly with the operating system. As a consequence, the developer must consider compatibility issues among different version of operating systems, as well as handling software dependencies in its code. Therefore, the installer code grows (because of the OS checks, libraries to be bundled) and the effort of writing a good installer becomes substantial. Moreover, there is no automatic rollback feature (as offered by MSI packages), thus the developer should implement its own uninstaller procedure.

However, custom installers have no limits in terms of functionality, while MSI are limited to the MSIExec capabilities. For instance, custom installers can implement fancy GUIs (e.g. using hardware accelerated graphic) or executing remote code (RPC calls) and so on. In fact, an increasing number of custom installers do not bundle all the data hey need. Instead, they download the latest version of the software at installation time.

[2]https://msdn.microsoft.com/en-us/library/windows/desktop/bb204770(v=vs.85).aspx

In this way the achieve a series of goals. In first instance, the installation executable installer is tiny (hundreds of Kb or few Mb) and easy to maintain. Secondly, the installer may implement download session recovery, handling possible network problems. In third instance, software dependencies may be resolved in the same way, by downloading missing components on-demand.

Third party frameworks

The necessity of automating software's installers installation was already evident before the birth of the MSI file format. For this reason, products like InstallShield became very popular and widely adopted by software vendors (Flexera, distributor of InstallShield, claims 80% of the current Windows installations are built with its technology). This class of products represent a hybrid approach between the MSI standard and the custom executable building. They provide a setup building environment, where most of the basic deployment tasks are simplified. However, they rely on their own proprietary building process. Moreover, installer frameworks generally produce executable files, therefore it is impossible to accurately predict system changes without running them.

2.2.2 Where do PUPs fit the most

Although MSI provides a standard way of installing software on Windows Operating systems, still the majority of software available on the most known freeware distributors does not conform to this new format. Indeed, at time of writing, among the 484 applications available from most-popular rankings from FileHippo, Softonic and Cnet, 472 expose *.exe* extension, while only 12 are in *.msi* format.

MSI vs Custom executables

A PUP may be installed in any of the previous ways. Yet, malicious PUP would hardly be bundled into MSI packages. In fact, thanks to their descriptive structure, MSI packages are easy to be analyzed by antimalware and antiviruses, even without running the payloads. Moreover, also benign PUP applications (or grayware) tend to be dynamically downloaded by the installer at installation time, following the PPI model. For these reasons, without excluding the MSI packaging possibility, we expect the majority of PUP to be bundled within custom executables.

Another case supporting the assumption of bundled custom installer regards the malware distributors. It makes sense, for malware installers, to use code obfuscation techniques in order to escape malicious payload detection. In these cases, the most convenient installation process is the custom installer building, using binary *packers* or *binders*[8].

Installers usually require admin rights

Microsoft defines a set of best practices and guidelines concerning software deployment. According to those, software executables should be placed under `X:\Program Files` (where X is the current drive letter). However, placing software under this directory requires elevated privileges. Thus, the majority of installers, following those best practices, will require administrative rights. In fact, previously to Windows Vista, users were supposed to always run program installers within administrative accounts[69].

In an attempt of providing better protection against malware, Microsoft introduced the User Account Control (UAC) system in Windows Vista. This protection system implements the *least privilege* security concept within the Windows Reference Monitor. The UAC system allows processes to run with standard user rights, acquiring elevated privileges on-demand, after prompting the users for their consent. However, since its first release, the UAC suffered of too much annoying pop-ups, causing user habituation, vanishing any security protection[54] offered by the system itself.

The the legitimacy of an installer to obtain administrative rights, combined with user habituation to grant it, serves as great help for malicious PUPs. Indeed, PUPs acquire elevated privileges at installation time and can perform deep changes to the hosting system, becoming a potential threat for the system.

2.3 PUP detection state of the art

Potentially unwanted programs represent a relevant but yet new threat to the user systems. Thus, antivirus companies are nowadays developing and improving their detection techniques in order to be more competitive on the market. However, given their complex and blur nature, it is difficult to detect PUP in a reliable manner.

2.3.1 How antivirus detect PUPs

Over the years, malware companies have developed different technique to spot malicious software, but those are generally poor effective against legitimate PUP [53]. PUPs represent a new threat, carrying new challenges and high risk of false positive in detection.

In order to protect themselves against legal issues [27], some antimalware companies have adopted cautious approaches to detect PUP and alert the user. For instance, they heavily rely on user's feedback [53] and develop signature-based databases in order to increment detection rates. Companies like *Sophos* [78], *Kaspersky* [41], *Symantec* [26], have developed their own signature-based databases for PUP detection[78], allowing legitimate software vendors to claim legitimacy when their software is listed as PUP. Beside being a conservative approach, signature-based PUP detection is ineffective against 0-day releases. On the other hand, signature based detection leaves little space for false positive detection.

To beat the concurrents, antimalware companies do not publicly share their own malware or PUP definitions. While malicious software is generally detected by most of the top antivirus technologies, for PUPs we observe very different classification behaviors [53]. This difference of classification depends on divergent definitions of PUP adopted by each company. Moreover, the effectiveness of PUP classification also depends on the technologies involved by each antimalware product. For instance, Symantec introduced one heuristic in their SONAR engine, aimed at detecting potentially unwanted applications [85]. Heuristic based detection is a powerful weapon against 0-day releases and malware variation, but also produces an higher rate of false positives.

Other companies decided to adopt an hybrid approach. They let the user choose the level of protection desired and PUP detection is generally optional. Examples are the *ESET LiveGrid* and the *Sophos Live Protection* technologies[3]. When active, signature based detection and advanced heuristics provide some defense against potentially unwanted applications. In order to mitigate the false positive problems, those companies allow users

[3]These technologies collect behavioral data about suspicious objects, relying on realtime cloud analysis in order to detect 0-day malware spreading

to report any possible classification error, by submitting a whitelisting request to the antimalware company [45, 79]. This approach is nothing new: some mail providers use the same basics to classify spam [27].

2.3.2 Suspicious behaviors

Whether applying proactive or reactive detection mechanism (heuristics vs signature), antimalware companies rely on a set of suspicious behaviors in order to classify PUPs. Crawling among the different antimalware policies, we have found the followings being applied:

1. Advertising exaggerate claims about software capabilities, not matching real functionality
2. Not asking user consent or seeking for not-informed user consent
3. Changing default browser, system and network settings
4. At installation time, providing no EULA or too long and complex texts
5. Requesting unnecessary permissions
6. Implementing inadequate uninstaller procedures
7. Adding advertisement or threating false claims
8. Manipulating search Engine results manipulation
9. Using great amount of system resources
10. Program is not digitally signed
11. Providing remote access listening on the network
12. Downloading unexpected files from the web

Due to the continuos evolution of PUPs, the list of common suspicious behaviors may grow in the future. Some of them may seem harmless: exaggerating claims or providing inadequate uninstallation procedures do not constitute direct threats for the user. Nevertheless, having hard-to-remove software (which may represent part of attack surface for an hacker) on a system can lead to vulnerabilities. Moreover, the exaggerate and false claims may lead the user to fraud attempts, as proved by previous researches [23].

2.4 Malware analysis techniques to the rescue

Antimalware companies have developed, during the years, advanced techniques to detect malicious software. It makes sense for these companies to adapt the same techniques in order to detect PUP. In this section we will discuss those techniques, pointing out why they are useful and what are their limitations.

In general, it is possible to classify all the malware analysis technology within two major classes: *static analysis* and *dynamic analysis*. *Static analysis* is the act of inspecting the structure of a computer program, given its binary code, without executing it [75]. On

the other hand, *dynamic analysis* corresponds to the analysis of the software behavior performed at runtime, by using debugging (and usually virtualization) techniques. Both the approaches offer different pros and cons. Given the variability of malware and their fast evolution, dynamic analysis is the most valuable information provider, thus it is generally applied by antimalware researches [87]. Nevertheless static analysis is a good source of information as well and it generally requires less resources and security knowledge to be performed.

In our work we will mainly rely on dynamic analysis techniques, due to some critical aspects of PUPs. However some static analysis techniques are also applied during automated analysis. Therefore, it is necessary to introduce most important static analysis techniques to understand which are their limitations and how dynamic analysis solves them.

2.4.1 Static Analysis

Discussing static analysis technique is out of the scope of this work, however we will briefly introduce basic concepts regarding relevant aspects when dealing with PUP analysis. Among the most widely used static analysis techniques we find:

- Hashing
- Strings extraction
- PE Header analysis
- Linked libraries analysis.

Hashing

For hashing we mean the act of applying *hash functions* to data and binary files that we want to analyze. A cryptographic hash function is a mathematical procedure *hash()* that takes an arbitrary long input of bits and produces a fixed-length unique (generally small) sequence of bits (digest)[67]. Given the unlimited domain and the limited codomain spaces, no hash function is injective, thus no output is really unique. However, hash functions expose *collision resistance property*, which states *it is difficult to find two inputs m1 and m2 such as hash(m1) = hash(m2)* [77]. Other properties, relevant to hash functions, are the so called *pre-image resistance* and *second-image resistance*. The former is formally declared as the difficulty, given a hash function *hash()* and a hash value *h*, to find any message *m* such *as h = hash(m)* The *second-image resistance* describes the difficulty, given an input *m1* and a hash function *hash()*, to find another input *m2* such as *hash(m1)=hash(m2)*.

Hashing is used by malware analysts in order to store digests of malicious binaries. This technique drastically reduces the space needed to store those information (sometimes this approach is addressed as *data reduction*). Moreover, it still provides comparison criteria: instead of performing byte-to-byte comparisons, files are hashed and then digests are compared. One relevant drawback of classic hash functions is that they do not measure how similar two inputs are. A single bit changed in the input produces a completely different output. For this reason, recent research works developed *Context Triggered Piecewise Hashes* (CTPH, a.k.a. Fuzzy hashing) [42, 43].

Fuzzy hashing, combined with standard cryptographic hashing, is nowadays particularly relevant for forensic analysis [31]. CTPH functions are based on a fixed-length sliding window, on a state and on trigger function. A sliding window (generally small, few bytes) is

used to roll over all the input. At each step, a hash function is calculated based on the bytes contained in the input at that time, then the input is moved forward by 1 byte. Once the window is moved, a trigger function is evaluated. If that function meets some pre-defined criteria, then a partition is defined. By reiterating the algorithm for the whole input length, we will obtain a set of partitions for the input data. At this point, the CTPH functions uses classic cryptographic hash functions (e.g. MD5) over the identified partition. By concatenating partial parts of the obtained hashes, the final fuzzy-hash result is obtained.

Fuzzy hashing exposes two important properties, i.e. *non-propagation* and *alignment robustness* [88]. The first property defines the resilience of the output to change, when little modification is performed on the input. Opposed to the cryptographic hashing functions, when a few bits change on the input, partial input modification will not affect the whole output. The second property states that CTPH algorithms are robust to shifting or padding operations. Again, those operations would cause drastic output changes when used with classic cryptographic hashing functions. It is worth noticing, however, that CPTH algorithms do not produce fixed length hashes. Output length depends on the partitioning function and on the input size.

String extraction

Binary and executable files may contain much information in different forms. Part of that is represented by text, usually bundled within the executable file and encoded with ASCII or Unicode. Information in text form can disclose several potential behaviors of the binary file, thus it represents a valuable source of information. Naming some of the possible information, we identify:

- Host names
- Http Urls
- IP Addresses
- EULA
- Dynamic Libraries
- Registry key names
- File Paths
- UI messages

Several technique may be applied in order to extract that information out of a binary file. One simple approach is to scan the whole binary input in accordance with a specific character set. Then, only detected strings that respect pre-defined boundaries are saved. For instance, we might want to detect all the ASCII strings at least 8 characters long, and at maximum 200 characters long.

Figure 2.3 shows a real-world example taken from the Avast Antivirus downloader[4]. In order to inspect the binary data, we used a web-browser HEX editor `https://hexed.it`. Scrolling the binary source at address *0x000A2FC7* we find some information in ASCII encoding. Some urls are listed among those values. That suggests the ability of the software

[4]MD5 hash: bf77838a15ae4e72d3438a0693b629e9, version 11.1.2245.1540

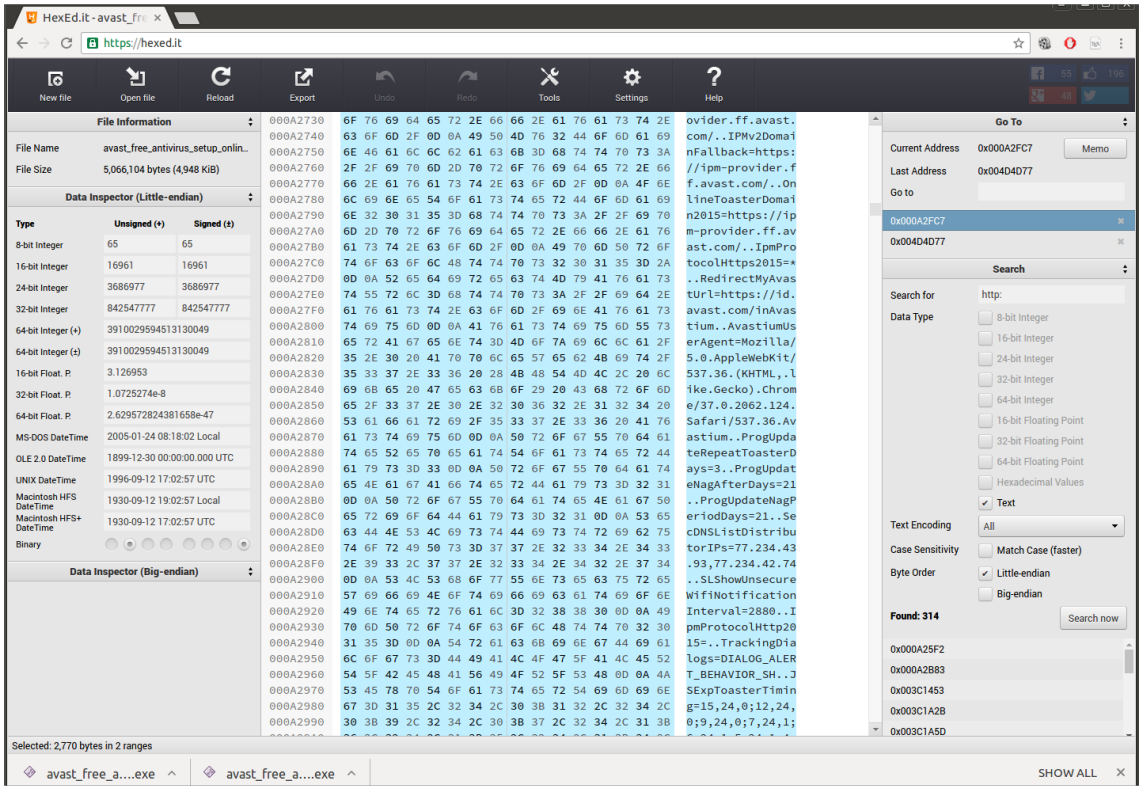


Figure 2.3: Avast Antivirus Installer HEX binary data

to contact those urls. If any of those URLs were blacklisted or known to be malicious, then the program itself would be suspicious. However, it is worth to stress that information extracted by string analysis does not imply the action will happen at runtime. Nevertheless, that information may be used as indicator for possible threats exposed by the binary program.

PE Analysis

On Windows NT, executable files follow the *Portable Executable* (PE) specification [38]. A PE file is composed by 5 parts and is called *module* (or *hModule*). The first one is a generic *DOS header* (64 bytes), containing general and legacy metadata, only relevant for *DOS systems*. The next part is the *DOS stub*, i.e. a portion of the executable which can be run under a DOS operating system. Although this part is generally irrelevant for Win32 applications, it is common behavior of compilers to put legacy code that displays an error message if the executable is run under DOS. Right after the DOS Stub, there is the PE Header portion. This area of the PE contains metadata describing the following part of the executable file. After the header we find the section table, i.e. a data structure mapping the rest of the PE file into sections. Each section can either contain code or data, and there may be many of them.

In Windows NT, a PE file must exposes 9 different sections, as showed in Figure 2.4.

Executable — *.text* or *CODE*, contains the code segments which will be loaded into paginated memory and then executed by the OS

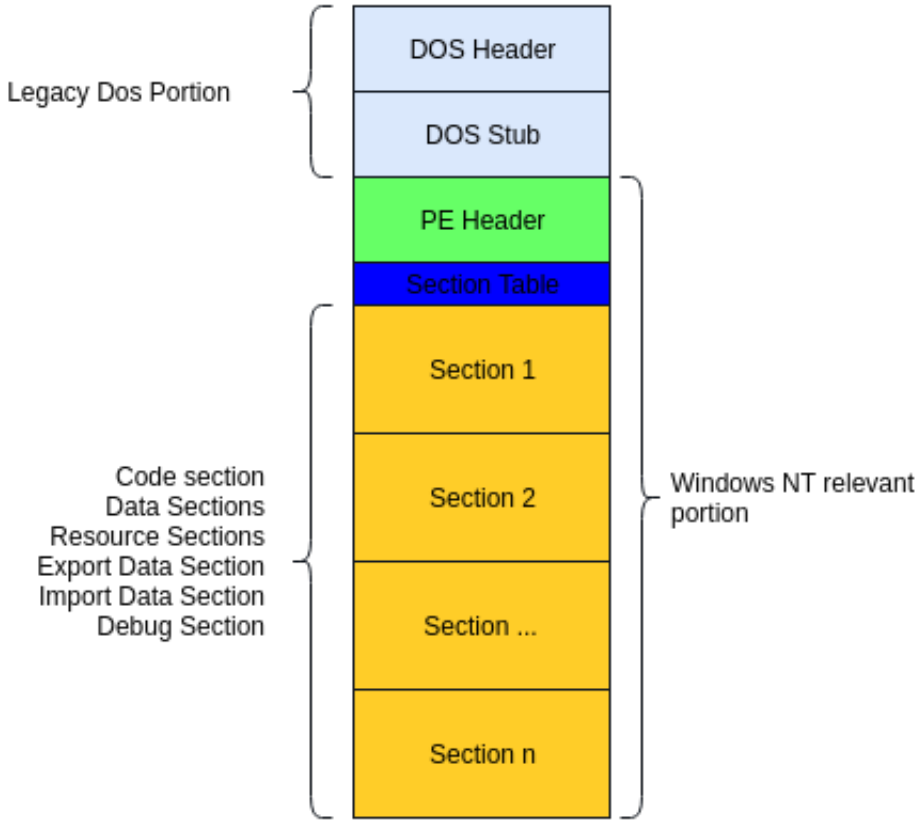


Figure 2.4: Structure of a Portable Executable file

Data — *.bss* and *.rdata*. The former contains uninitialized data for the application (such as static variables), while the latter contains readonly information, such as literal strings or constants.

Resources — *.rsrc*, this section includes resource data, usually icons and images, needed to the application.

Export — *.edata*, exposes the address of exported functions contained in the module

Import — *.idata*, contains information about imported functions

Debug Information — *.debug*, includes some debug information.

Relevant information may be extracted by the resource section. For instance, in a PUP installer, several files (binary and not) may reside in this section of the module. By extracting bundled files and using hash functions, we might immediately identify possibly known malicious payloads within the module.

The read-only data section usually contains strings and constants used by the application. However, string extraction is generally applied to the whole binary file and not limited to a single section. On the other hand, there may be cases in which the string extraction procedure can be limited exclusively to read-only data segment. In this way, the probability of gathering invalid or meaningless strings may be lower in respect to the techniques described in 2.4.1.

Export and Import sections are important for static analysis too. Modules can export functionality to other applications, or may require some functionality exported by other modules. Either the cases, it is possible to identify a set of imported or exported functions that a module requires to run correctly. For instance, an unpacked *keylogger* might import *SetWindowsHookEx* (to install keyboard hooking) from `User32.dll` and some file manipulation functions such as *WriteFile* from `Kernel32.dll`. At the same time, the analyzed module may export *LowLevelKeyboardProc* and *LowLevelMouseProc*, used by the *SetWindowsHookEx* to notify the application of inputs received [75].

Other information lays in the PE header. *TimeDateStamp* is a data field, belonging to this header, representing the time and the date when the module has been produced by the linker or compiler [64]. When an executable file is old, the possibility to find signature for it is generally higher [75], so this information becomes relevant.

Library linking

Imported functions listed in the PE header do not always represent the totality of referred functions. In fact, functions used by a module may depend on the linking policy chosen by the developer. Windows operating systems support three different policies of linking: *static linking*, *dynamic linking at loading* and *dynamic linking at runtime* [62].

Static linking means the act of coping the referred code from the library into the module, increasing the compiled module dimensions. This approach is generally used when size of the module is not critical and when developers wants to avoid code dependencies problems. On the other hand, dynamic linking defers the linking procedure, producing smaller modules and allowing code re-usability. Moreover, compilation time for the module is shorter because linked code does not need to be compiled. Dynamic linking may be performed in two different moments: either at loading time (dynamic linking at loading) or at runtime (runtime dynamic linking). The former method requires the Windows Loader to check, once the program has been loaded into memory, if all the requested libraries are available on the system. In case of missing libraries, the operating system refuses to run the program. The latter case, instead, requires the developer to handle dynamic linking within the code, by using ad-hoc procedures, such as *LoadLibrary()* and *GetProcAddr()*.

When static linking is adopted, it is hard to identify which library has been bundled into the executables. However, PUPs generally rely on dynamic linking in order to produce lighter modules. Furthermore, when loading the required libraries ad load-time, the operating system needs to know what kind of functions have to be linked. Those functions are listed in the Import section of the PE file, and are pretty easy to identify. To do so, we only need to look into the *export* section of the PE file and interpret the relative data structure contained. On the contrary, when the dynamic linking happens at runtime, there is no explicit area - in the PE file - where those functions are listed. In fact, the developer uses the LoadLibrary() functionality of Windows in order to load an external module at any place in the execution flow. Then, using GetProcAddr(), the developer is able to retrieve the offset of the function within the module. At that point, the function can be invoked.

String extraction represent a last resort against runtime dynamic linking. In fact, by scanning the code section of the PE file, it might be possible to find linked function names and modules. Most of the common operating system modules and functions are known, so it might be possible to match extracted strings with a list of known functions and module names.

Disassembling applications

One more advanced technique among the static analysis tools is the disassembly process. Without running the application, an analyst may decide to reverse-engineer the binary, trying to interpret the meaning of the code. By using ad-hoc software, such as *IDA Pro*[5], an analyst would be able to translate binary code into assembly. After that, looking at the code, it might be possible to recognize some behaviors of the software.

In order to facilitate the reverse engineering process of an executable, some tools interpret assembly code and produce constructs in higher level languages, such as C/C++. However, without debugging symbols, the generated code is hard to read, and hardly matches the original source code structure.

Static analysis limitations

Static analysis techniques are able to extract a relevant set of potentially useful information about the executable file. However, new grayware introduced many challenges, not handled by static analysis.

Particularly challenging are *code obfuscation* and *packing* techniques. When dealing with PPI PUPs and malicious payloads, distributors generally obfuscate their code [8]. Packing is a manner of hiding the binary code into a packed format, so that static analysis cannot directly extract information out of the file. A relatively small portion of code is left in clear form: that part of the code is in charge of extracting (or *unpacking*) the rest of the code into memory, before executing it. Once the file is executed, the Windows Loader extracts the data and runs the unpacked code. The rest of the code is then unpacked by that stub and, later on, executed. A possible way of achieving this goal would be by encrypting the code to be executed, and let the unpacker decrypt it at runtime. The decryption key may be retrieved through network, or would be bundled into the executable itself. In this case, static analysis would not be able of extracting all the information such as strings and dynamic linked libraries, because most of them would be available only after the unpacker code runs.

Packed and obfuscated software also make it difficult to process reverse-engineering without executing the program. When running a disassembler, the first instructions of the unpacking process are visible: an analyst would recognize the software is unpacking itself. However, the unpacked code would not be visible until the unpacking routine runs. Therefore, it is close to impossible to predict, a priori, the final outcome. Moreover, human interaction is needed to perform this step, and it generally takes a considerable amount of time.

Another limit of static analysis comes to light when the PUP is composed by a simple network downloader. In this case, the software components to be installed on the system are retrieved over the network, at installation time. Therefore, execution of the downloader has to precede the time of analysis . However, this is not possible, according to the definition of static analysis itself (binary execution should not happen).

2.4.2 Dynamic analysis

Dynamic analysis is defined as the inspection of an application, during its execution [75]. The main goal of this approach is to go beyond static analysis limits, especially when dealing with obfuscated or packed software. In the most general case, a dynamic analysis

[5] https://www.hex-rays.com/products/ida/

technique is represented by a debugging session: in that way an analyst can observe, step by step, all the modifications performed on the system by the analyzed program.

The main difference between static analysis and dynamic analysis resides in the execution of binary code. While static analysis does not require program execution, dynamic approaches require code execution. Therefore, caution is needed when executing possible malicious code, in order to avoid malware infection. In particular, *isolation* and *revertability* must be enforced.

Safe execution environments

A test environment where to run unknown software is named *sandbox* [95]. A sandbox takes into account risks and threats possibly exposed by the unknown software, thus three main properties characterize such an environment: *observability*, *containment* and *efficiency* [95]. The first property states the ability of detecting actions performed by the unknown application within the sandbox, such as file system and network IO. Good observability generally implies much grained granularity in auditing systems. *Containment* is the ability of the sandbox to prevent attacks to hosts external to the sandbox itself. Lastly, *efficiency* measures how good and relevant the extracted information is, in comparison with the execution time.

In order to implement sandboxes, *virtualization* is often used. Virtualization is a technology aimed at abstracting applications from the underlying supporting hardware, by providing a convenient Hardware Abstraction Layer (HAL), implemented by an *Hypervisor* [44]. In other words, virtualization decouples software components and hardware infrastructure [32]. One of the main goals of virtualization is to maximize hardware resources utilization, by multiplexing several virtual machines on a single hardware infrastructure. Flexibility and portability are also relevant goals for this technology: providing an abstract hardware layer, virtual machines can be moved (i.e. migrated) from one hardware infrastructure to another one, in a relatively short time.

Other than resource sharing, any virtualization technology offers *isolation* among virtual machines [73]. Software running on a virtual machine (vm) may not interact with software running on a different vm. By implementing isolation features, virtualization improves reliability and security, thus containment of the sandbox is enforced. Moreover, deep inspection of software behavior is possible either with Hypervisor support [52] or thanks to hardware accelerated support (i.e. Intel VT-d[6]) [48].

Virtualization also offers a convenient way of starting the analysis by a known state. By configuring a virtual machine and saving its state in the form of a *snapshot*, it is possible to repeat the analysis multiple times, always starting from the same known state. In this way, results are comparable and it is possible to perform the same analysis even when the underlying hardware changes, without affecting the VM configuration.

Virtualization drawbacks

Being a convenient way of implementing sandboxes, virtualization is being largely used in grayware and malware analysis. Therefore, virtual environment detection systems (a.k.a. *anti-VM*) have been implemented by malware developers, and are widely used by deployed malwares [75].

[6] https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-in

Company and Products	MAC unique identifier (s)
VMware ESX 3, Server, Workstation, Player	00-50-56, 00-0C-29, 00-05-69
Microsoft Hyper-V, Virtual Server, Virtual PC	00-03-FF
Parallells Desktop, Workstation, Server, Virtuozzo Virtual Iron 4	00-0F-4B
Red Hat Xen	00-16-3E
Oracle VM	00-16-3E
XenSource	00-16-3E
Novell Xen	00-16-3E
Sun xVM VirtualBox	08-00-27

Table 2.1: Known virtualization vendors codes

The goal of anti-VM is to inspect the underlying system in order to find some proofs of being ran on a VM. This goal is achieved with heuristics targeting vendor-specific products, but also general applicable techniques exist. For instance, when a specific *toolset* (VirtualBox Guest Additions, VMWare Tools, etc.) is installed on the Guest OS, an application may simply detect the existence of those services running on the background, or scan the registry for known key locations. Even when no toolset is found on the system, application may recognize known driver signatures installed on the Guest OS. Another simple way of detecting a virtualized environment is to rely on the manufacturer (OEM) portion of the MAC address of the network interface card (NIC). Table 2.1 shows some of the most known virtualization vendors OEM MAC portions, that might be interpreted as proofs of virtualization environment.

Beside the anti-VM techniques exist, the popularity of countermeasures seems to grow down [75]. The main reason is that malware coders are now considering virtualization as a much more common technology, often used even by consumer users. Thus, virtual machines may still represent valuable victims to be infected [75].

Another drawback of virtualization consist into the overhead caused by the hypervisor and by the resource sharing among VMs. For instance, by running multiple IO-bound processes on different VMs, the IO capacity of the hardware media is overstressed. This phenomenon causes a bottleneck that slows down all the VMs. In general, performance degradation is strictly dependent on the underlying hardware infrastructure, hypervisor architecture and number of VMs simultaneously running. This aspect is relevant when considering the efficiency property of sandboxing: the time of the analysis may increase proportionally with performance degradation.

Live Debugging

A debugger is a software or hardware tool aimed at examinating the execution of a program, by inspecting its internal state [75]. By using a debugger, a software developer is able to check the execution of binary code, instruction by instruction, in form of assembly code (*low-level debugging*). Moreover, the developer has full control over memory and registers of the system, being able to modify their values while running the program itself.

Originally debugging was meant to spot potential errors into software components. However, malware analysis uses the same approach to investigate what a piece of software does. Security analysts use *low-level debuggers* in order to detect the program behaviors, which are hard to spot with simple static analysis.

Live debugging provides the biggest source of information when performing software analysis, since it represents the deepest inspection technique applicable. On the other hand, the debugging process is completely manual and requires security expertise interaction. For this reason, although this technique is useful to analyze particular instances of malware, it is not applicable to PUP analysis.

Automated Asset Monitoring

In order to perform analysis on a large number of software components, automation is needed. Thus, debugging is not a viable option in this case. Instead, monitoring tools represent a convenient way of gathering low level information about software execution; moreover it can be easily automated. Before presenting existing tools and technique to audit resource accesses, it is important to identify which are crucial resources we want to monitor.

On Windows OS, a program may interact with several resources exposed by the operating system. Particular relevant assets are:

- File System
- Registry
- Network

As we will discuss later in Section 6.2, there are multiple ways of accessing system resources. On Windows, a particular resource may be accessed through different level APIs. Some APIs are performance oriented, thus implement a very strict set of operations, providing low-level access to the resources. That is usually the case of device *drivers*. On the other hand, the OS also provides high-level access to the resource, reducing the developer's pain, at expenses of performance.

File System. The file system (FS) consists in the logical abstraction of hardware storage devices [76], implemented by the operating system. Thus, it provides access to data and system files both to the applications and to the operating system itself. In other terms, the FS exposes a homogeneous view of underlying hardware devices, which may vary in terms of technology used. This abstraction consists in a set of files and a directory structure, mapped on hardware device sectors, cylinders and blocks.

By interacting with the file system, an application may impact on the system behavior in a permanent way. For instance, an application may overwrite a system executable, affecting the operating system behavior. Moreover, an application may read and leak user's data, by using network connections.

Registry. The Windows registry represents a system-wide database, containing settings controlling the OS behavior, alongside per-user configurations and system information [71].

By editing the registry, it is possible to control system boot options, decide which programs have to be auto-started at system boot, configure the windows firewall, etc.

Network. A process that interacts with the network may perform unknown operations on the system, which are hardly guessable a priori. For example, malware may download and install other malware, or might wait for commands before acting in a certain way. Moreover, malicious software with network access might leak personal information, compromising user's privacy.

Monitoring tools

A process may interact with system resources at various levels, therefore access auditing may happen at different layers. In general, we might identify the following possible logging approaches:

- Hardware level
- Kernel driver level
- User space level

Hardware tools

The lowest level approach consists in inspecting media access at hardware level [81]. This goal is achieved by interconnecting specialized devices among hardware components of a system. Network monitoring is often performed using hardware inspection tools.

Hardware inspection offers some key advantages in respect with other logging techniques:

Inescapability	No software component will be able to elude the logger, since every access to the media is intercepted at hardware level.
Detail level	Maximum detail level at hardware layer, skipping every abstraction layer offered by the OS.
Performance	System performance is not affected by hardware loggers, since no system's CPU cycle is wasted in logging those accesses.

On the other hand, hardware logging suffers from following drawbacks:

Verbosity	Collected data may be too verbose and usually needs to be aggregated to provide meaningful information
Encryption	End-to-End encryption frustrate hardware logging.
Costs	Hardware loggers are expensive.

Software tools

Logging at software level is a pretty common technique, natively implemented in Windows OS (e.g. *event viewer* and *performance counters*) and in antivirus products. System monitoring may happen at two different stages: kernel level or user level. Kernel level monitoring permits to log the totality of system resources at the lowest software layer, regardless of the actor accessing the resource (processes, OS). As a consequence, privileged access is needed to perform kernel level monitoring. On the contrary, user level monitoring is much more restricted. In fact, this technique only allows to log actions performed by the *current principal* over non-privileged resources.

An example of kernel level monitoring is a *File System Filter Driver*[7]: every access to the FS will be passed to the driver, which may log it somewhere on the system (or in memory).

An example of user level monitoring is given by RegMon[8]. RegMon is a simple utility which takes a snapshot of the entire Windows Registry and is able to compare it with other possible snapshots, calculating differences between them. When launched without administrative rights, the utility will be able to intercept all the information visible to the current logged principal.

Among the most used monitoring tools for Windows operating systems, we find the *Windows Sysinternals*[9] utilities. In particular, *Process Monitor* represents a tool combining many legacy utilities, aimed at monitoring FileSystem IO, Registry Accesses and Network operations in real-time [75]. According to its implementation, Process Monitor installs kernel-drivers in privileged mode and collects data into volatile memory during system execution. Beside being very powerful, this tools suffers from high memory consumption when no logging filter is applied.

2.5 Malware Sandbox Analysis Systems

Malware Sandbox Analysis Systems (*MSASs*) consist of security-minded sandboxes, aiming at providing a safe environment where to analyze unknown binaries [95]. Therefore, MSASs generally apply both software and hardware auditing techniques[10], taking full advantage from cloud technology.

MSASs can either be Internet-connected or network-isolated. When a sandbox is Internet connected, the running software is able to communicate with the external network, thus potentially acting as an infected host. On the other hand, isolated sandboxes only provide emulated network, so that no interaction is really routed over the Internet. Software may behave differently according to the connectivity capabilities available within the sandbox. For instance, some applications may act only in response of commands received from a remote host. According to the recent malware evolution, Internet connectivity is becoming a strong requirement for many classes of malware, and becomes absolutely necessary for many kinds of malicious PUP (offerware, scareware, etc).

MSASs offer a great resource when combining hypervisor automation capabilities with dynamic analysis monitoring tools. It becomes possible to automate the information collection process. Once information is gathered, it may be presented in a structured form, such as a human readable report. This approach has become rather common lately, giving birth to *Public MSASs* [95]. Those services aim at providing public and freely accessible cloud-based MSAS, which analyze user provided executables. Their goal is to provide a structured report about software behaviors, detected during the sandboxing process, stressing potential maliciousness indicators.

Public MSASs might apply both static and dynamic analysis techniques to uploaded software, and may provide emulated or real Internet connection. Each sandbox might also provide different environment configurations, such as Guest OS version, hypervisor type, maximum test durations, etc.

[7] https://msdn.microsoft.com/en-us/windows/hardware/drivers/ifs/introduction-to-file-system-filter-drivers

[8]

[9] https://technet.microsoft.com/en-us/sysinternals

[10] Hardware logging in virtual environment consist in software logging within the hypervisor

At the time of writing, two of the most known public and free MSASs are malwr.com (based on *Cuckoosandbox*[11]) and hybrid-analysis.com (powered by *Payload security*). Others, such as *Anubis*[12], have turned into commercial products (e.g. *LastLine*), after being public long enough to gather data from the samples submitted by users.

2.5.1 Limitations

Classic malware analysis techniques are able to provide valuable information about software behaviors. Nevertheless, PUPs present some unique characteristics, very different from classic malware (e.g. they generally offer an UI). Also, PUPs installer are continuously evolving and evading signature detection, vanishing the work of classic dynamic analysis through low-level debuggers.

In order to respond to the growing phenomenon of malware evolution, public MSASs try to automate part of the analysis. However, at the time of writing, we did not find any public sandbox able to interact actively with offline or web installers. Just to give an example, by submitting the Avast Antivirus online installer to Malwr.com, we obtain a poor report, which does not include most of the basic information we might expect. Just to mention one, the Avast's executable, which is dropped upon successful installation, is not listed among the dropped files[13]. Also, screenshots of the analysis clearly show the installation process did not go beyond the first two steps of the procedure. The problem resides in the poor capabilities of the UI interaction offered by the MSAS, causing the anslysis to time out when the UI asks for user's input.

MSASs usually provide uncorrelated pieces of information. For instance, malwr.com enumerates the *dropped files* into its section and, separately, the list *HTTP requests* into another section. In terms of analytics, it would be crucial to correlate a dropped file to the relative HTTP request, thus to its source host address.

Finally, MSASs rely on virtualization technology, which may be ineffective against certain type of malware or PUPs. Beside malware and PUPs are loosing interest in evading detection in virtual environment, it may still be convenient to rely on a hybrid sandboxing approach, capable of using bare metal hardware.

2.6 Automated GUI interaction

A relevant part of our work focuses on automating the PUP installation procedure. However, being nothing more than computer programs, PUPs usually show graphical user interfaces (GUI or simply UI). During the installation process, the user is supposed to interact with graphical elements provided by the installer. By doing so, the installer is configured and the installation process begins. Therefore, it is crucial to find a convenient mechanism aimed at automatically interacting with the GUI of a PUP installer.

In this section we will briefly introduce the basic concepts of the Windows operating system. Although most of the basic architectural concepts of the UI system are common to any modern Microsoft OSes, we will focus on Windows 7.

[11] https://uckoosandbox.org
[12] http://anubis.iseclab.org
[13] https://malwr.com/analysis/M2IwYjE3NDVlM2QxNDk1ZGExODQxZjFkM2JiNzA5MTQ/

2.6.1 Windows UI Architecture

Every GUI-enabled application on Windows 7 is based on two fundamental concepts: *windows* and *messages* [55].

The windowing system, used by Windows 7, is based on an object oriented architecture, inspired by code modularity and reusability. The base class, that every stand-alone graphical element must extend, is the *Window*. In the most general case, a window just identifies a portion of the screen. Each window may contain specialized child windows (subclasses of Window), implementing particular graphical widgets. Those widgets are called *child window controls* [62]. Examples of specialized window controls are textboxes, checkboxes and buttons. Thanks to object orientation and to window modularity, the Windows OS supports hierarchies of windows: each control may have a parent and children.

The Window class implements an interface exposing the `WndProc()` function [56], formally named *Window Procedure*. Beside representing the most important function of the Window class, the window procedure is inherited by all the sub-classes, which may specialize it. The main objective of `WndProc()` is to handle special pieces of information, called *messages*, which are dispatched to every window by the OS. Messages correspond to system's events, that windows might want to process [55]. In particular, they need to handle inputs event from the users and special messages from the OS. In fact, user's interactions with the windowing system happen through input devices, such as keyboard, mouse or touch devices. Then, the operating system translates those inputs in events, that are dispatched to relative windows in form of messages [62].

As an example, we might refer to a very basic use case: button click. Whenever the user moves the mouse cursor over a button and left-clicks, the operating system generates a sequence of messages describing both the cursor's movements and the click event. Those messages are dispatched to the main window (containing the button) and to the button itself. Therefore, the WndProc procedures exposed both by the containing window and by the button are invoked. In order to let the user understand that the button has been pressed, the WndProc of the button usually repaints the area occupied by itself with a new pressed button aspect. Once the system detects the mouse left-button release event, than another message is delivered to the WndProc of the button, which will paint the area again, making it appear unpressed.

Given the object orientation of the UI system and the re-usability of most of the UI elements, each window has to register its class to the UI system. This operation happens thanks to the windows RegisterClass() API function, which takes as argument a structure describing the window's class. Among the parameters of that structure, the most important is represented by a pointer to the WndProc() procedure associated with the class itself. Once the class has been registered, the program can allocate one or more windows based on that same class, through the CreateWindow() function. All the window instances, based on that class, share the same WndProc() implementation.

2.6.2 Low level UI interactions

The interaction mechanism between user's inputs and UI elements is based on the message exchange system. To support this kind of approach, the Windows operating system allocates one system-wide message-queue and one message-queue for each application's UI thread. Whenever an input event is triggered by an input device, its driver converts that input into a message, which is sent to the system's message-queue. The operating system implements a dispatcher mechanism that continuously looks into the main system queue. Once a

message is found, the OS examines it and forwards it to the queue of the targeted window's thread. Each UI thread, then, implements a message-looping architecture [62], so that each message gets popped out of the queue and processed by the associated WndProc(). This message exchanging mechanism is defined *posting*: the OS (or another application) may *post* a message to a window by calling the *PostMessage()* function and specifying the system-wide identifier of the target window (also known as window's HANDLE). The post operation is asynchronous: as soon as the message is stored in the target's queue, the control returns to the caller.

A second way to interact with a window is by *sending* messages. In this case, the message is passed directly to the target's WndProc function, which will react right away, skipping the message queue. This operation is synchronous: the caller will wait until the message is processed.

The decision whether sending or posting a message is left to the application developer. However, the OS itself uses the two methods in different cases. For instance, both mouse clicks and key strokes events are posted to the queues. This decision strincly finds its motivation in efficiency and caching. On the contrary, the OS sends a message to the interested window when the user wants to focus it. Therefore, this operation is executed synchrounously, and may block the UI in case the WndProc() takes much time.

2.6.3 Basic Win32 UI messages

Windows defines a very large number of possible messages that windows can exchange. Many of them are meant to support the standard GUI system, while others are left for user's defined functionality[55]. Messages are used by the OS even for the very basic functionality of the controls, such as element creation, painting, and destruction. In fact, the whole life-cycle of any UI element is handled by its WndProc() function, handling messages received from the OS or from other windows. Each basic UI element should, at least, handle the following messages in its WndProc():

- WM_CREATE
- WM_PAINT
- WM_CLOSE
- WM_DESTROY

When the program wants to create an instance of a window (or a control), it calls the CreateWindow() function [62], after registering the base class of that window. Then, the OS sends a `WM_CREATE` message to the relative WndProc() function, which consequently provides appropriate functionality in response to that message. In the simplest case, the WndProc() handles `WM_CREATE` messages just allocating memory necessary for the control. At a certain point of the execution flow, the UI thread will need to show the control it has created before. This is done by invoking the *ShowWindow()* function, which will cause the OS to send another message to the component: `WM_PAINT`. In response to this message, WndProc() performs all the needed drawing operations necessary to paint the control on the screen's surface. Finally, when the UI thread exists and the associated resources have to be released, the `WM_DESTROY` message is sent to the relative WndProc() function. In this case, the handler is in charge of releasing all the resources previously acquired, returning them to the system.

The list of messages, handled by every component, may vary in length. For instance, many UI elements are able to handle the `WM_SETTEXT` message: labels and textboxes respond to this messages by showing the text passed as parameter of the message. Other elements, such as ComboBoxes or ListViews generally handle `XXXX_SETITEM` and `XXXX_GETITEM` messages (where XXX stands for a component class dependent prefix), in order to react to selection events.

2.6.4 Windows UI libraries

So far, it is evident that even a simple GUI-enabled application causes a lot of overhead for developers, when using low level Windows API to deal with UI elements. However, many programming languages and UI frameworks rely on UI libraries, which facilitate the usage of the UI system, providing commodity high-level APIs.

For instance, the Microsoft .NET framework implements the so called *Winforms library*[61]. This library provides many different widgets ready to use, and provide commodity high-level api to facilitate their usage.

There are also multiplatform UI libraries, providing similar sets of elements. An example is *Qt*[14]. Qt is a software framework, using C++, aiming at building cross-platform GUI applications [18], providing libraries for different operating systems and various programming languages.

Another framework which is gaining importance nowadays is *Sciter*[15]. This multiplatform GUI framework enables native UI support for HTML5 based frontends. In other words, Sciter allows developers to use HTML5 and Javascript programming languages to define the UI appearance of an application frontend. The framework also uses advanced hardware capabilities of the hosting system to show powerful UI animations, providing low level Win32 API translation.

Each UI library mainly has two objectives. The first one is to provide an high level interface for the developers, reducing the overhead due to the complexity of native Win32 GUI APIs. Secondly, each library aims at providing good look and feel for applications, enabling new visual effects. Figure 2.5 gives an idea on how much different two UIs can be when built via different frameworks. In the proposed images, the top screenshot represents the first window that appears when installing *Avast Antivirus*, built with the Sciter library. On the other hand, the image at the bottom represents a more classic installation window. In the latter case, the *Internet Downloaded Manager*'s installer uses classic Winform elements to build its UI.

Windows does not enforce any strict implementation constraint for custom controls and windows, leaving maximum freedom to the developer. Nevertheless, each library implements its widgets in a distinc manner. For instance, different UI frameworks may rely on custom messages to implement standard functionality and ignore certain messages. For example, a framework might provide an implementation for textboxes, ignoring `XXXX_SETTTEXT` and `XXXX_GETTTEXT` messages. Instead, the framework may use custom defined messages for implementing the same functionality. When an external application wants to set a particular text to the given textbox, it would probably send a `XXXX_SETTEXT` message, which will be just ignored by the target window. In that case, the interaction basically fails.

[14]This framework taks its name by the developer company, *Quasar Technologies*
[15]http://sciter.com/

2.6.5 Inspection tools for windows: Spy++ and Snoop

In order to facilitate debugging and try to automate UI interactions, some software utilities have been developed. One of the most famous and widely used in the last 20 years is *Spy++*[16]. This tool is currently owned and maintained by Microsoft and belongs to the suite of the Sysinternals' utilities, a toolset for debugging and monitoring various aspects of Windows' OS. The main goal of Spy++ is to provide information about the windowing system of the OS, inspecting all the available windows. The most relevant information that Spy++ is provides, for each window, is the following:

- Identifier (or Handle)
- Position on screen
- Hierarchy information
- Class name
- Owner process and thread

This information can be used to debug applications or to interact with them. In fact, by knowing the window's HANDLE, a program might send or post messages to it. Moreover, it is possible to determine which message has to be sent by looking at the class of the window. If that class is documented (or simply is a well known class), then the developer would probably know which kinds of message does it handle. Nevertheless, it is also possible to identify different parent-child relationships among UI elements within windows, by taking advantage of the EnumChildWindows() Win32 API [57].

The way Spy++ works is described by an article available in the MSDN repository[17]. In short terms, this utility sniffs all the messages sent to each window, by registering a global system hook for the send and post messages operations [22]. Then, whenever a process sends or receives a message, the hook is triggered, and relevant message information is copied into a shared memory structure. Afterwards, Spy++ reads that memory structure and provides information about the windows on the screen, in an organized manner through a simple user interface.

Like Spy++, many other tools have been developed. Many of them behave exactly in the same way and provide some interaction functionality (Spy+++ only displays information). In general, they differ for the support offered for each UI framework. In other terms, different utilities are aware of distinct window class sets. For instance, Spy++ is only capable of extracting detailed information out of Visual C++ elements, which provide known class names and correctly handle most of the standard windows messages. *Snoop*[18] represents another example. This utility provides advanced information when analyzing UI programs developed using the Microsoft *Windows Presentation Foundation* (WPF) framework[19]. Some GUI frameworks provide their own specific inspection tools. For example, the Sciter framework provides a debugging tool. That tool is capable of displaying detailed information about the nested HTML elements (composing the frontend): other generic tools are unable to do the same when dealing with UI generated by Scitor framework.

[16] https://msdn.microsoft.com/en-us/library/aa242713(v=vs.60).aspx
[17] https://blogs.msdn.microsoft.com/vcblog/2007/01/16/spy-internals/
[18] https://snoopwpf.codeplex.com/documentation
[19] https://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx

2.6.6 Inspection tools limitations

Beside inspection tools are capable of getting much relevant information about on-screen windows, that is not enough to enable automated interaction in every context. In fact, when applications rely on unknown custom controls, there is no guarantee about the message handling feature they will implement. Moreover, applications may not expose widget modularity, by only providing monolithic controls, generated at compile time by their GUI frameworks.

An example would be an application showing just one custom window, which handles all types of message. Figure 2.6 shows a possible implementation of such mechanism. The application would only generate a single window object, and will just register that class through the RegisterClass() API call. The associate WndProc() function might treat messages differently according to an internal mapping, defined within the logic of the handler itself. Thus, the window is able to simulate behaviors of any type, from buttons to comboboxes, but only one window element is published to the OS. In this case, inspection tools will just detect one window, even though the WndProc() is able to simulate sub-elements.

The proposed case is not far from the reality. The Sciter framework works in a similar way. When building an UI on Windows, the framework will just register one window, and route all the messages to an internal event-handler. This function (WndProc()) translates those event into DOM events [3], handled internally within the Sciter's HTML engine.

Beside generic GUI inspection tools are useful with standard GUI implementations, they do not provide exhaustive information when dealing with advanced graphical interfaces. Their main limitation resides in the impossibility of inspecting custom monolithic UIs, that are admitted by Windows. Therefore, UI interaction mechanisms cannot only count on such detection techniques.

2.6.7 Image recognition frameworks

GUI analysis may be applied through other techniques. The most generic one is represented by general purpose image-recognition engines. Those engines are based on *digital image processing*, i.e. the process of acquisition and transformation of visual information operated by a computer [90]. In other terms, they capture images (acquisition), apply a specific set of filters (transformation) and compute *image analysis*, i.e. examination of image data to solve a specific imaging problem [90]. Although the main use cases of those frameworks mainly concern military and civil applications [90], they can be applied in our context. Given the scope of our application, the imaging problem consists in identifying UI elements on a specific window, so we might automate interaction with them. Therefore, pattern recognition algorithms serve as great assets for our goal.

At the time of writing, a few opensource computer-vision frameworks are available. Among the most known we find *OpenCV*[20] and *AForge.NET*[21]. They mainly differ for application level libraries, performances and functionality; nevertheless, the two can serve for our goals. In particular, AForge.NET provides both libraries aimed at manipulating images (filtering) and computer vision algorithms. Moreover it supports some other functionality, such as *machine learning* algorithms and *evolutionary algorithms* [25].

A practical example of functionality provided by AForge.NET framework is given by

[20] http://opencv.org/
[21] http://www.aforgenet.com/

Figure 2.7. The screenshot shows an use case of the AForge.NET library, applying shape checking algorithms to an image with black background. The image clearly shows that AFgorge.NET is capable of identifying a series of geometric shapes such as quadrilaterals, triangles and circles.

2.6.8 Optical character recognition techniques

Among computer vision applications and image analysis techniques, one is particularly relevant for our scope: optical character recognition (OCR). This technique is defined as the process of extracting textual information out of a digitalized image. In general, this process is characterized by two stages: text preprocessing and text identification [40].

Image preprocessing consists in applying a series of filters or transformations to a digitalized image, aimed at removing non-significant pixel information (noise) out of the source image. If the source of information is in a material form, digitalization is required: this is usually performed through cameras or scanners. Afterwards, in the preprocessing stage, the source image is manipulated with transformation tools. In particular the image is oriented through rotation or flipping, while color information is removed (usually through binary threshold filters). Then, potential text areas are identified through computer vision algorithms. Finally, text identification is applied, using the technology specific of the OCR library used.

One of the most known opensource OCR engine is Tesseract[22] [40]. This software framework solves the second part of OCR process, i.e. performs text identification starting from a digitalized image. In case a source image needs any preprocessing phase, another framework has to handle that. Thus, Tesseract is usually used alongside one of the major computer vision libraries, such as OpenCV or AForge.NET, which provide tools for preprocessing images before applying text identification techniques.

Text recognition technologies are particularly relevant for our scope, because they can be used to extract information out of the UI during of an installer process. In particular, it is possible to use them in conjunction with other image analysis algorithms in order to identify buttons, checkboxes and other UI elements, even when they belong to monolithic custom controls.

2.6.9 UI automation frameworks

Automating interactions with GUI is an important objective with several use cases. One of the most common is software test automation. In this particular case, automating GUI interaction allows the developers to run a suite of predefined tests and check for their results, speeding up the testing processes and drastically reducing costs [15]. For instance, IBM offers a specific framework for developing test-cases and automating GUI interaction, called Rational Functional Tester (RTF). The framework enables developers to design functional and regression automated tests, supporting a variety of applications, such as .NET, Web, Java [5].

Another relevant application of UI automation frameworks consists in automating software deployment or configuration. This can be done by simply simulating user's input programmatically, based on timers or in response to UI events. A widely adopted tool, serving this matter, is AutoIt[23]. AutoIt mainly consists of a BASIC-like scripting

[22] https://github.com/tesseract-ocr/tesseract/wiki
[23] https://www.autoitscript.com/site/autoit/

language, aimed at automating Windows GUI. This scripting language provides high-level functionality to automate winform applications, enabling users to simulate keystrokes and mouse events on the running system. Although being a community-based freeware platform, providing no warranty neither paid support, AutoIt has been successfully applied to industrial contexts [24].

When dealing only with native supported Windform or Visual C++ widgets, UIAtuomation[24] library provides another automation framework to work with. Developed by Microsoft, this library is capable of providing programmatic access for most of the UI elements natively supported by the operating system. In other terms, UIAutomation consists of a library able to interact with any native windows window and expose its information in convenient way, hiding most of the background low-level messaging system.

It is worth to notice that automation framework only enable GUI interaction, but still require manual configuration. In fact, all the cited frameworks need scripting configuration for each different use case. For instance, we might use AutoIt to script an automatic installation of the Microsoft Office package. Once the automation script is ready, then we might use it to automatically deploy the same package on many different machines. However, using the same script with another installation package would probably fail to install the software. This situation is a consequence of the many distinct and highly configurable GUIs, used for application deployment. Therefore, automation frameworks by themselves are not able to automate every installation process. Instead, they need some scripting inputs, which could be produced by another application component, which is aware of the installation context.

2.6.10 Installers vs Applications, considerations

Some operating systems enforce installation procedures, by providing standard GUIs and OS native support for application management. Android™ [25] is one of them. This mobile operating system implements some standard installation procedures and enforce them thanks to a dedicated module, called *PackageInstaller* [29]. Thus, regardless of the application being installed, the PackageInstaller always shows a standard and uniform installation GUI. It gives information about the application name, developer and required permissions. Moreover all the installer files have to comply to a standard format, called APK, which is the only one processed by the Android Install Manager. Given such a context, it is rather easy to discriminate installer files (APK) from other executable files; moreover automation of GUI interaction would be straightforward.

However, our application focuses on Windows OS. Every application running on windows is potentially an installer. Therefore, there is no systematic way to discriminate installers GUIs from application GUIs, theoretically. However, when observing windows application installers, GUIs tend to be homogeneous. One of the main reasons depends on the usage of installer builder frameworks. They enable semi-automated GUIs implementation, facilitating application deployment procedures. Hence, installers bundled with a common framework tend to be similar, in terms of exposed GUI. Moreover, some installation frameworks are very popular, therefore they have somehow imposed a *de-facto* standard for installing GUIs. An example is the Windows Installer system. When running a MSI file through the MSIExec service, the GUI is built from templates, filled with the information contained into the MSI file. That information describes the UI at an high level, by using

[24] https://msdn.microsoft.com/en-us/library/ms747327(v=vs.110).aspx
[25] https://www.android.com

concepts like *windows*, *buttons*, *labels*, *dialogs*,*checkboxes* and so on. In this way, the GUI is generated at runtime, matching the *look and feel* of the installing system. Another example regards the executables produced by popular installer builders, such as Install Shield and Inno Setup. Both these frameworks produce most of the binary files following a fixed file structure. Therefore, *unpackers* (a.k.a. decompilers) have been developed in order to manually extract information about these kinds of installers. For instance, archive extractors like *Total Commander* or *7zip* may be able to extract files out from self-extracting installer. Still, those methods only enable partial extraction of bundled files, providing no further information about the installers.

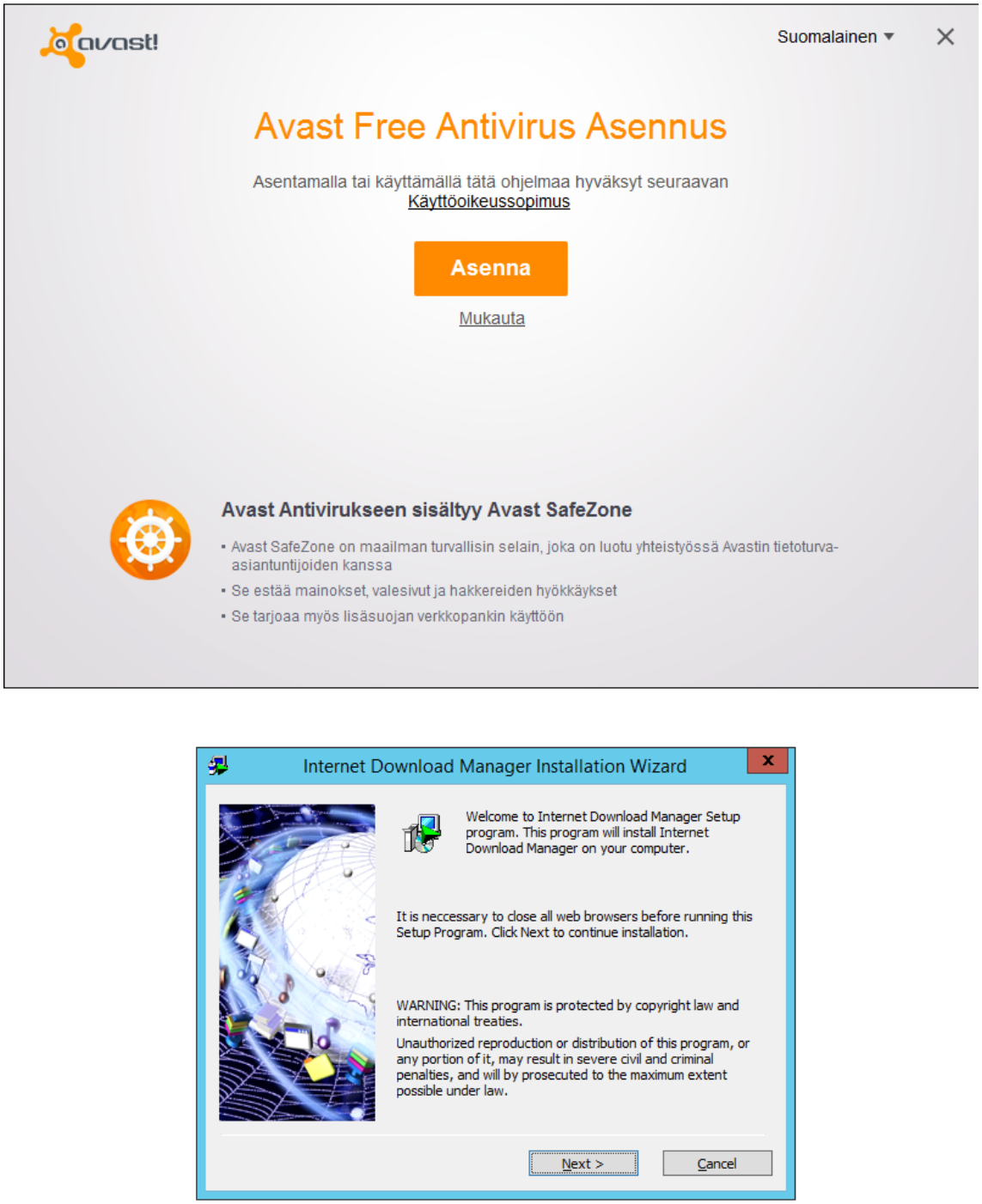


Figure 2.5: Appearance comparison between two installers. On the top, Avast Antivirus Installer, built with Sciter. On the bottom, Internet Downloader mangager, built with InstallShield.

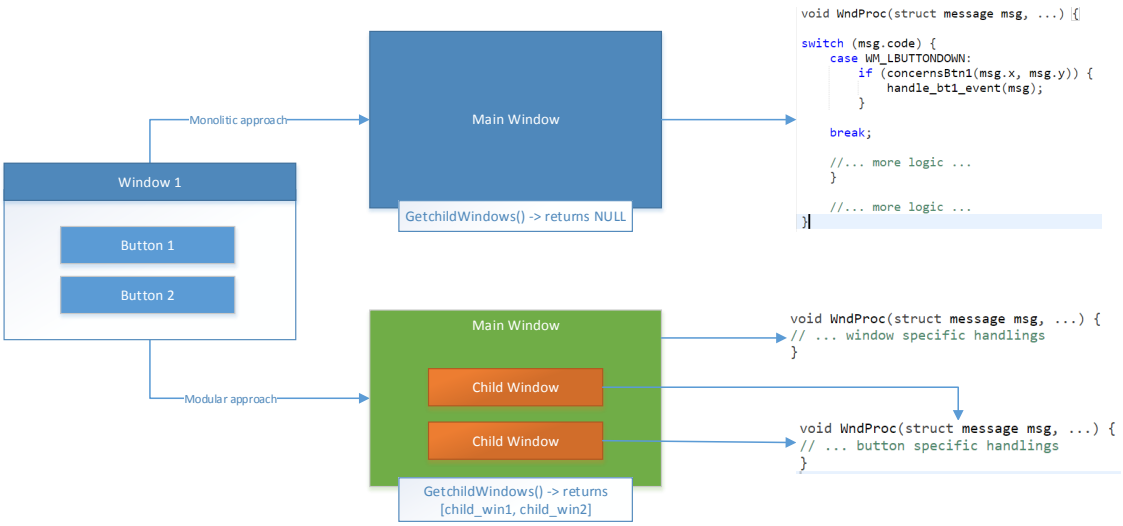


Figure 2.6: Comparison between monolithic GUI approach and modular GUI approach.

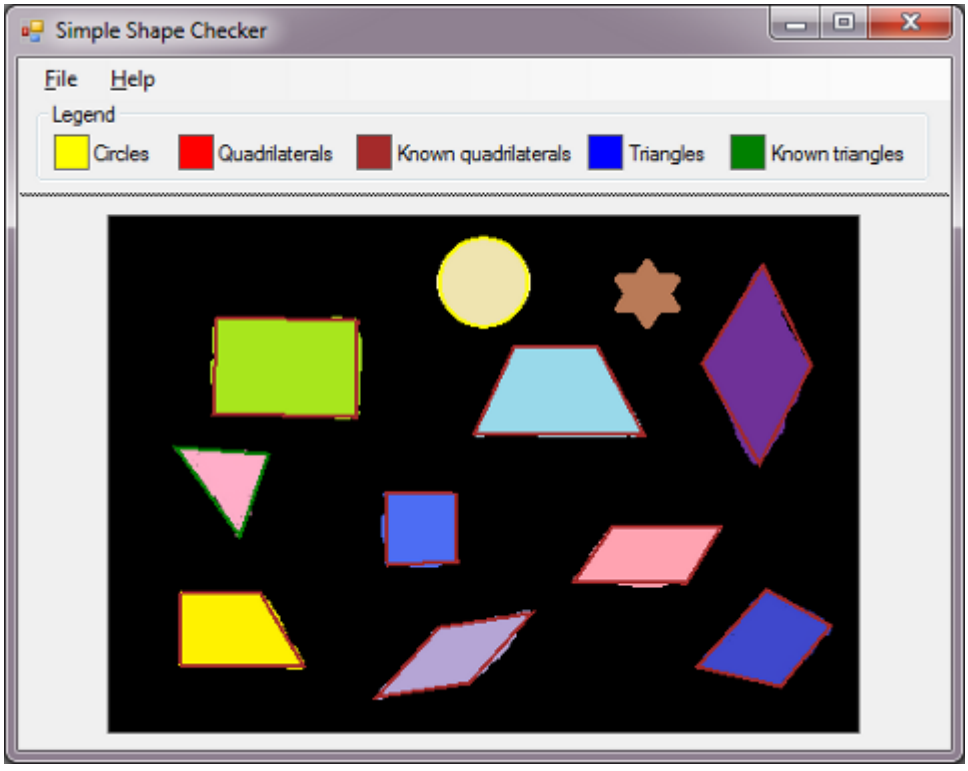


Figure 2.7: Example application provided by AForge.NET applying shapes recognition algorithm.

Chapter 3

Related Work

Beside the PUP threat was little addressed in the past, recent researches show users starting to care about those [83]. Antimalware companies are offering client-side solutions to fight the PUP threat, while academics are focusing the research over the PUP area, as proved by a recent massive PPI distributors study [86].

Despite some tools exist for automated malware analysis, there is no specific system aimed at studying the PUP phenomenon. At the best of our knowledge, the system we propose is the first one targeting automated PUP analysis, taking care of a number of challenges not directly addressed by current available sandbox mechanisms.

3.1 Automated Malware analysis

Potentially unwanted programs are semantically different from malware, nevertheless they still share many common aspects with spyware. Current solutions aimed at analyzing spyware threats include client-side anti-spyware engines and cloud-based sandboxes. While the former runs on the users' systems, the latter runs on the cloud, performing automated and unattended analysis.

3.1.1 Anti-spyware solutions

The majority of antimalware vendors have already recognized the popularity of the PUP threat. Thus, they have started to integrate PUP signatures within their databases, offering a basic layer of protection against this threat. However, two problems affect classic antimalware products. The first one is the difficulty in detecting new PUP versions. The second regards legal implications caused by automatic PUP removal and false positive detections.

Concerning PUP detection task, the majority of current antimalware products implement active PUP recognition by inspecting a subset of potential dangerous behaviors, commonly exposed by PUPs [8, 34, 47]. Those actions are listed in 2.3.2. Moreover, other compoanies adopt more aggressive detection policies, based on heuristics. As an example, Symantec implements a specific heuristic aimed at spotting PUPs, formally named *SONAR.PUA!gen5* [85]. Another more conservative way for detecting PUPs is by relying on user's consent. As an example, Malwarebytes offers a support forum where to publish PUP or false positives, so that balcklisting/whitelisting is possible [46].

Given the ambiguous legitimacy of PUPs, antimalware products are facing legal issues in performing automatic clean-up of such type of software, because users tend to give their

uninformed consent to their installation [19]. As a consequence, antimalware products do not have the explicit right to remove that software, because they cannot verify the legitimacy of PUP installation.

In order to find a trade-off between PUP removal and consequent legal issues, antimalware vendors apply different strategies. Some vendors chose a conservative approach: antimalware might spot the PUP threat but will not take any action against it. It would just warn the user about the threat and leave the final decision to him. Such an approach is often choosen as the default behavior of many antimalwares, such as *Malware bytes' Anti-Malware PRO* [50]. Another possibility is the one applied by *Eset's Nod32*, which asks the user whether activate the PUA protection at install time. By doing that, the user gives its consent to automatically remove potential unwanted applications, as soon as the antivirus engine spots them.

Main limitations of client side antimalware solutions heavily depend on the user's ability of understanding the PUP threat. Indeed, the majority of antimalwares tends to be quite conservative when it comes to automatic PUP removal. Moreover, those products do not keep track of the UI offered during software installation, which instead offers relevant information for classifying PUPs [17, 21].

3.1.2 Sandbox Analysis

The closest tools available today for analyzing fast-spreading PUPs are MSASs. The *Cuckoo Sandbox*[1] represent an effective option for analyzing unknown software, but sill lacks of crucial automation aspects for interacting with installers UI. Also, MSASs generally require virtualization technologies to perform malware analysis, which sometimes may be too restrictive for certain classes of PUPs or malware.

A similar approach has been adopted by Stamminger et al. while analyzing spyware [82]. Differently from other systems, Stamminger et al developed a virtual sandbox that takes into account UI interaction (even though little information is given on the UI interaction mechanism) and collects information to discriminate spyware from malware. On the other hand, that analysis system focuses on browser extensions. Therefore, it only describes how many installers installed a browser helper or plugin, but does not consider possible third party utilities, such as fake anti-malware or registry cleaners.

Thomas et al., in a very recent study [86], investigated the PPI networks applying both sandbox analysis and automated PUP binary collection. After manual inspection of a large number of PUP downloaders, Thomas et al. reverse-engineered some PUP download protocols from different PPI distribution networks. Then, special crawlers (*milkers*) were developed in order to collect PUPs in large quantity. Gathered PUPs were finally ran on a sandbox, without any UI interaction mechanism (which would not be necessary at this stage). However, the emulation of PUP installers is subjected to protocol variations. As a result, reiteration of the study requires new manual inspections of PUP installers, in order to reverse-engineer new versions of the protocol. The same limitation affects the case in which new PPI networks arise. As a result, the proposed analysis mechanism of Thomas et al. does not meet requirements for an automated PUP installers inspection system.

Some malware started to react to public Internet connected MSASs, by blacklisting their ip addresses [95]. So, some bare metal or hybrid sandboxes have been developed. Spensky et al. introduced Lo-Phi, a monitoring mechanism taking advantage of special hardware for auditing [81]: malware hiding in virtual environment did not evade that *metal*

[1] https://cuckoosandbox.org/

sandbox. However, the costs of high-specialized hardware is relevant and that approach does not offer any facilitation for UI automation.

Another kind of approach is offered by DRAKVUF™[2]. This technology relies on Intel's hardware assisted virtualization capabilities (VT-x), in order to implement transparent in-depth analysis of unknown software. DRAKVUF is built over XEN virtual machine manager and is able to analyze binary code execution within its VMs, reducing the number of artifacts exposed to the running software [49]. DRAKVUF basically combines low level inspection techniques with automation-capable virtualization technologies. However, like all the other solutions presented, it still does not implement any GUI automation approach to interact with the unknown software being analyzed.

3.2 Automated UI Interaction

Concerning GUI automation, many research works focused on automated GUI testing and validation. Among all, Grilo et al. [35] presents a reverse-engineering based approach, aimed at automatically extracting GUI models during application execution. This goal is achieved by using Microsoft UI Automation libraries[3]. Unfortunately, UI Automation alone is not able to deal with custom native GUI, thus this approach is not robust enough for analyzing certain classes of software.

Murphy Tools [16] offer a way of automatically extracting GUI models from applications, with a platform independent approach. Its goal is to improve previous GUI extraction tools in a platform independent way, by providing a modular architecture and pluggable UI drivers (os dependent UI scanners). Its goal is to generate automatic GUI tests model, by dynamically crawling the application GUIs, testing all the possible user interactions.

Automatic interaction for software installation is also addressed by a number of commercial products. *Silent Install Builder* claims the ability of automating installers produced by popular frameworks, such as InstallShield, Inno Setup, Nullsoft Installer and Windows Installer [9]. However there still are unhandled installer frameworks, such as Self-Extractors and Sciter, which require manual UI automation scripts. For instance, we were unable to perform unattended installation of *Spotify*[4], a popular music streaming application.

[2] http://drakvuf.com/
[3] https://msdn.microsoft.com/it-it/library/ms726294(vs.85).aspx
[4] https://www.spotify.com

Chapter 4

Problem Statement

Boosted by its valuable business model (PPI scheme), the distribution of PUPs has become a relevant phenomenon, estimated as 60 million download attempts per week [86]. In response, the majority of antimalware companies have developed proprietary techniques aimed at contrasting the PUP threat[8]. Most of those techniques share the same basis: they classify PUPs by looking for a set of particular behaviors, as listed in 2.3.2 [33]. Although PUP classification is still a hard-to-tackle problem, there are few tools aimed at studying the PUP phenomenon. Moreover, most of them rely on classic antimalware products, installed on users' machines. Such an approach has a series of drawbacks:

- The PUP analysis happens on client side, at the time the PUP is being installed. Its analysis is triggered only when it reaches the users' systems.
- In order to identify PUP distributors in the PPI model, client-side solutions require collection of user's browsing history. As a consequence, installation of security toolbars or extensions is necessary in order to track the provenience of PUPs. However, toolbars might negatively impact on user's experience. Moreover, users might disagree to disclose their browsing history with antimalware vendors.
- There is no support for automated PUP analysis: current MSASs systems are unable to click through the installation process in a reliable way.
- In order to avoid legal issues, most of antimalware companies require users' feedback to confirm the PUP classification for the binary [27]. That feedback might arrive several days after the first spread of the PUP, letting the infection grow in the meanwhile.
- Current systems do not take into account the UI presented by the installer program at runtime. Those UIs contain relevant information that might facilitate a better classification of unwanted applications, via common EULAs [21] or graphic patterns.
- A program may explicitly expose none of the dangerous behaviors, still it might be undesired for the users. In such cases, classification based exclusively on dangerous behaviors is ineffective.

In order to tackle those issues, we developed an automated system, capable of automating software installation and analysis. By taking advantage of UI-recognition and OCR technologies, alongside experience-driven heuristics and malware dynamic analysis

techniques, we adapt classic malware analysis approaches to a new class of software, not necessarily malicious, defined *grayware*. As a consequence, the result of our work provides a new powerful tool in the fight against the PUP threat. In doing so, we addressed a number of challenges, described in the following sections.

4.1 Automating software installation

At the best of our knowledge, there is no automated system aimed at analyzing PUP installers. Current automated solutions mainly target malware, which generally hide itself to the user. Thus, UI interaction is not directly addressed by those solutions. PUP installers are different: they rely on user habituation and *consent-extortion*, by displaying complex and incomprehensible EULAs [17], alongside linear and simple interfaces during the installation processes. Thus, our goal is to provide a way of emulating a *habituated user*, who puts low attention at the installing process and potentially falls in the trap of PUP installers.

4.2 Data collection and correlation

While classic malware directly alter data in the user's system or silently steal information, PUPs may monetize by affecting user's system in many different ways. One of the most common ways is to pretend to offer better search results while surfing the web, altering the classic web surfing experience. Invasive advertisements and surfing data collection constitute common ways for earning revenues, in this context. PUPs may also led to the installation of other software, for which no warranty is given.

As a consequence, it is crucial to monitor all the most relevant assets of the system, such as network, file and registry accesses, similarly to classic antimalware software. At the same time, the analysis system must take care of collecting information about the structure UI structure proposed during the installation phase. Moreover, the information needs to be collected in aggregated form, providing a convenient way to query and mine gathered data.

Differently from classic antimalware approaches, running on users' systems, our analysis system must collect information about the source of the analyzed sample. In this way it is possible to extract valuable correlations. Once an installer has somehow been found responsible of installing PUPs, the system must be able to answer questions like the followings:

- Which other installers installed the same PUPs?
- Where did this installer come from?
- Is this installer distributed by other distributors?

4.3 Scalability and performances

Potentially unwanted software evolves rapidly. Nowadays PPI distributors are evaluated to trigger as many as 60 million of download attempts per week [86]. With such a market, the number of PUPs grows rapidly. Within this context, the performance of the analysis process plays an important role. Therefore, the analysis has to be engineered in order

to perform thousands of analysis per day, and to scale up performance conveniently. In particular, we identify two important performance counters: the throughout of the analysis system and the time taken by every single analysis. The former represents the number of analysis that the system is able to perform in one minute. If dealing with millions of download attempts per week, the system must be able to scale up to thousands of analysis per day. The latter expresses how long does a single analysis take. In this case, our objective is to limit the single analysis time to a few tens of minutes.

4.4 Avoiding MSASs detection

Automated systems for analyzing malware exist. They take the name of Malware Sandbox Analysis Systems (MSASs). Beside they offer a convenient way of collecting behavioral information about the analyzed binary, they still suffer from easy detection [95]. Most of those anti-MSASs techniques are based on virtual environment detection, taking advantage of artifacts affecting the virtual sandbox. Therefore, some malware may behave differently when a sandbox environment is detected, biasing the analysis.

The analysis system presented in this work should then implement convenient counter-measures to MSASs detection, as secondary objective.

Chapter 5

Design of a Windows PUP analysis infrastructure

In this chapter we present the general design choices adopted to develop our automated MSAS, aimed at analyzing PUPs on Windows OS. The first part of the chapter is dedicated to our objectives. It follows a general presentation of the infrastructure, from high architectural level. After that, we focus on each sub component of the infrastructure, presenting its roles and objectives, describing its main characteristics.

5.1 Design goals

The starting point of this work is the concept of MSAS. Malware Sandboxing Analyzing Systems provide a safe and scriptable environment where it is possible to analyze malicious software. Indeed, MSASs already tackle most of the limitations of other malware analyzing approaches, such as:

Signature based detection. Malware evolves very fast, thus simple signature based detection is ineffective, especially against 0-day attacks

Simple static analysis. Malware is generally packed: static analysis technique is ineffective for analyzing malware.

Reverse Engineering and Debugging. Both reverse engineering and debugging require human interaction and are not applicable for large scale analysis.

Beside being particularly effective against malware, those sandboxes are characterized by a number of limitations, making them inappropriate for PUP analysis. As previously anticipated in Section 2.5.1, we identify the following problems when dealing with MASAs and PUPs:

- Missing automated GUI interaction
- Poor information correlation
- MSASs artifacts can bias analysis

Our primary objective is to evolve the classic MSAS, by addressing those limitations. In other words, the main goal of this work is to provide a **fully automated analyzing**

infrastructure, able to download, install and analyze software installers, on Windows OS. To do so, the MSAS has to take into account GUI interaction problems, should provide conveninent and menainful data correlations and should offer some defense against MSAS detection mechanisms. Moreover, in order to facilitate further analysis, our idea is to store collected data into a centralized database, as well as providing human-readable reports for single PUP analysis.

Secondary objectives address the efficiency, scalability and flexibility of our design. Moreover, performance represents a key factor in our system. Indeed, in order to be effective against the fast spreading of PUPs, PUP analysis should take a relatively short amount of time. At the same time, data collection has to be detailed enough to spot relevant correlations. Thus, some trade-offs have to be achieved: too much verbose data collection causes longer analysis periods. For this reason, it is necessary to develop a scalable system, relying on a **distributed architecture**, taking advantage of **multiple hypervisors** and **cloud technologies**. To achieve such objectives, we designed the whole system with software modularity in mind, applying - when possible - multiplatform technologies.

Finally, in order to avoid MSAS detection, our design has to take into account side effects caused by virtualization technologies. Thus, the system should be able to perform malware analysis on non-virtualized systems, usually named *bare metal* systems.

5.2 Architecture

As any MSAS, our system has to take into account different aspects of unknown applications analysis. Thus, as main design principle, we applied *separation of concerns*, following the *divide et impera* idea [89]. This means that every part of the architecture aims at serving for a specific set of tasks, solving particular issues.

From an high level perspective, the system architecture has to comply with the process defined in Figure 5.1. The first task of our system is to gather application binaries and relevant associated metadata. The consequent action is to analyze the collected binaries, producing verbose analysis data. Then, that data has to be aggregated in a structured manner. The last step in the process corresponds to data analysis and result evaluation.

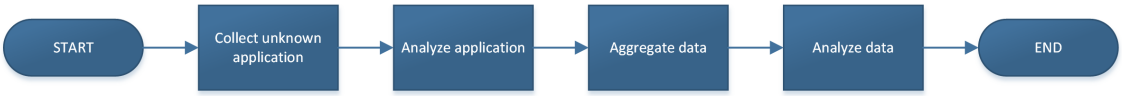


Figure 5.1: PUP Analysis process

In order to implement such a process, the infrastructure must take care of:

- Collecting binary sources to be analyzed
- Administrating the sandboxes
- Performing execution analysis
- Storing results
- Analyzing data and produce reports

To serve those demands and hit our goals, we designed the system with a multi-tier distributed architecture. Figure 5.2 shows an overview of the architecture scheme.

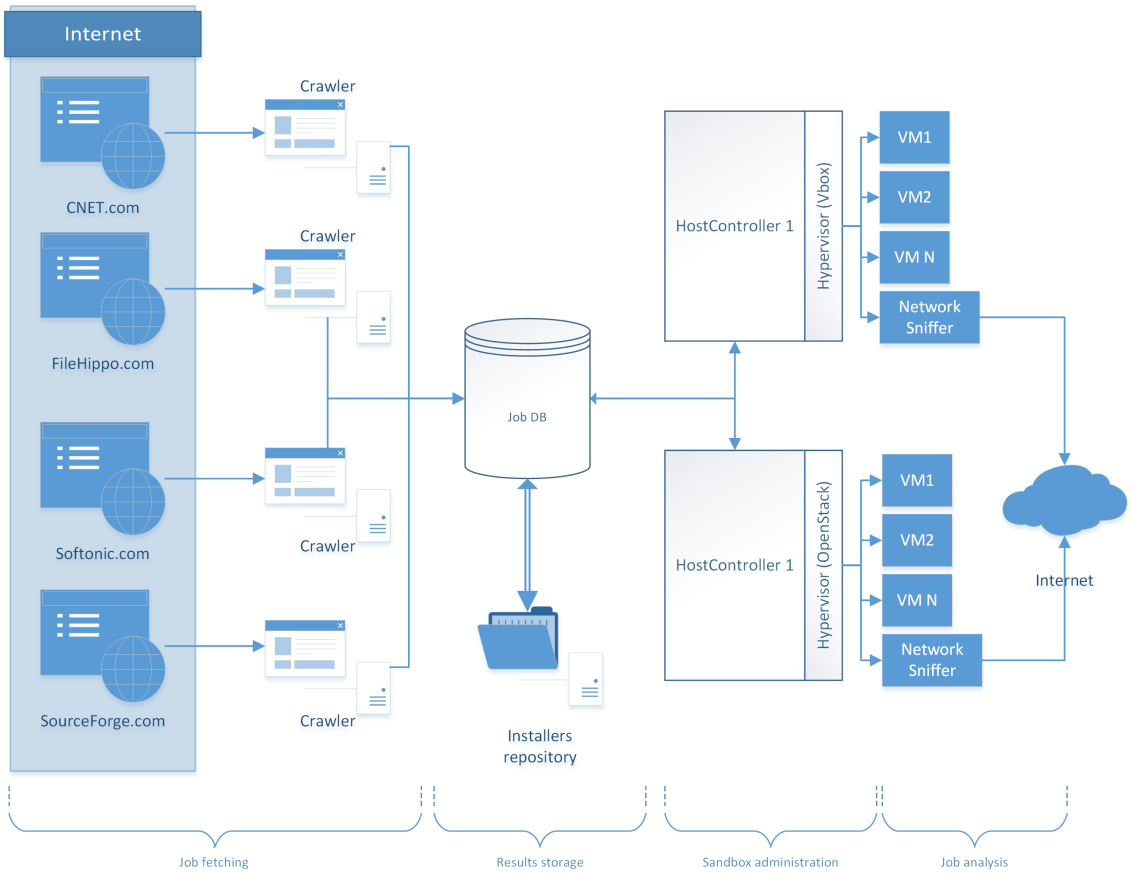


Figure 5.2: Architecture Overview

5.2.1 Overview

Four different modules layers take into account distinct aspects of the system. More specifically, the entire architecture is composed by the following entities:

- Crawler(s)
- Central database
- Host Controller(s)
- Guest Machine(s)

A relational database (*er-db*, or simply *db*) represents the central point of the architecture. The database holds all the metadata regarding the binaries to be analized, which are instead stored on a shared network location. The combination of a binary file with its associated metadata is a *job*. *Crawlers* are software modules in charge of scanning the network and collecting jobs, by storing the binary data and the relative metadata, respectively on the network share and the db. At the same time, the database also manages results of previous executed jobs. The analysis of each job occurs in isolated machines, called *guest machines*, with dynamic analysis and asset-monitoring techniques. Guest controller are not directly attached to external network, instead their traffic is routed through special network entities, named *sniffers*. Sniffers both provide basic networking services to the guest controllers

and at the same time provide networking sniffing capabilities. In between guest machines and central db, we find one or more *host controller*. Those software entities are in charge of managing the life-cycle of guest machines; moreover they fetch jobs from the central db and serve them to guest machines. Once the analysis of a particular job is done, host controllers also elaborate the results and store them in the central db.

The proposed scheme clearly points out both scalability and flexibility capabilities of the infrastructure. In first instance, a certain number of crawlers can simultaneously collect information from different software distributor's websites. Those crawlers may run on the same host where the DB is installed, or may be external, relying on network communication to access the DB. At the same time, multiple host controllers can collaborate in job analysis. Several machines are managed by each host controller and run in parallel: each machine handles a distinct job. Moreover, a single network sniffer serves many guest machines, and collaborates with the host controller associated to those machines. Once the analysis is concluded, gathered data is returned to the associated host controller, which integrates network information (gathered from the sniffer) and stores everything into the central db.

As a drawback, the central database might possibly cause the bottleneck of the infrastrucure; moreover it represents a single point of failure of the entire architecture. However, in order to avoid db-bottlenecks, the system minimizes the number of queries processed against the DB. Moreover, host controllers and crawlers use timers in order to perform retries attempts in case of failures. On the other hand, no specific software countermeasure has been adopted for solving the single point of failure problem. In that case, hardware redundancy and High-Availability (HA) technologies can serve for this matter, at hardware level. Detailed implementation choices, regarding the database querying system, are exposed in the next chapter.

5.2.2 Crawlers

The *job fetching* area of scheme shown in Figure 5.2 is characterized by crawlers. Generally speaking, a crawler is a software module specialized in automated web traversal, extracting useful information out of visited pages [37].

The main objective of crawlers in out infrastructure is to collect inputs for the system. In particular, the following list enumerates inputs for our system:

Application binaries: Binary data of the program to be analyzed

Application file name: File name of the downloaded program

Application hash: MD5 hash of the downloaded data

Download URLs: Full web host and path of the downloaded program.

Download date: Date and time in which the program has been downloaded

Popularity: An optional index, associated with the application, representing its popularity

Each successful downloaded binary represents an input job for our system. Once a crawler finds a new application, downloads it and calculates its MD5 hash, taking note of source URL and download date. Then, it stores binary-related metadata into the db, as a job. However, the database may refuse the job, by checking MD5 hash and download URL. If there already is a job with the same URL source and same hash, the database refuses the new entry. On the other hand, the database will accept a job with same hash but

	Same hashes	Different hashes
Same URLs	No change (Refused)	Binary updated (Accepted)
Different URLs	Application spreading (Accepted)	New application (Accepted)

Table 5.1: Database jobs acceptance policies, applied when in case two jobs share hashes or source urls.

Domain	Global Rank
cnet.com	171
softonic.com	247
sourceforge.net	316
filehippo.com	696

Table 5.2: Alexa.com rankings of most known freeware download websites

different source URL, or with same URL but distinct hash. In this way, the db will contain information regarding both the "evolution" and the "spread" of a certain application, as described in Table 5.1.

The current implementation of the described infrastructure provides four similar crawler implementations. In fact, given the PPI business model behind PUPs, most of the applications are distributed through centralized websites (affiliates). For this reason, we focused our attention on the most important freeware software distributors at the time of writing. Thus, the most four popular freeware distributor websites have been selected, according to the Alexa's global rankings[1], as shown by Table 5.2.

For each website listed in Table 5.2, a specialized crawler has been developed. Moreover, each website in the list offers an internal ranking of most popular freeware downloads, therefore a popularity index has been derived directly by the download websites.

5.2.3 Central DB

The central node of the entire system is the transactional database, which implements a number of important roles. More specifically, the database is in charge of storing both inputs and outputs of the system. Furthermore, it resembles the synchronization point of the entire distributed architecture: every crawler and host controller will interact with the same db instance. Thus, from a technical point of view, the database is used as synchronization mechanism to coordinate all the system components.

The logic scheme in Figure 5.2 demonstrates how the database both servers as *input buffer* and *output storage*. Moreover, the same db is used as transactional storage for ongoing analysis.

The db schema used in our system is based on several entities, which are grouped among three categories: **inputs**, **outputs**, **transactional state**. More specifically, Figure 5.3 exposes the entity-relational (ER) relationships of db entities taken into account.

The system fulcrum is identified by the `experiment` entity. An experiment represents the conjunction of inputs and outputs. In other words, an experiment links specific inputs to relative outputs. For each configuration of inputs, there will be an associated experiment record, to which outputs will be linked.

[1] http://www.alexa.com

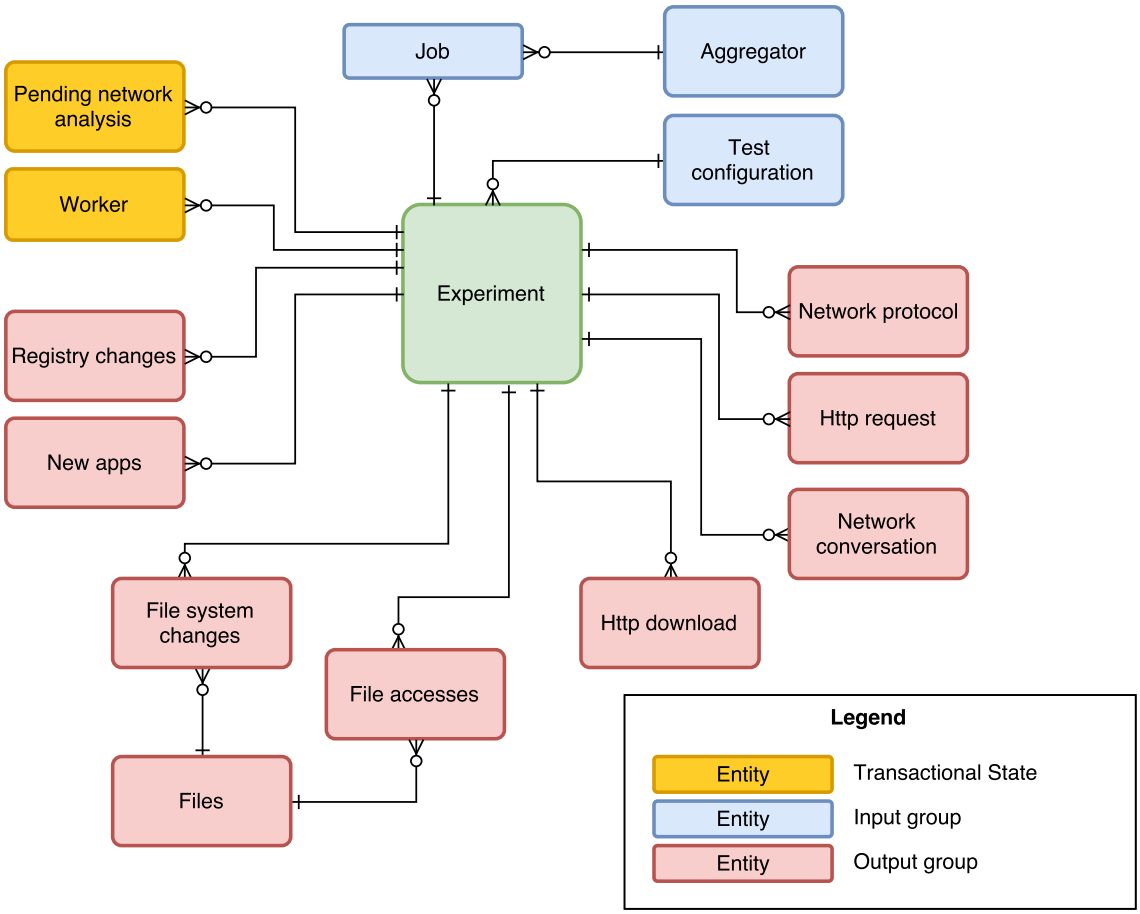


Figure 5.3: DB ER diagram

Inputs of our system consist of `jobs` and associated `test-configurations`. A job identifies an application to be analyzed. Thus, it is mainly characterized by a cryptographic hash, download date and source URL. Binary data, associated to a job, is not stored directly on the database: instead the db contains path references to network locations, where binary data is available. Moreover, each job is linked to the download website where it comes from, named `aggregator`. The relationship between jobs and aggregators is useful to impute potentially source of PUPs to one or more affiliates or PPI distributors. Another input of the system is the configuration of the test environment. As we will describe later on, our system offers some parameters that can be tuned. Thus, it is important to keep track of the system configuration used for each analysis. In this way, it is possible to identify different behaviors of an application under various configurations, in order to spot similarities or anomalies.

The number of entities concerning the outputs is considerably higher. Indeed, we can group output entities in four main sub-classes:

- Network
- File system
- Registry

- Control panels new apps

Regarding network data, the system intercepts used protocols, http requests, tcp/udp flows (conversations) and http downloads. Moreover, statistics about network flows are collected into the network conversation table, which also takes track of contacted hosts addesses, alongside the number of exchanged bytes. Finally, http requests and downloads metadata are collected, such as source host address, path and so on.

Data regarding file system access is based on three entities: `file accesses`, `files` and `file system changes`. The first entity, `file access`, models every attempt performed by the application when trying to open a file (with write access), based on its path (on the FS) and a sequence number. Each file access is referred to a file, which is basically a sequence of bytes. Thus, the `files` entity contains hashes (cryptographic and fuzzy) of every file encountered during the analysis. In order to quickly aggregate file system changes performed by a certain application, a new entity is derived: `file system changes`. This table contains the path of the affected file and some binary flags describing the outcome of application interaction(e.g. new, modified, deleted).

The entity `Registry changes` registers all the interactions between the analyzed application and the Windows registry, similarly to `file system changes`. This table presents, in an aggregated way, the results of application execution over the Windows registry. In particular, for each affected registry key or value, three flags express the outcome of the interaction (new, modified, deleted). Moreover, in case of edited or deleted key/values, the entity takes track of new nad old key/values, thus offering a clear vision of what has been modified and how.

The last output information provided by the system is `new apps`. This entity maps the new installed apps detected by the operating system upon application execution. In other words, *new apps* contains the list of new records spotted into the control panel *Installed programs*. A new record into that list is not necessary and sufficient condition to determine successful application installation. However, it still provides a quick hint to spot installed programs following the Microsoft best practices.

The third group of entities contains implementation-related temporary objects, necessary to the system during its running state. Indeed, *transactional state* group includes `worker` and `pending network analysis` entities. A `worker` represents a running machine performing one job analysis. Thus, this entity contains data useful to administrate that machine, such as start time, id of the job, number of execution attempts and so on. On the other hand, `pending network analysis` maps part of the data aggregation process, and it is used to store jobs that have completed but still are lacking network data processing.

5.2.4 Host Controller

The *host controller* (*hc*) represents a fundamental part of the architecture. In general terms, we define an host controller as a stand-alone software component, aimed at orchestrating the life-cycle of analysis sandboxes. Therefore, an host controller is in charge of starting the machines, serving them the job to analyze, retrieving the results from them and shutting them down (or restarting them).

From software point of view, host controller implementation aligns to Figure 5.4. The main software components of this architecture are the database interface, the main logic module and the machine manager interface. The logic module is in charge of administrating the whole host controller, therefore it closely interacts with the other software components. The database interface handles data persistence, in accordance with the particular database

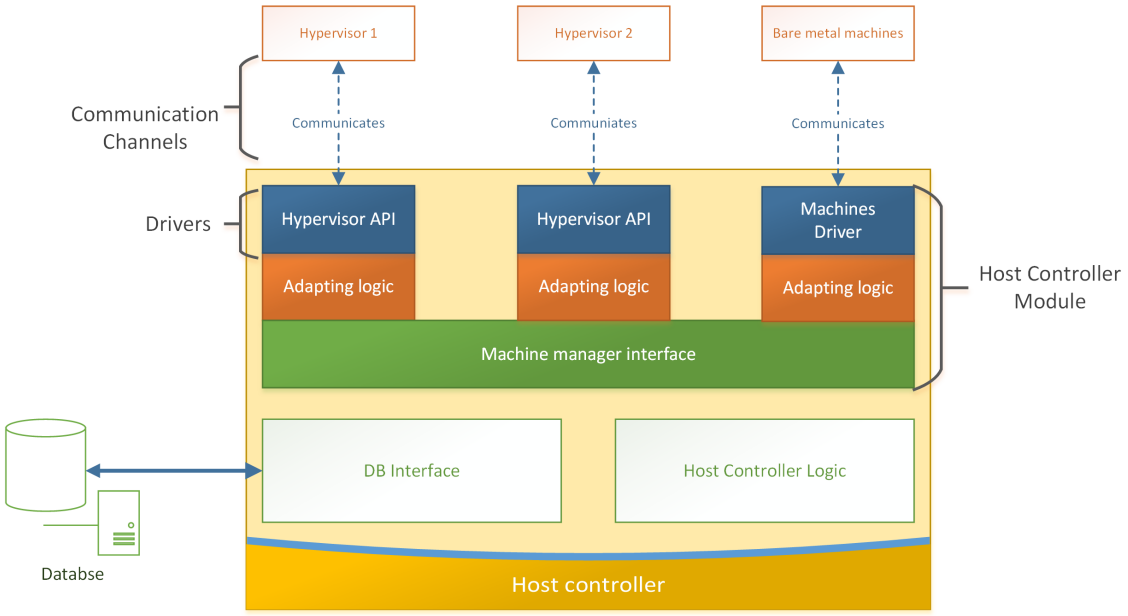


Figure 5.4: Host controller modular software architecture

technology used. In order to independently handle both virtual machines and bare metal machines, an uniform abstraction layer is necessary. Thus, a common interface has been defined: *machine manager*. As a consequence, interoperability of host controllers is easy to extend to many different hypervisors, as well as bare metal machines, by just implementing hypervisor-dependant middleware.

Among the main tasks assigned to the host controller, machine life-cycle management is the most important.Figure 5.5 describes life-cycle of guest machines through a state diagram.

The host controller has to allocate and to setup a parametrized number of machines. After that, the hc will start them. Once booted, each machine requests a job to the host controller, which queries the central db in order to atomically pop one job, marking it as assigned. In this way, multiple host controllers may require jobs at the same time, but each hc gets a distinct job.

Given its nature, the analysis state may persist for a relative long period of time, compared to the other states. Moreover, the machine may fail the analysis and might enter an unrecoverable internal state. For instance, this situation might happen when a buggy application is submitted for analysis. That program may cause a hard failure of the operating system, causing a *Blue Screen of Death* (BSOD)[2]. To recover these possible failures, host controllers are in charge of monitoring the execution of the analysis. This is done by starting a timer at the moment in which the job is served to the guest machine. When the timer expires, if the job has not completed yet, the host controller reports failure state into the central DB, hence takes care of hard resetting the relative guest controller.

Whenever a machine completes the analysis of a given job, a XML report is returned to the host controller. The hc will then persist the report on its disk and will update the database accordingly. After that, the host controller will revert the state of machine which has completed the job. In this way, each machine always starts the analysis from a known

[2]https://msdn.microsoft.com/en-us/library/windows/hardware/ff547224(v=vs.85).aspx

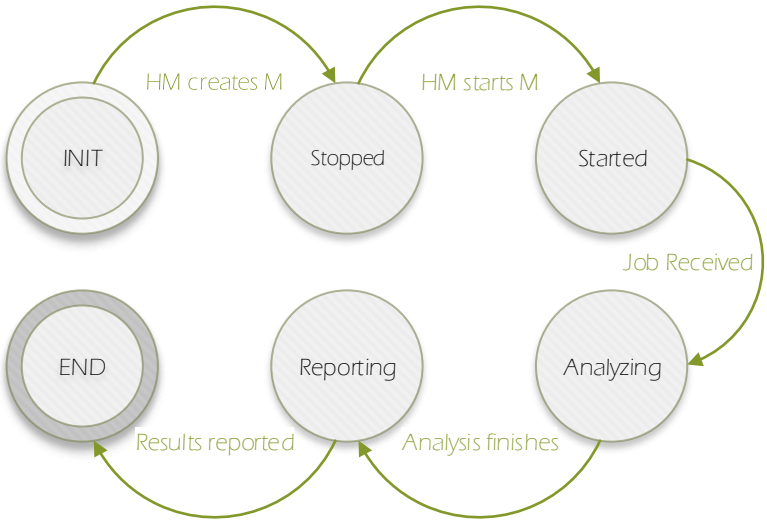


Figure 5.5: Machine life-cycle

clean state, named *started*, according to the Figure 5.5.

Figure 5.6 represents a flow diagram describing how each host controller works. More specifically, the bottom part of the chart marks the parallelism between the result listening task and the jobs serving task. It is crucial to to ensure a good grade of parallelism, in accordance to the number of machines handled by each host controller. In fact, implementing concurrent operations, the host controller increases the throughput of the system.

5.2.5 Sandbox Machine

Each job is analyzed within a *guest machine*. Guest machines represent controlled environments where software analysis happens. Those sandboxes are administrated by host controllers as reported by Figure 5.5. The main objective of a guest controller is to run the job offered by the hc, analyze it through dynamic analysis techniques and then report back all the information collected.

In section 2.5 we introduced the basic properties characterizing any sandbox: observability, containability and efficiency. Our design takes into account those requirements by combining both virtualization support and bare metal support. However our particular context (i.e. PUP analysis) requires additional shrewdness. In accordance with our goals, guest machines require automated GUI interaction capabilities, as well as MSAS detection-avoidance mechanisms. To do so, we designed guest controller with a particular architecture, shown in Figure 5.7.

Machine-Scoped architecture

A guest machine is equipped with an operating system, configured with basic system drivers, in accordance with the underlying hardware (which can also be virtualized). Since our objective is to analyze executables on Windows environment, the chosen operating system is Windows 7 Professional 32bit.

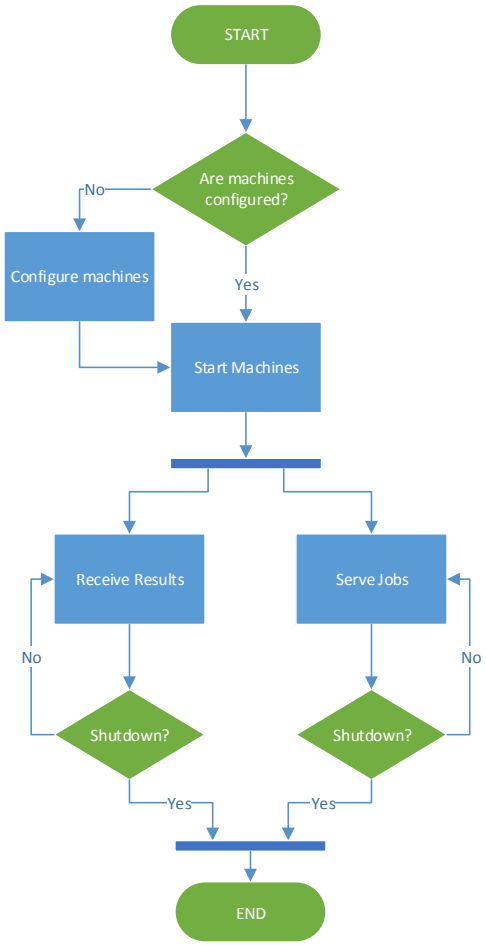


Figure 5.6: Host Controller data flow diagram

A particular process runs within the operating system of the guest machine: the *guest controller*. This software program constitutes the logic core of the guest machine, being in charge of:

- Handling communication with the host controller
- Performing UI interaction
- Reporting results to the host controller

The guest controller delegates another software module to perform job execution: the *injector*. This program is a Windows 32 bit application, written in C++, accessing low level API of the OS. Its main objective is to start the given binary by altering its memory space, in such a way that a custom library gets loaded beforehand. This operation is achieved by using *DLL injection* techniques, explained in section 6.2.3. The loaded library represents the third software entity of guest architecture. Named *HookingDLL*, this module is in charge of intercepting low level API calls performed by the target process and to report them to the guest controller. This is obtained by applying *API hooking* techniques,

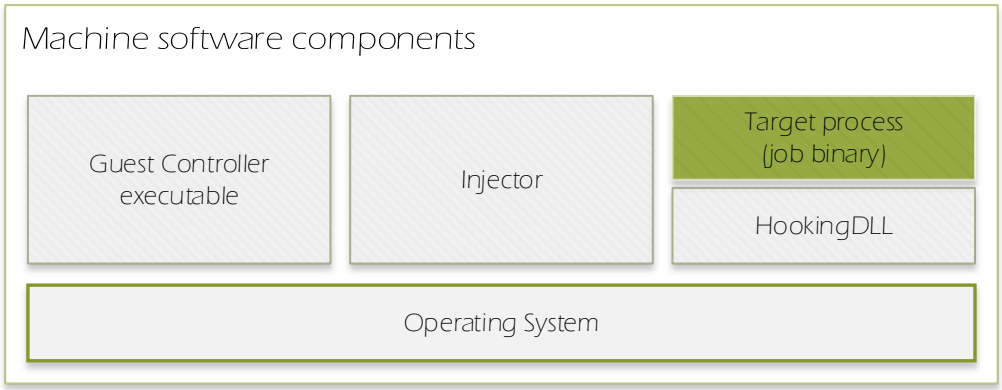


Figure 5.7: Software components of sandbox machine

discussed later on in section 6.2.3. The HookingDLL module is the software component that enables *observability* in our sandbox. By intercepting a certain set of APIs, the sandbox takes track of behavior of the processes, in accordance with the chosen API set. On the other hand, the number and the type of hooked APIs impacts on *efficiency* of the sandbox.

Communication between HookingDLL and guest controller occurs locally to the guest machine. Whenever the target process invokes an hooked API, the routine will first communicate some information to the guest controller, then the actual operation is performed. Therefore, the guest controller has a chance to log what the target process is going to do, before the action happens. This approach is particularly relevant for registry and file system accesses: before any interaction happens, guest controller gathers information about the file being opened (storing its hash, size and so on). Then, when the analysis is terminated, guest controller compares the state of the file before and after the interactions, so that file system changes are detected.

Guest Controller

The guest controller has to accomplish different goals by running distinct tasks. Some of them are time-consuming and live for the entire lifetime of the machine. Moreover, all of them have to run simultaneously. For instance, the GUI monitoring task runs in parallel with the HookingDll logging task. For this reason, the software architecture of geust controllers was defined with software modularity and parallelism in mind. Figure 5.8 describes the inner software modules of the guest controller and marks the interactions with the other two software elements of the guest machine.

The GUI interaction logic is handled by an ad-hoc module, called *GUI bot*, in charge of scanning the user interface, exposed by the target process. This module scans the UI (if any) offered by the target process, and according to a set of policies, interacts with it. The GUI bot applies some scanning algorithms and decision heuristics in order to determine how to interact with the target UI. However, the decision policy may also rely on IPC messages received by the HookingDLL in order to improve the its effectiveness.

The *IPC handler* is a software module responsible of handling interprocess communication with the target process. Messages are exchanged via Windows named pipes, preallocated by the guest controller and filled by the logging functions, which have been injected into the target process. The throughput of messages flowing from target processes to the guest controller is in general high, so multiple IPC handlers are defined in order to

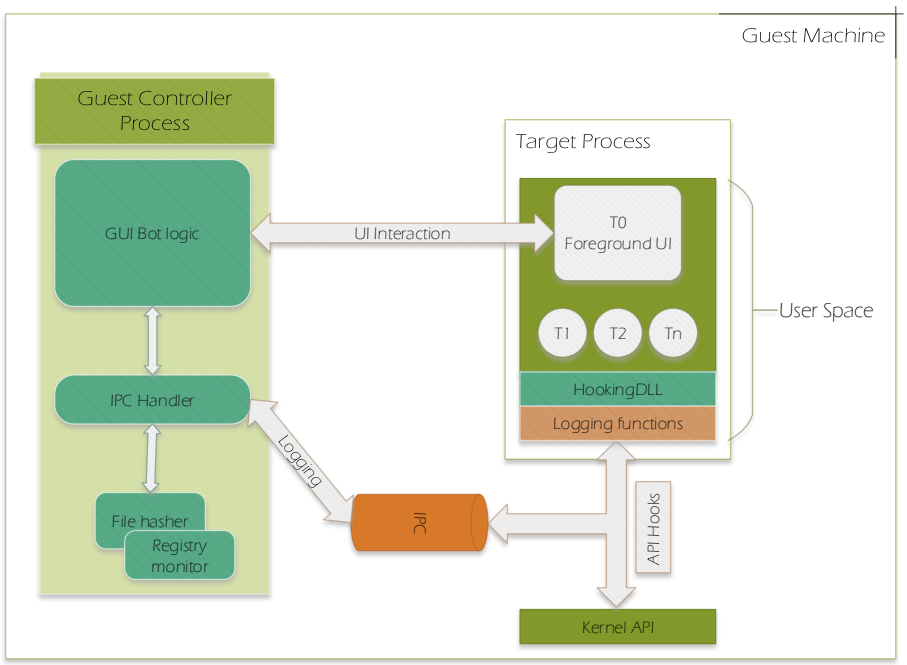


Figure 5.8: Insight architecture of Guest Controller

take advantage of multiprocessing capabilities of underlying hardware.

The third software part of the guest controller includes file and registry monitoring libraries. Those components are mainly used by the IPC handler during massage processing. Among the functionality exposed by those modules there are file hashing and registry querying libraries.

A full overview of the general behavior followed by the guest controller is given by Figure 5.9. When the machine starts, the operating system boots up. The OS image is pre-configured to start a *batch script* after the *auto-logon*. This script waits until network access is automatically configured, afterwards it downloads guest controller binaries from a remote server. In this way, software changes to guest controller do not need manual deployment, since its binary is automatically updated at each boot. Afterwards, the script executes the guest controller application, which immediately initiates network communication with the host controller. If no jobs are ready to be processed, the guest controller holds up for a timeout and contacts back the host controller after a timeout. When a job is available, that is returned by the host controller to the guest controller. Therefore, the gc delegates the injector to start the given job and inject the HookingDLL. At the same time, the guest controller starts listening for log messages coming from the HookingDll and handles UI interaction with the GUI belonging to the target process. Only when no more UI interaction are possible, or when the target process ends, then the guest controller prepares the resulting report and sends it back to the host controller.

5.2.6 Networking Design

According to the system architecture proposed in Figure 5.2, Internet access for guest machines is mitigated by a particular entity: the *network sniffer*. This device has three important roles: to provide basic network services, to sniff traffic and to enable network

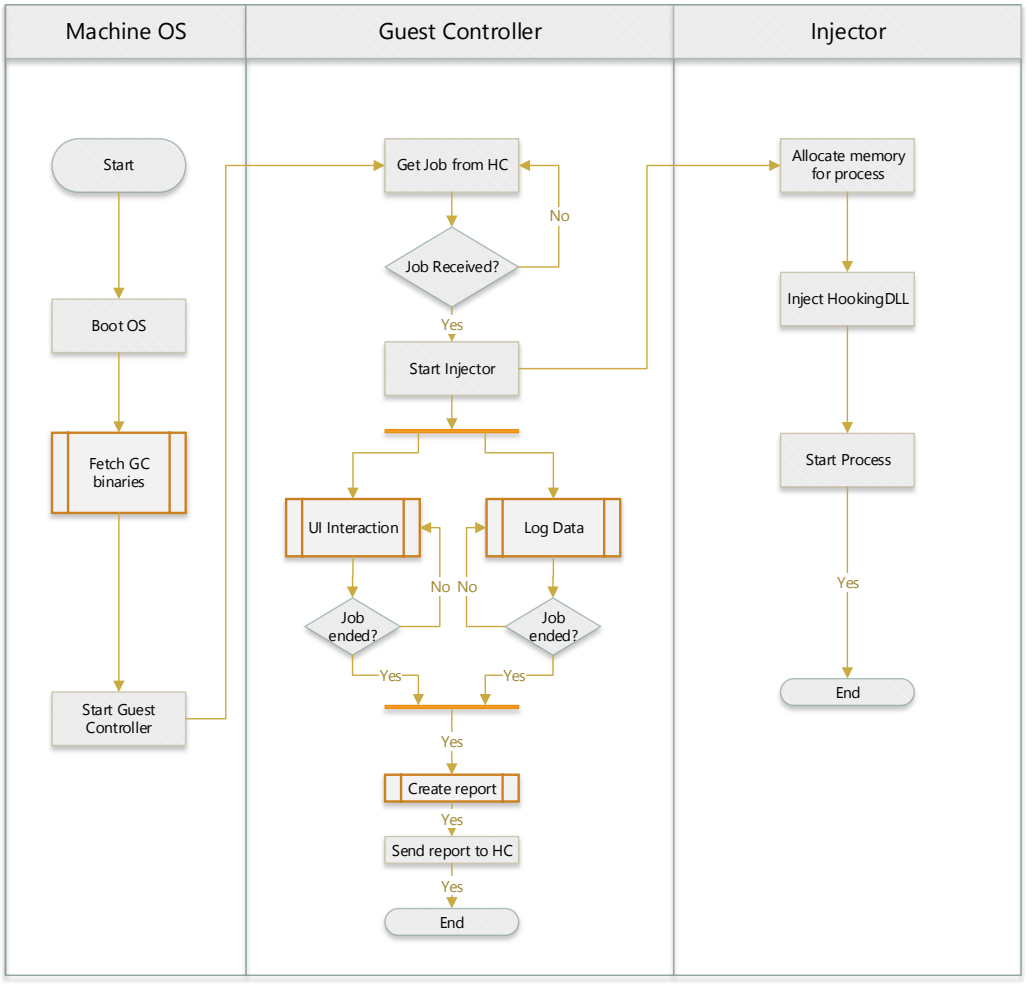


Figure 5.9: Workflow diagram of the Guest Controller

isolation. Figure 5.10 shows the relationships among guest machines, host controller and external nextwork.

Isolation of guest machines is ensured by providing a separation layer in between guest machines and the rest of the network. Guest machines belonging to the same host controller reside in the same local area network (LAN). Therefore, they are not attached directly to the Internet; instead, their traffic is routed through the sniffer.

Basic network services

Guest machines are capable of communicating with external hosts though a gateway: the *sniffer gateway*. This entity provides a number of necessary network services for the internal LAN, such as:

- IP level routing
- DHCP

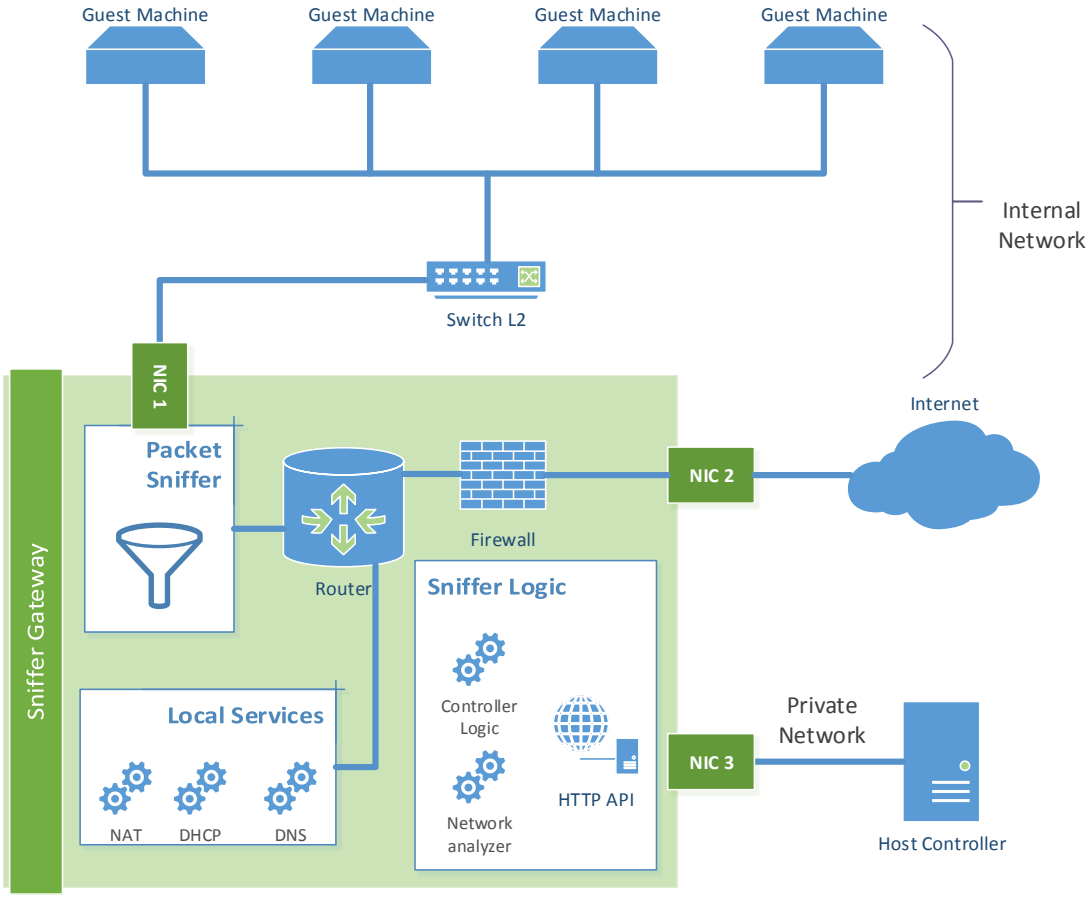


Figure 5.10: Architecture of Network Sniffer

- DNS
- NAT
- Firewall (Optional)

The most important and basic service provided by this gateway is IP routing (or IP forwarding). In fact, the gateway is attached to three different networks, so it behaves mostly like a L3 router. It is worth noticing that the routing service is crucial also for the guest controller process: communication with the hc happens through TCP/IP sockets, thus it requires packet routing from internal to private network, as shown in Figure 5.10.

The second most important service offered by the gateway is the *Dynamic Host Configuration Protocol* (DHCP) service. Indeed, guest machines are configured to automatically acquire an IP address at boot time. This approach is particularly convenient because it reduces the configuration overhead regarding every single guest machine. Without DHCP service, each guest machine would require a preassigned unique IP address for its LAN domain, resulting in the need of some manual addressing task.

At an higher level in the OSI stack, the sniffer gateway also provides *Domain Name System* (DNS) resolution and *Network Address Translation* (NAT). The former is used in order to translate domain names into IP addresses. However, the gateway does not

implement a proper DNS server system, instead it provides a DNS proxy: every DNS request coming from the internal LAN is delegated to the ISP's DNS server. On the other hand, network address translation is necessary in order to multiplex the same external and public IPv4 address among the different guest machines. Therefore, this service would not be strictly mandatory if many IP address were available. That is generally the case of IPv6, when many public addresses are usually available even for a domestic contract.

Guest machine isolation and containability is improved thanks to a firewall. For instance, the firewall prevents guest machines from contacting any other host in the private network other than the host controller. In other terms, firewall rules should be defined in such as way that *any host*, belonging to the internal network LAN domain, can initiate a TCP connection only to a HOST-TCP port combination (i.e. host controller), forbidding all the others. In the meanwhile, firewall rules allow outbounds network connection to the Internet, and forbid any incoming TCP/UDP connection from the Internet. However, the firewall may impact on the analysis, biasing results. In fact, in case the analysis focused on malware or particular p2p installation programs, the firewall might affect the quality of the results. Therefore, its usage is optional and strictly depends on the context of the analysis.

Sniffing capabilities

The second important task performed by the gateway is network sniffing. Providing sniffing capability within the gateway causes two major advantages. The first is about network topology: the sniffer is the only bridge between the internal network and the Internet. Thus, any connection to external hosts needs to pass through the gateway. Therefore, implementing the network monitoring and sniffing capabilities at gateway level represents the most logic choice. In fact, it is theoretically impossible for a guest machine to escape network sniffing under this network topology. In second instance, guest machine is absolved of monitoring low level network traffic. This feature is particularly relevant because low level network monitoring techniques are platform dependent. The given solution, instead, is platform independent. Therefore, this approach absolves guest machines of implementing network monitoring features and provides a uniform platform independent network monitoring layer.

In order to intercept all kinds of network interaction performed by guest machines, sniffing must capture traffic directly at OSI layer 2, i.e. MAC. By adopting such an approach, the gateway is able to intercept any communication between the guest machines and other hosts, at any layer above the MAC. In other terms, it is possible to intercept data in form of Ethernet frames (OSI L2), IP packets (OSI L3), TCP segments or UDP datagrams (OSI L4) and HTTP requests/response (OSI L7). However, this capturing process produces huge amount of data. In general the size of captured network data depends on the network activity processed by the guest machine. Roughly, each analysis session capture file might require from few tens up to several hundreds of megabytes to be stored on disk. Given those high space requirements, storing all the captured network files is not a convenient option. Moreover, some of the capture data is not relevant for analysis purposes (i.e. DHCP exchanges, conversation with host controller). For these reason, after capturing network data, the host controller delegates to a *network analyzer* some post-analysis operations, which will extract and summarize only the relevant information out of the captured network dumps, discarding original capture files.

On the other hand, network monitoring through sniffer gateway introduces new management overhead. In first instance, dedicated logic is needed in order to start and stop

network traffic sniffing. Moreover, the host controller needs to retrieve captured network data from the gateway. These requirements led to the implementation of a custom logic component, installed on the sniffing gateway, in charge of handling local configuration. The HTTP API provides a convenient management interface, letting host controllers to configure gateways and administrate sniffing tasks.

Figure 5.11 points out the interactions between an host controller, guest machines and a sniffer gateway. As pointed out by the sequence diagram, some specific interactions regard both the host controller and the sniffer gateway. In particular, the host controller starts a sniffing session on the gateway, before starting the guest machine. In this way, the gateway creates a sniffing process by applying convenient network filters (i.e. capture only traffic regarding that guest). Once the sniffer has been allocated, an acknowledge message is transmitted back to the host controller. At this point, the host controller starts the guest machine through the relative driver. Now on, every network interaction performed by the guest machine is catch by the sniffer. When the analysis ends, the host controller stops both the guest machine and its relative sniffer. Afterwards, network captured data is transfered from sniffer gateway to the host controller, which will submit that data to another entity, i.e. a network analyzer. This process may reside in the same host controller, in another machine or in the sniffer gateway itself. The main task of the network analyzer is to scan the captured data and to extract relevant information out of it, so that the rest of the captured data can be discarded.

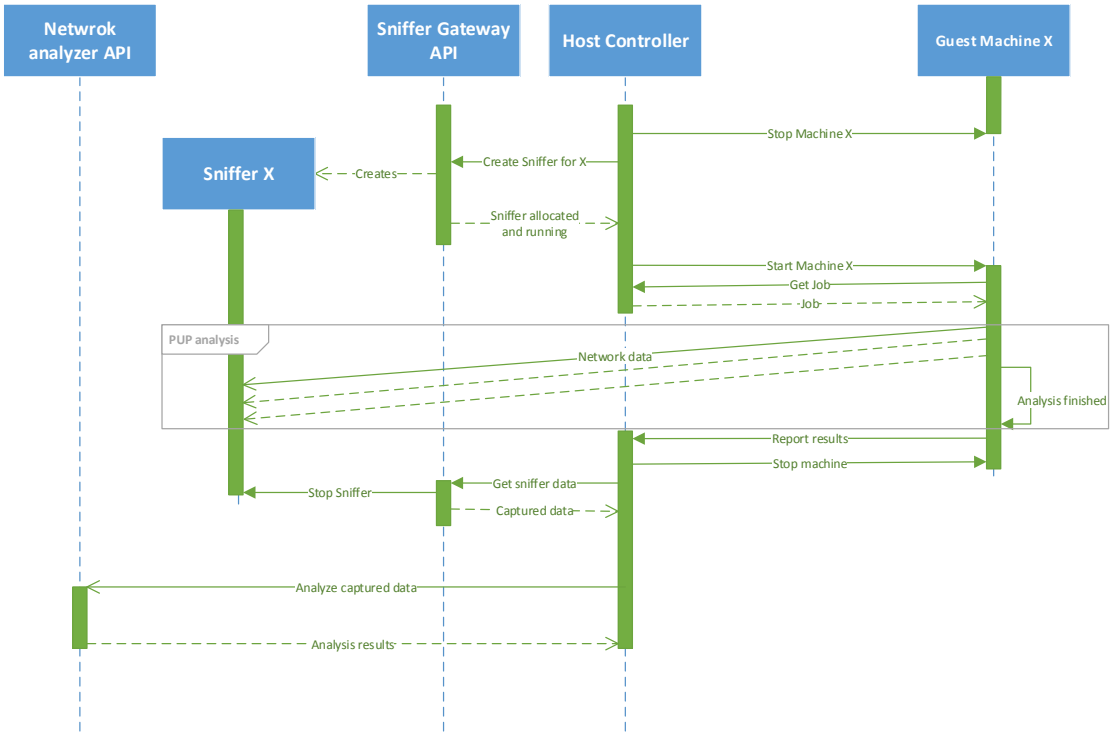


Figure 5.11: UML sequence diagram describing system interactions among Sniffing Gateway, Host Controller and Guest Machine

Gateway considerations

Delegating network sniffing and analysis to a dedicated device, external to the guest machines, has many advantages. We have already mentioned most of them in the previous sections. For sake of clarity, we will resume them here, stressing also possible drawbacks regarding the gateway structure.

In general, the gateway approach discharge the guest machine of monitoring low level network access. This speeds up performances of the guest machine. Moreover, being an external entity, the sniffer gateway does not expose any artifact to the guest machine: any process within the guest will not be able to identify any sniffing attempt. As an extra, there is no way for a guest machine to communicate to external network without being spotted. Therefore, observability property of the sandbox is truly improved.

From the point of view of extendability, the gateway provides a platform independent analysis layer: it exposes functionality via a web service and is capable of analyzing low level network data coming from any OS. Even though our target is Windows OS, the same approach will work with other operating systems. Moreover, it is worth to mention that part of the guest controller can run on custom hardware or be entirely virtualized. In particular hardware implementation of switches, NAT, firewall and DHCP can be used for achieving maximum performances. The trade off in this case is represented by the manual configuration overhead introduced by using custom hardware devices. On the other hand, the sniffer gateway may be implemented by a Linux machine, and used in almost any environment: bare metal or virtual machine. On our current implementation, the gateway is implemented completely via software, by relying on a virtualized Linux machine.

However, the given architecture introduces a possible performance bottleneck. A single network sniffer handles multiple guest machines, therefore its resources must be dimensioned consistently with the number of assigned guest machines. Also, available network bandwidth must be taken into account when deciding how many guest machine to assign to a single sniffer gateway.

Another important drawback to take into account is the impossibility to intercept end-to-end encrypted data or encrypted proxied connections (such as IPSec or encrypted VPN protocols). In such cases, the data flowing within a network stream is unreadable. Moreover it is hard to identify tunneled connections through an encrypted channels: in that case, only the network information of the front-end hop are available. Some workarounds are available for SSL traffic (using SSL proxies), but they do not offer any general solution for the end-to-end encryption issue.

Finally, external network sniffing does not intercept socket connections local to the guest machines. For instance, if two processes on the guest machines use local sockets as communication channel, no records of them will be present on the relative network capture file. However, given the details of windows operating system, it is possible to intercept those operation though API Hooking, therefore this drawback does not represent a limitation for our analysis system.

5.2.7 Bare metal and virtual environments support

One of the crucial properties of our design is the ability to run on both bare metal hardware and virtual environments. This unique characteristic improves the number of use cases of the infrastructure, providing a good defense against MSAS detection systems (when exploiting artifacts of the virtualized system). Running MSASs on dedicated hardware also improves performances. However, some design constraints have been introduced by

such a requirement. In particular, two aspects of the architecture were affected: *resource monitoring technology* and *messaging system* among the infrastructure nodes.

Resource monitoring is performed within the operating system of guest machines, thanks to API Hooking and DLL Injection. This design choice does not require any hardware support from the underlying layers. As a consequence, the logging system runs within any virtual or dedicated machine, independently of its hypervisor technology or underlying hardware infrastructure.

Communication among different nodes of the architecture relies on TCP/IP sockets. Therefore, both virtual machines and dedicated hardware can be used to run different nodes. Moreover, mixed environment are possible. For instance, in our specific implementation, network sniffers are virtualized, while host controllers run on dedicated rack servers.

However, virtual environments provide some management facilitations which are not natively available for bare metal machines. First of all, hypervisors usually expose commodity APIs to manage the life-cycle of its machines. Operations like *power on*, *power off* and *restart* can be scripted, thus automated easily. A classic computer, instead, requires custom hardware in order to be controlled remotely via software. Although server grade machines usually have such hardware support, they might be expensive and inefficient when uniquely dedicated to PUP analysis. In order to solve this problem, our design separates the logic of the host controller from the logic of the guest controller, by interposing a driver interface between them. When dealing with a particular hypervisor, this driver implements interface's specifications, by taking advantage of hypervisor's specific APIs. In the same way, a driver might be implemented to control the power cycle of guests, by using cheap network commanded power switches, as shown in Figure 5.12.

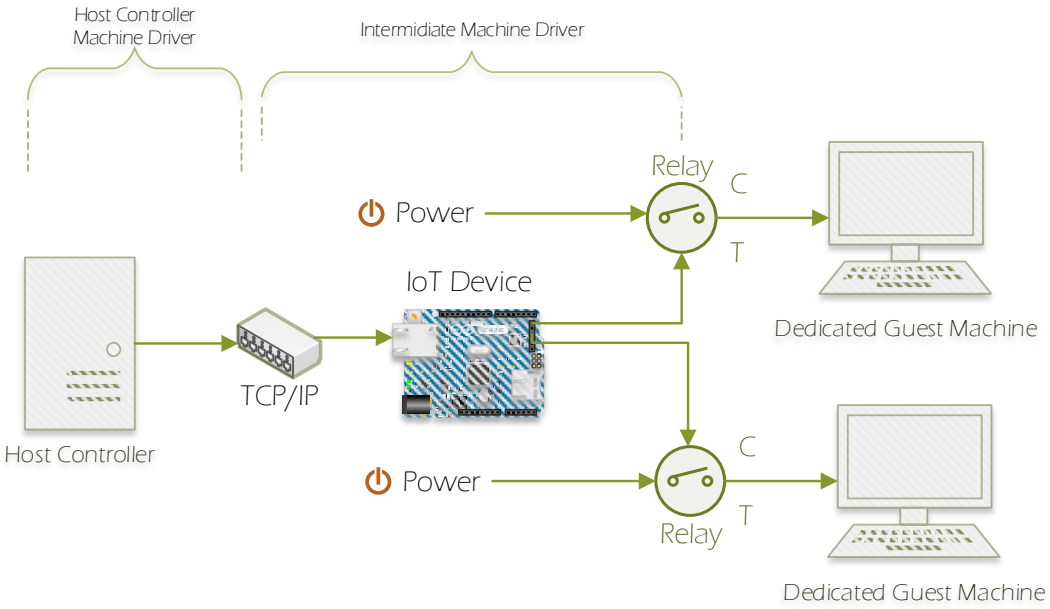


Figure 5.12: Using IoT device to manage power cycle of bare metal machines

Thanks to the spread of cheap *Internet of Things* (*IoT*) devices, implementation of network commanded switches is straightforward. In Figure 5.12 we propose a solution

based on a single Arduino Ethernet device[3], using simple relay switches to power on, power off and restart hardware guests. In this particular case, the machine driver is distributed: part of it is implemented in the software layer of the host controller, while the other part is implemented into the firmware loaded into the Arduino device.

Lastly, bare metal support introduces another technical challenge: automatically restoring the original known state after each analysis. While virtualization technologies support *rollbacking* and *snapshooting*, dedicated machines do not natively provide that capability. However, it is possible to workaround this problem by applying one of the following approaches:

- Backup and automatic restore
- Commercial software
- Ramdrive or ramdisks
- Diskless boot and virtual disks

A very easy way for implementing rollback capabilities is to use backups. Given an known image of a specific system state, we might restore that backup at every machine boot. However, this operation is hard to automate: it requires low-level scripting at bootloading time. Moreover, restoring a system backup may take a long time, depending on the read/write speed of the media. Other than that, every machine requires a dedicated disk, and power-cycle management must take care of clean shutdowns (preventing mechanical disk failures). Lastly, every update on the known state causes a great maintenance overhead to synchronize the backups. Commercial software has been developed to provide software solutions to this problem. However, the main drawback like power-cycle management, disk-image updates and backup restoring time still persist. By using dedicated hardware or reserving part of the RAM memory of each machine, it is possible to restore the known system state faster. Although improving boot time and system restoration time, this approach requires much memory or dedicated high-performance hardware, thus it becomes expensive. Moreover, synchronization of known image still represents an unsolved problem. In order to combine central state management, simple power-cycle management, performance and scalability, we designed a *diskless network-boot infrastructure*, shown in Figure 5.13. The idea is to store an immutable disk image within a central boot server, and to create n virtual differential disk images, one for each guest machine. Each machine boots via USB drive, which launches a *preboot execution environment* (PXE). Within the PXE, the machine executes a local http request to the boot server, asking for the image to boot. At that point, the boot server handles the request by allocating a differential virtual disk on the top of the base image, returning its iSCSI target address. Afterwards, the guest machine attaches the iSCSI target disk and performs a diskless boot. Figure 5.14 summarizes the various steps in this process, by pointing out interactions among the central server and the guest machines. The proposed solution allows central image management, enables base image *deduplication*, provides good performances and reduces costs. Whenever known states of guest machines need to be updated, it is sufficient to change the base image; all the guest machines will automatically synchronize their image at next boot. Also, this solution saves much storage space: the full disk image is stored in one single location and not duplicated among the guest machines. Moreover, guest machines can

[3]https://www.arduino.cc/

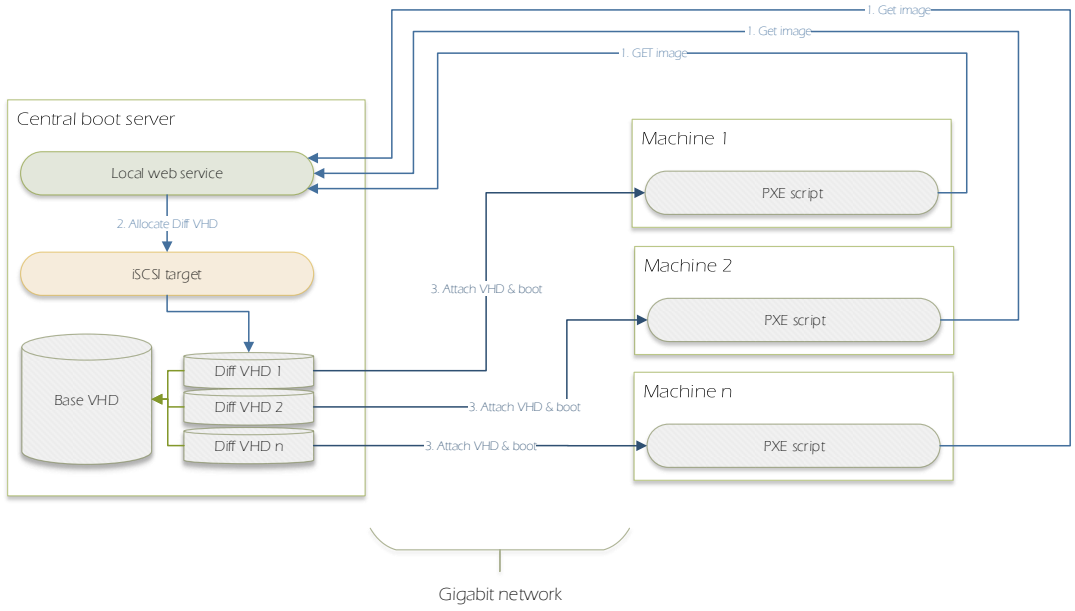


Figure 5.13: Diskless boot and virtual disk management architecture

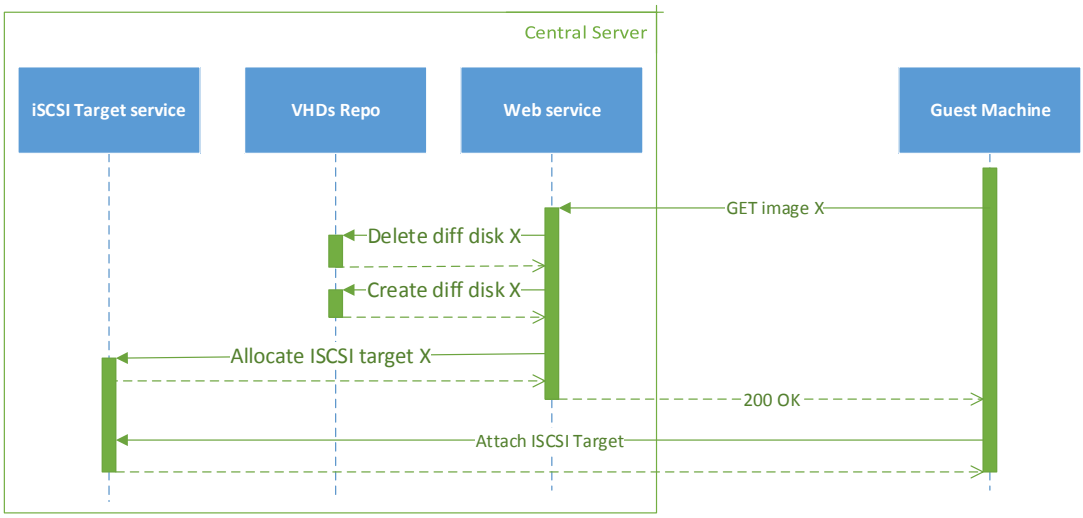


Figure 5.14: Diskless boot sequence diagram

run totally diskless. As a consequence, it is possible to hard-reboot those systems without any hardware hazard (because there will be no mechanical parts running), therefore the power-cycle management proposed in Figure 5.12 is safe to be used. Concerning the costs, this solution reduces expenses when compared to the other bare-metal approaches. It does not require hard disks for guest machines, neither any dedicated hardware; moreover is widely applicable to existing infrastructures. However, good network capabilities are mandatory: large bandwidth among guest machines and boot server is crucial to have

good performance. Also, the boot server has to implement iSCSI target services as well as virtual disk management services.

5.3 UI Interaction

The interaction with graphic user interfaces constitutes the key feature of PUP analysis automation. Differently from silent malware, PUP installers usually offer GUIs, in order to give the impression of legitimacy of the software [17]. Without interacting with those GUIs, only partial software behaviors are observable. When the analysis is targeting software installers, this constraint becomes even more restrictive. In other terms, GUI interaction is mandatory in order to perform analysis for application installers. Another important reason supporting the need of GUI interaction regards the information contained by the GUI itself. Indeed, researches demonstrated that it is possible to identify spyware by simply looking at the EULA they provide, at installation time [21]. Therefore, information concerning the GUI of an installer may be used as input for application classification, as support for other malware analysis methods.

5.3.1 UI interaction engine basic architecture

Given the scope of our work, the UI interaction engine is in charge of automatic software installation GUIs, by simulating an habituated and lazy user's behavior. In other terms, we designed a system that aims at successfully completing the installation procedure, minimizing the number of interactions with the GUI. To do so, it has to accomplish three different tasks, executed in loop, until the application installation is considered completed: *detecting* UI elements, *selecting* next interaction, *executing* the interaction. The whole process is described in Figure 5.15. The GUI interaction engine takes place within the guest machine, and it is implemented by the guest controller.

The first step performed by the engine is to check whether the process under analysis is still alive or not. If there is no process alive, the engine terminates. Otherwise it identifies and locates all the windows currently available on the desktop, owned by the processes under analysis. Processes might own many windows, but only one receives the keyboard and mouse focuses among all. Therefore, the detection engine tries to detect the window currently active, holding the focus. Once identified the active window, the engine performs the inspection by applying different techniques. The result of the inspection consists of a data structure containing information about the window and contained controls. For each detected control, the following information are collected, when available:

- Parent window (container)
- Position on screen
- Position on parent window
- Bounds of the control
- Contained text
- Type of the control
- Focus status

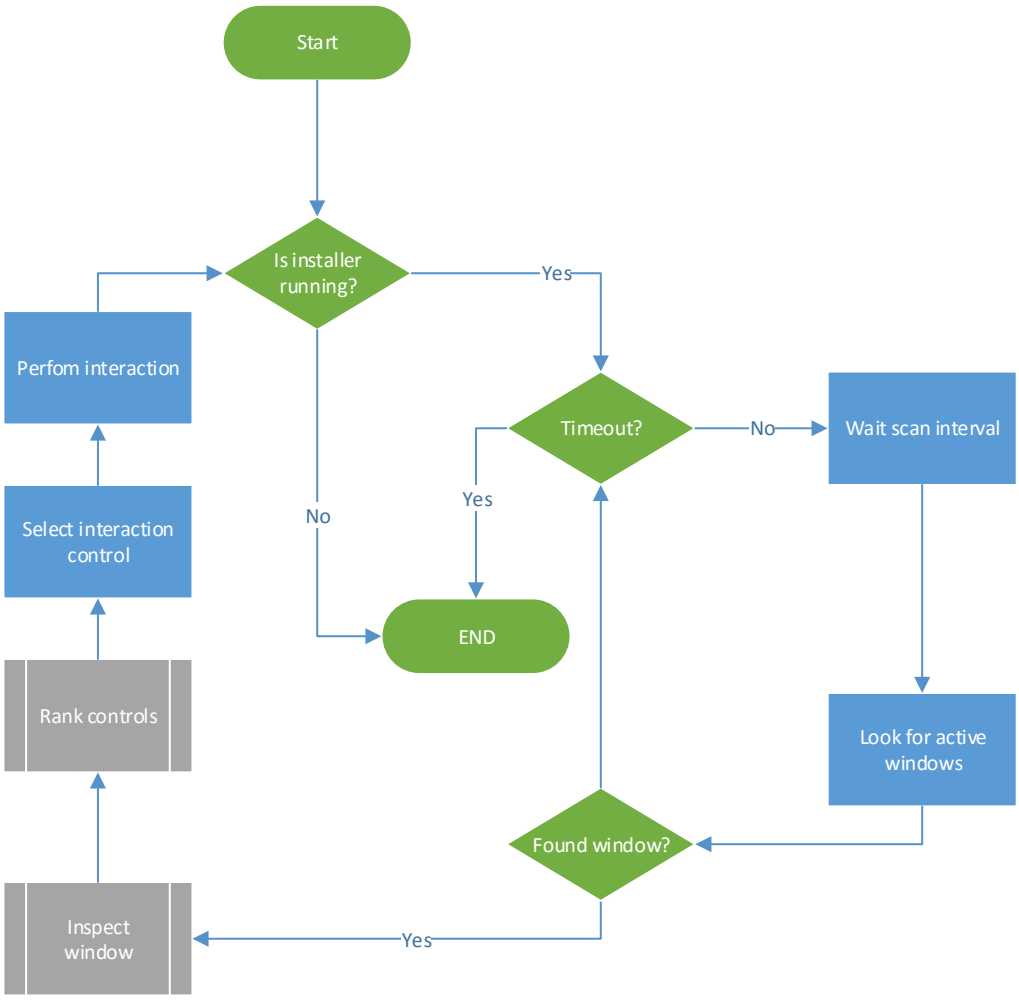


Figure 5.15: Control flow of the UI Interaction engine

- Enabled status or disable status

If no window is available or no data is collected, the interaction engine repeats the scan again, after waiting for a short period of time, giving a change to the installer process to build its UI. In case no UI is found within a certain timeout, the interaction engine gives up and terminates. On the other hand, when controls are found on a window, the engine applies a number of ranking policies, in order to select the best control to interact with. The ranking policies will then assign a score to each control listed, according to a number of heuristics. After that, the control totalizing the highest score is selected. So, the interaction engine tries to interact with the selected component, waits for its reaction and repeats the process again.

5.3.2 Automated UI interaction challenges

So far, the described portion of the interaction engine is pretty linear. However, the proposed scheme is not complete. In fact, there are a number of issues that the architecture had to take into account. Those issues inevitably increase the complexity of the GUI

interaction engine. In particular two major problems have to be considered: *when to perform the inspection* and *how to avoid UI loops*.

Given the dynamic nature of user interfaces, some controls may change their state just after the inspection process. If that happens, the ranking process might assign a wrong score to elements that have changed. Therefore, the engine might select a wrong control, or even worse, a control which does not exist any longer. Hence, the interaction will inevitably fail. For this reason, it is necessary to trigger the window inspection only when the window is stable, or better, waiting for the user's input. However, the problem now is shifted to detecting stable windows. Unfortunately, there is no programmatic general way to identify windows waiting user's action. Therefore, some heuristics have been applied, as reported in Figure 5.16.

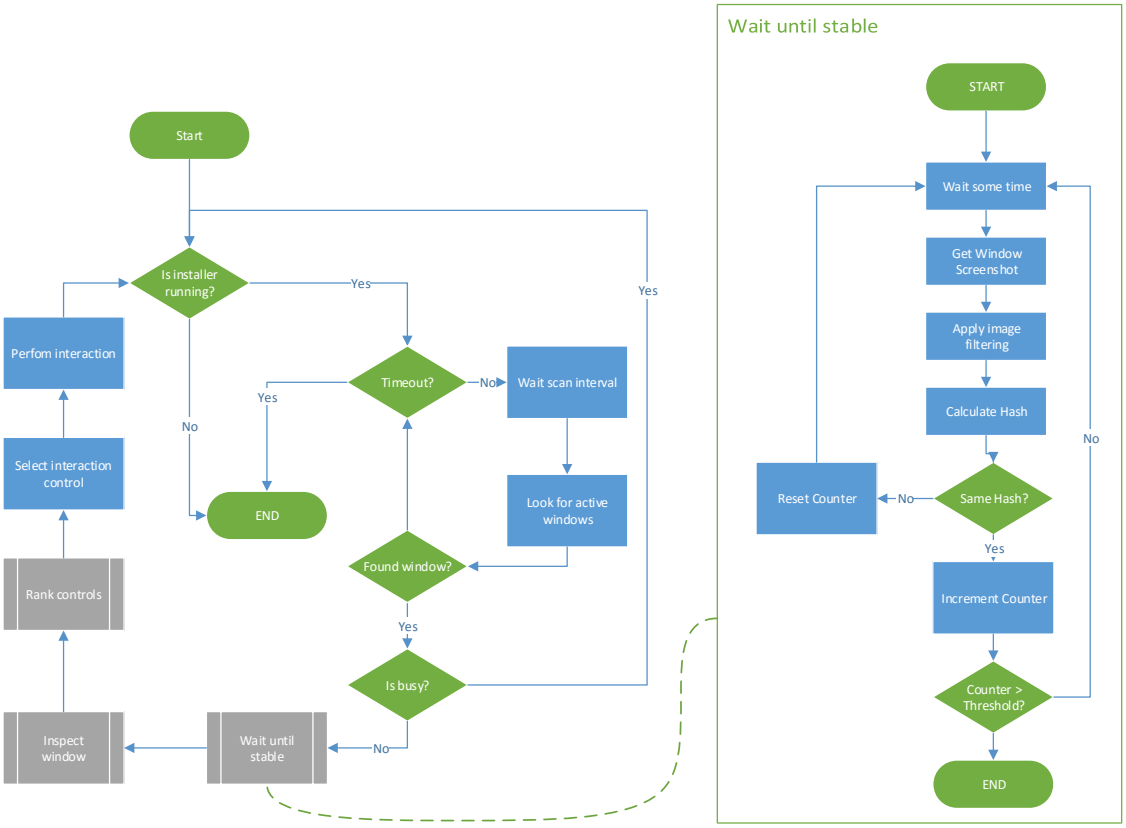


Figure 5.16: Flowchart of the UI interaction engine with UI stability checks

Banally, the interaction engine considers a window to be stable when it does not change in terms displayed graphics. If the graphics of the window remains the same for long enough, then the engine assumes the window is waiting for input. For this reason, a scanning loop is added to the process. At every iteration of this loop, the engine scans the active window and saves its current status. The status of the window is basically provided by its appearance, i.e. by its pixel matrix. To save memory among the iterations, the interaction engine takes a screenshot of the window's area, applies some graphic filters and calculates a cryptographic hash of the screen. Image filtering is necessary in order to reduce the details of the capture screenshot. In fact, animation effects (such as shadowing, blinking, etc) cause small color variation in the source pixel matrix. Those differences would led

to different hash calculation. Therefore, by applying adequate image filters, it is possible to remove animation noise, and calculating hashes only on relevant image characteristics. Figure 5.17 shows an example of this hash derivation process.

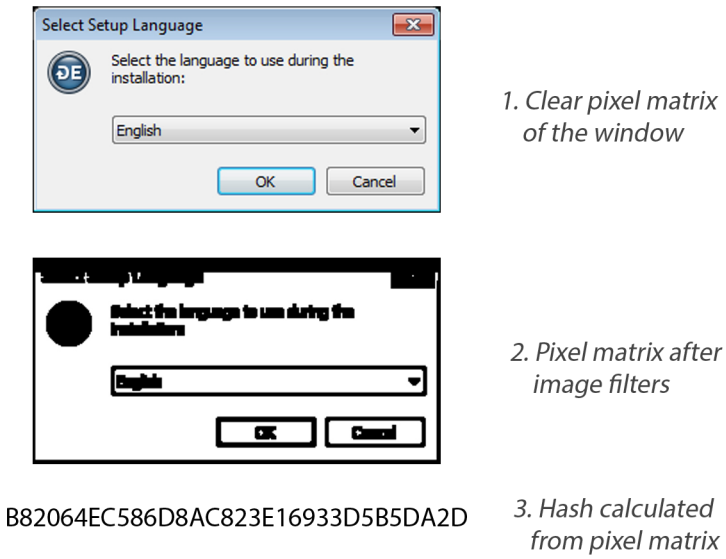


Figure 5.17: Example of hash derivation and screenhost manipulation

The hash is held as fingerprint of the current window status, which is compared at each iteration. If the status remains constant for a certain number of iterations, than the engine triggers the UI inspection and passes the results to the ranking system. However, there is no guarantee that the window will stay stable until the interaction happens. In the unfortunate case in which the window changes during the scan, an error will occur. Therefore, the interaction engine must be robust enough to detect those cases and to recover from them.

Yet considering UI timing challenges, another issue arises. When a single threaded installer process is performing heavy background calculation or waiting on IO, its GUI may be stuck. In such a case, the GUI may remain blocked for long enough to trick the GUI engine, appearing like a stable window. Therefore, if no countermeasure is adopted, the interaction engine might try to interact with an unstable window, causing undesirable behaviors. In order to avoid such a problem, the interaction engine performs some checks on the status of the guest controller, by looking at the log messages received by the HookingDLL. The workload of the process is derived by the frequency of the messages received by the dll injected into the analyzed processes: whenever this frequency is higher than an arbitrary threshold, the installer process is considered busy. As a consequence, the interaction engine queries the *log-ratio* value kept by the guest controller, and takes that value into account in the interaction process.

Problems also occur whether GUIs admit loops. A GUI loop is nothing more that a repetition of an already encountered window, as a consequence of a previous interaction. A very immediate example is provided by most of the application installers. When an user wants to abort an installation procedure, the GUI may ask for confirmation. If the user does not confirm, then the installer shows the previous window again. At this point, the same sequence of interaction may happen again, therefore the GUI interaction mechanism

might fall into an infinite loop. Figure **??** gives an explicit example of this possibility.

In order to detect and recover from loops, the interaction engine has to keep a list of already visited windows. This task is simply achieved by storing all the hashes of stable windows encountered. Each time a window is considered stable, its hash is added to an in-memory table, alongside a counter. If the hash is already present into the table, its associated counter is increased. When the counter exceeds an arbitrary threshold, then the interaction engine realizes it has fallen into a loop and gives up, terminating.

5.3.3 UI elements detection

After detecting windows belonging to the analyzed installer, the interaction engine performs an inspection of the contained UI elements. As explained in Section 2.6, detecting UI elements on Windows OS is not a trivial task. In particular, developers may build GUIs using standard Visual C++ libraries (or Winforms), they may use custom frameworks (such as Qt, Sciter) or use a combination of both. Each of those methods handles low-level UI messages in a distinct way, implementing the WndProc() functions of windows differently. Winforms and Visual C++ GUI elements are easy to identify, thanks to their known class names and standard message handling procedures. On the other hand, custom frameworks GUIs are much more complex. The most relevant problem with those frameworks depends on the way they combine multiple widgets into one single window. While Winforms follow a modular approach (declaring one window for each widget in the UI), frameworks like Sciter only declare one window, hiding the presence of child widgets behind it. In this way, functionality offered by child widgets become hard-coded into the WndProc() handler of the parent window. As a consequence, the OS is only aware of one window, and offers absolutely no way to identify sub-widgets contained in that.

Instead of developing ad-hoc systems to inspect GUI elements produced by each different GUI framework on Windows, we decided to design an hybrid system to identify UI elements. The interaction engines scans elements within a window by applying the following technologies:

- Microsoft UiAutomation library
- AForge.NET Image recognition framework
- Tesseract OCR engine

The whole process is described by Figure 5.18. As soon as the foreground window has been identified (i.e. active window), the interaction engine scans it using two concurrent approaches: one using UIAutomation library, another one using OCR and Image recognition techniques.

The left branch in the figure describes how logically simple is to perform the analysis with the UIAutomation framework. In this case, the library is able to identify all the native windows elements, with little developer's effort. For each detected widget, its data is then available through a convenient abstraction interface, called `UIAutomationElement`. Therefore, data extraction is straightforward.

On the other hand, UI recognition approach requires much more computational effort and a more structured logic. The first step is to retrieve raw pixel data information about the window being analyzed (digitalization). After that, the digitalized image is preprocessed through a series of filters using the AForge.NET image manipulation library. In particular the following filters are used:

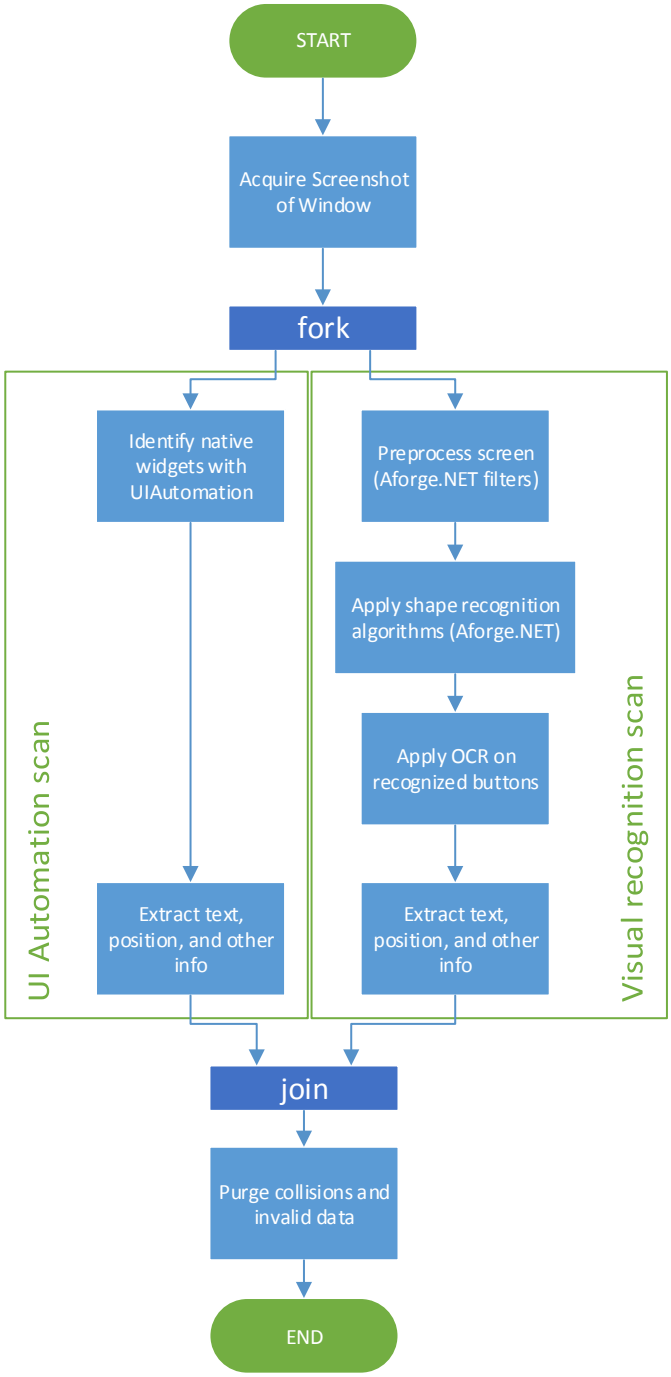


Figure 5.18: Flowchart about designed UI elements detection process

SISThreshold	Performs image thresholding according to [14]. Color information is then discarded.
FillHoles	Fills black holes in the image, resulting from noise [13].
Dilatation	Removes noise by filling gaps among shapes close one another [12]

Later on, the AForge.NET framework is used again, taking advantage of its shape recognition algorithms. In particular, the interaction engines looks for quadrilateral shapes, ideally representing buttons. At this stage, the only information available is the position of the detected shapes. Afterwards, the interaction engines processes all the potential buttons areas through Tesseract, which tries to extract textual contents out of the detected button. After that, a result set of potential UI elements is returned and combined with the results obtained by the UIAutomation library.

Applying separate recognition techniques to the same window may cause redundant results. For instance, both the scanning techniques would be able to detect classic UI buttons. Therefore, the produced list of results would contain duplicates. Thus, in order to reduce overhead and avoid ambiguity, the interaction engine performs some sanitizing actions on the result set. In particular, if both the scanning system have found elements with same bounds, the ones detected by the visual recognition system are discarded. The reason why we adopted this approach depends on the quality of information provided by the two scanning system. In general, UIAutomation is able to provide much more detailed information about the scanned components. In fact, the visual recognition techniques is only be able to extract the following:

- Owning window handle (through WinAPI)
- Position of the element on screen (through pattern recognition and WinAPI)
- Position of the element relatively to container window (through pattern recognition and WinAPI)
- Owning Process (through WinAPI)
- Contained text (through OCR)

On the other hand, UIAutomation provides much more detailed and relevant information. Other than the previously mentioned, we find:

- Control type
- Supported interaction patterns
- Specific control-type dependent information

Therefore, whenever overlapping results are detected by the two frameworks, it makes perfect sense to discard less detailed records in favor of UIAutomation results.

5.3.4 UI element selection

Once detected all the UI elements belonging to the active window, the interaction engine ranks all the items. Each UI element is assigned a score, calculated by applying scoring rules, based on its properties. Ranking rules are meant to provide higher scores to UI elements which would advance the installation procedure. Therefore, higher scores have to be assigned to elements which possibly push the installation process to the correct path.

Due to the recurrent patterns used during installation procedures of any kind, we decided to apply a series of ranking rules based on our experience. Each of the rules assigns a positive, negative or neutral score to a single property of element being analyzed. However, there may be rules evaluating properties not exposed by the element being analyzed. In

Property	Evaluation	Score when not available
Control is enabled	Enabled = 0 Disabled = -1000	0
Control contain text	Empty = -30 Exact match of whitelist = 280 Partial match of whitelist = 30 Exact match of blacklist = -280 Partial match of blacklist = -30	0
Position of control	Proportional to bottom right corner of the container window	0
Type of control	Button = 50 Checkbox = 15 Radiobutton = 15 Hyperlink = 10	0
Checked	Unchecked = 50 Checked = -100	0
Control is focused	HasFocus = 50 NoFocus = 0	0

Table 5.3: Ranking rules and assigned scores

List	Words
Whitelisted	next, continue, agree, accept, ok, install, finish, run, done, yes, i agree, i accept, accept and install, next >
Blacklisted	forward by small, amount, back, by small amount, back, by large amount, forward by large amount, disagree, cancel, abort, exit, back,<, decline, quit, minimize, no, close, pause, x, _, do not accept, <, back

Table 5.4: Comma separated blacklisted and whitelisted words

fact, detection through visual recognition is only capable of extracting basic properties, such as position and, possibly, contained text; UIAutomation elements provide instead much more details. Whenever an UI element does not provide a particular property, the assigned score would be 0.

Table 5.3 summarizes evaluated properties and assigned scores adopted by our current implementation. Among all the properties, two of them really impact on the ranking system: enabled/disabled status od the element and contained text. Whenever any UI widget detected through the UIAutomation library exposed the "disabled" property, its score will be penalized by 1000 points. This basically prevents any kind of interaction with it. On the contrary, if the item is enabled, no point is assigned. In order to evaluate the contained score of each control, the interaction engine applies both blacklist and whitelist approaches. In particular, there are two sub cases to take into account: when the contained text matches exactly a word or a sentence contained in the lists, or when the match is only partial. Whenever the word-matching is complete, a score of ±280 is given (+280 if whitelisted, -280 if blacklisted). If the text of the control matches only a subset of the whitelist/blacklist words, a score of ±30 is given for each matched word (+30 for whitelist, -30 for blacklist). Table 5.4 lists all the words used in both the whitelist and blacklist.

The following example helps the reader to understand how the ranking system operates within the interaction engine. Let Figure 5.19 represent a screenshot captured by the

Rule	Element 1	Element 2	Element 3
Control is enabled	NA = 0	NA = 0	NA = 0
Control contain text	NO = -30	NO = -30	"71 Install" = 30
Position of control	(24,21) = 6	(216,29) = 7	(205,368) = 10
Type of control	NA = 0	NA = 0	NA = 0
Checked	NA = 0	NA = 0	NA = 0
Control is focused	NA = 0	NA = 0	NA = 0
Total	**-24**	**-23**	**40**

Table 5.5: Drill-Down of scoring assignments for Figure 5.19

interaction engine, taken during the analysis of `Smart Defrag 5` installer. The figure stresses the elements identified by the interaction engine. Moreover, for each identified widget, the associated score has been printed. The item marked in red identifies the element with higest score, thus the one leading the result-set. The analyzed installer does not use classic windows widgets for its UI, therefore the UIAutomation was unable to recognize any item. In fact, three elements have been spotted by the image recognition tools.

Figure 5.19: Screenshot of first stable window during analysis of `Smart Defrag 5`

Table 5.5 describes how scores where assigned in relation with applied rules. The only widget totalizing a positive score is the Installation button, identified as `Element 3`. Indeed, this control contained the text *Install*, which has been recognized by the OCR engine as *71 Install*. Although the OCR recognition was not perfect, the scoring system assigned +30 points for the whitelisted contained word. Moreover, the position of the control contributed to gain other 10 points, for a total of 40 points. The other two elements, identified by shape recognition engine, did not represent any valid widget. In fact, the OCR was unable to extract any textual information out of them, therefore those elements were penalized by 30 points each. However, the position scoring system attributed +6 and

+7 points respectively to `Element 1` and `Element 2`. Eventually, neither `Element 1` nor `Element 2` got positive scores.

Once built a ranking table of elements, the selection is trivial. The interaction engine just pops out the top ranked UI item out of the list with a positive score, and interacts with that. If no items are available or all the items present a negative score, then the inspection is performed again. Every time a stable window is inspected and no control obtains a positive score, an internal counter is increased. Therefore, whenever that counter hits a threshold, the interaction engine gives up the inspection, allowing the GuestController to terminate the analysis.

5.3.5 Interacting with UI elements

A given widget may expose many possibilities of interaction. For instance, a checkbox can be selected or unselected, while button may be pressed, released or clicked. Automatically identifying which kinds of interactions are supported by scanned widgets is not trivial. The problem behind this challenge still concerns the custom UI elements. In fact, native winform controls implement a well defined set of interaction patterns, clearly recognized and exposed by the UIAutomation library. However, this library is not capable of recognizing which interaction patterns are supported by custom widgets. As a consequence, all the elements recognized by the image recognition engine are affected by this problem.

In general, the interaction with any kind of visual widget requires user's input. Thus, keyboard or mouse drivers are the main actors in such an environment. For this reason, interactive UI widgets have to handle at least one mouse or keyboard event. In case of a button, it usually can be clicked by pointing the cursor in the widget area, pressing the left mouse button and releasing it after a short time interval. In most cases, the same interaction can be performed by focusing the button and pressing the `ENTER` key on the keyboard. Checkboxes and radio buttons exposes similar behaviors: their internal state changes both when clicked (also focus and key press affect them).

The interaction engine takes into account all the available information of the scanned UI. Therefore it knows whether the selected widget belongs to the set of winforms components or it is a custom component. If the selected widget is not recognized by the UIAutomation library, the engine simulates the user's click event. The click event is triggered by sending two messages to the owning window, interleaved by 1 second pause: `WM_LBUTTONDOWN` and `WM_LBUTTONUP`. Both the messages require the coordinates of the mouse pointer to be expressed as parameters. Those are retrieved by simply calculating the center of the scanned widget, taking advantage of their position coordinates.

If the selected widget is recognized by the UIAutomation library, more complex and convenient interactions are possible. However, the engine only takes into account very simple actions, such as:

- Checkbox: check
- Radiobutton: select
- Button: click

The interaction engine behaves as shown in Figure 5.20. After performing one of the interactions, the engine waits for a reaction of the UI. This operation basically mimics the user's behavior: UI is expected to react to a successful interactions. From a programmatic point of view, this goal is achieved by applying the same window stability check described

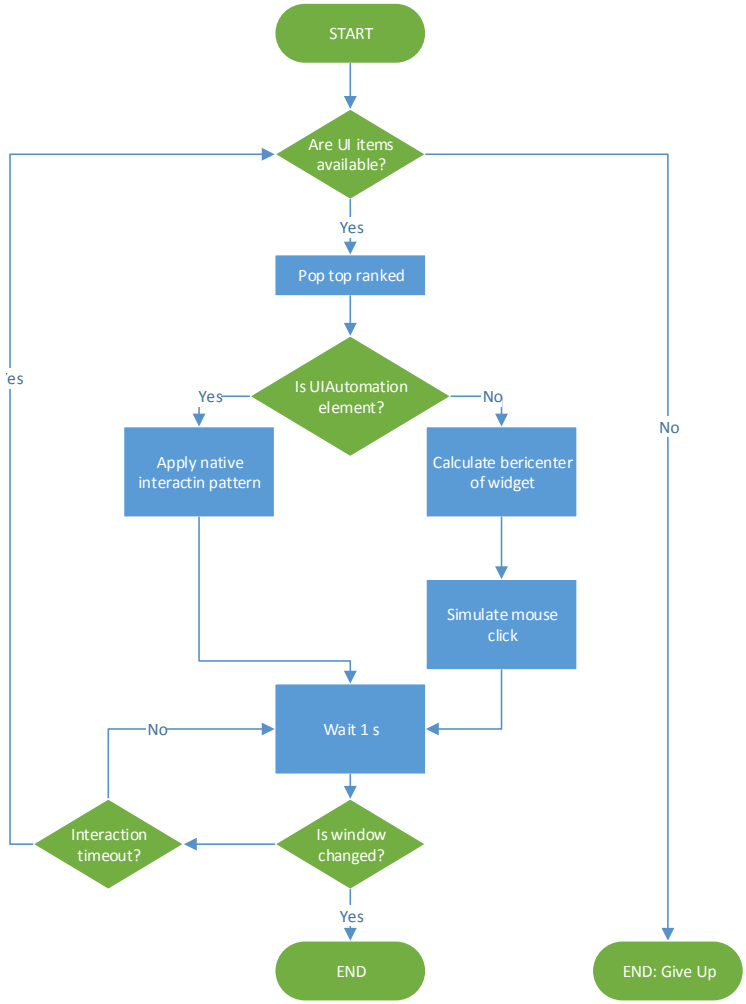


Figure 5.20: Flowchart describing UI interaction process

in Section 5.3.2. If no reaction is observed within a given time interval, the engine assumes the interaction has failed. Therefore, it pops another UI element out of the ranked UI widget list, and tries to interact with that. This loop is repeated until an interaction is successful or the ranked list is emptied. In both the cases, the interaction engine starts again the whole process described in Section 5.3.2; however, if no interaction was performed, the interaction engine increases an internal counter, called STUCK_INTERACTIONS. Whenever this counter hits a predefined threshold, the interaction engine gives up and allows the guest controller to terminate the analysis.

Chapter 6

Implementation details

In this chapter we present various technical aspects regarding the implementation details of the MSASs. The chapter begins with a description of the database technologies used in our implementation. In the same section we describe the synchronization issues arising in our distributed architecture and demonstrate a valid way for solving them. Afterwards, focus is given to the resource monitoring policies applied by the guests. In particular we briefly describe the Windows NT architecture and discuss possible approaches for monitoring access to operating system's resources. Later on we introduce the concepts of API Hooking and DLL injection, used for implementing the injector and the HookingDLL. The chapter ends with a detailed description of the technologies used for implementing the Sniffer Gateway.

6.1 Central database

The central database represents the fulcrum of the entire system architecture. It stores both inputs and outputs of the analysis, alongside transactional state for host controllers. The database is accessed directly, via network, by distinct host controllers and job crawlers: the former popping jobs to analyze, the latter pushing new jobs. Therefore, the synchronization among different nodes of the system heavily relies on this mechanism.

In our specific prototype we used *PostgreSQL 9.3*[1]: an open-source object-relational database system. It offers multiplatform support (Linux, Unix, Windows, Mac OSX, etc) and provides native programming interfaces for multiple programming languages, such as C/C++, Java, Python, ODBC and so on. PostgreSQL also complies to ANSI-SQL:2008 standard, offering advanced features for nested queries, transactions and advanced replication features (such as online hot-backup and asynchronous replication).

In order to facilitate and speed up the development of database queries, database access is performed through a special middleware: *SQLAlchemy 1.1*[2]. SQLAlchemy is an *Object Relational Mapping* (ORM) system, developed for Python language. Its main goal is to abstract the underlying DBSM, by providing a set of common features aimed at handling the storage of object oriented models. However, SQLAlchemy also provides native access to SQL quering mechanisms, when database specific capabilities must to be used.

[1] https://www.postgresql.org/
[2] http://www.sqlalchemy.org/features.html

6.1.1 DB Schema

In Section 5.2.3 we introduced the basic data model which the analysis system is based on. Thanks to SQLAlchemy, the implementation of such a db schema is straight forward and only takes few lines of code. For instance, Listing 6.1 points out the definition of the *Job* entity.

```
# ...
class Job(Base):
    __tablename__ = 'jobs'
    id = Column(Integer, primary_key=True, autoincrement=True)
    fname = Column(String(255), nullable=False)
    downlink = Column(String(4096), nullable=True)
    downdate = Column(String(4096), nullable=True)
    assigned = Column(Boolean, nullable=False)
    path = Column(String(4096), unique=True, nullable=False)
    md5 = Column(String(32), nullable=False)
    sha1 = Column(String(40), nullable=False)
    fuzzy = Column(String(150), nullable=False)
    aggregator = relationship(Aggregator)
    aggregator_id = Column(Integer, ForeignKey("aggregators.id"))

# ...
```

Listing 6.1: Source code of database library shared between host controllers and crawlers

Both crawlers and host controllers import the same database library, called *db.py*, that initializes all the SQLAlchemy objects and prepares the database for querying. Moreover, the db module directly imports the database configuration string from the *settings.py* module, which is also shared among the crawlers and the host controllers.

Once SQLAlchemy is loaded, the db schema is automatically generated, reflecting the model specified in the db.py module. The resulting schema obtained through SQLAlchemy is reported in details in Appendix A.

6.1.2 Multiple Host Controller synchronization

The system architecture supports multiple host controllers running simultaneously, boosting scalability and throughput of the system. However, this situation introduces the necessity of synchronization mechanisms, in order to mitigate accesses to shared data structures and to avoid race conditions.

According to the system design, race conditions among host controllers may happen in the following contexts:

- Popping a *Job*
- Popping a *Pending network analysis*
- Updating a *Worker*
- Updating an *Experiment*

When two host controllers concur in order to pop a job from the database, each DB driver performs a *SELECT... job* query against the centralized DB. If the two DB queries are started at the same instant *t0*, and no synchronization mechanism is applied, both the

host controller may receive the same job. This situation introduces two problems. The first is that the system will waste more resources to perform a single analysis. In fact, two or more guests may receive the same job id, thus they perform possibly identical analysis. The second problem is that the race condition is then reflected at the time when both the host controllers report back results for the same job. In this case, one of the two host controllers may detect the conflict and fail, or both of them may race again. Figure 6.1 graphically marks the race condition event, when no countermeasure is adopted.

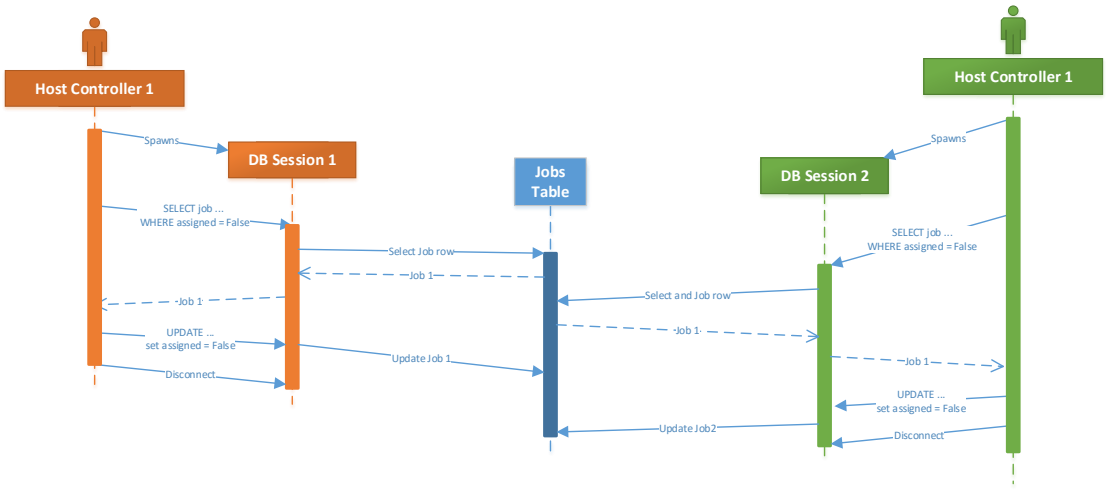


Figure 6.1: Race condition between two Host Controllers when popping a job. Both of them obtain the same job.

To fix this issue, we decided to add an ad-hoc flag in the structure of the job table, named *ASSIGNED*. Moreover, the select query has been implemented using a transaction with *Read Committed* isolation level. This isolation level is obtained by using the structure "SELECT ... FOR UPDATE", which adds a write-level lock into the selected rows, preventing dirty reads by other instances. Once the selection successes, the ASSIGNED flag relative to the selected job is raised. This approach basically causes the serialization of select queries over the job table, as shown in Figure 6.2

The same approach has been used to solve the other possibilities of race condition: a transaction is started with *read committed* isolation level, ensuring only one entity at a time may modify the transient state.

6.2 Resource monitoring implementation

Observability capabilities of guest machines are implemented via software. The main idea is to monitor a set of processes under analysis, auditing access performed against valuable assets of the system. In particular, a few resources are particularly interesting for our analysis: file system, network and registry.

In order to understand how to intercept resource access of a given process to a specific asset, it is necessary to inspect the architecture of the target OS. Therefore, in this section we introduce the basics regarding Windows NT architecture and then we explain how our specific auditing approach works thanks to *API Hooking* and *DLL Injection*.

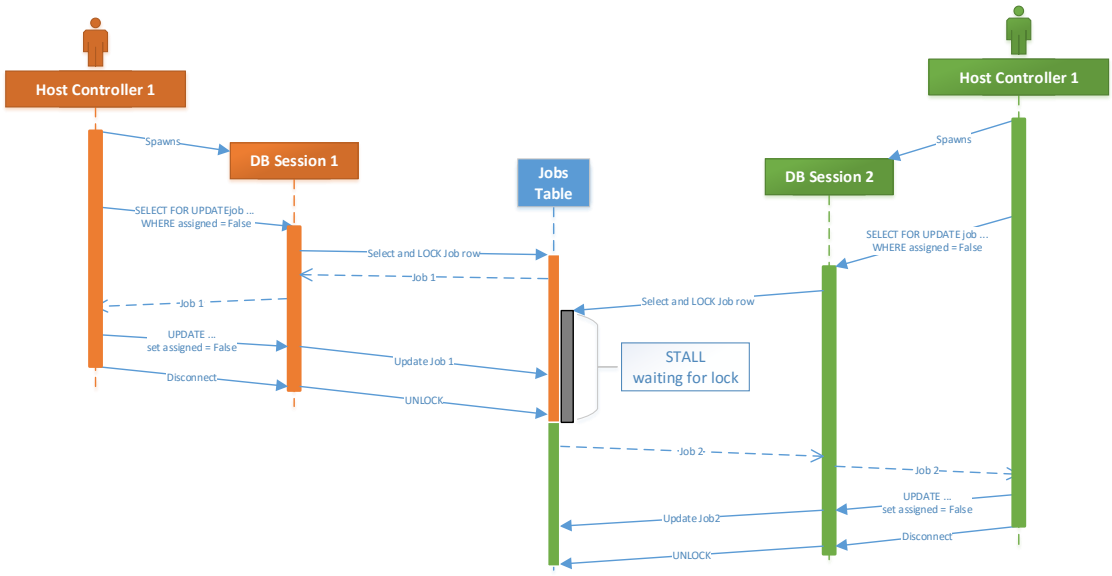


Figure 6.2: Race condition resolved by locking rows being updated

6.2.1 Shared resource access in Windows NT

The way a process can access a particular asset strongly depends on the architecture of the operating system. In general, any recent OS provides mitigation mechanisms for accessing shared assets, eventually providing security enhancements [76]. Given a particular hardware asset, operating systems tend to expose different software abstractions of the hardware interface, by providing distinct API libraries for the developer. Those API sets usually differ one another for their level of abstraction, implemented security features, ease of usage. Moreover, OSes usually follow software modularity principles, enabling stacked module architectures for accessing low level assets.

Windows 7 follows the NT architecture, described by Figure 6.3[71]. The lowest software interface, able to directly deal with hardware resources, is the *Hardware Abstraction Layer* (HAL). This layer of code provides very basic software interface between kernel objects and hardware specific technologies. On the top of the HAL, multiple *device drivers* are stacked. Drivers provide translations of interface I/O function calls into specific hardware requests. Moreover, drivers can be stacked, providing multiple layers of abstraction at each stage. Other kernel entities of the system make use of drivers in order to expose functionality to the upper user space. Interaction between kernel services and user-mode processes is enabled by a trap handler mechanism, implemented via software interrupts. Whenever a user-mode application requests access to a resource asset, a software interrupt is triggered (through the *sysenter* instruction). Therefore, a context swap is performed and the control flow passes to the dispatching service, running in kernel mode. Whenever the requested operation is concluded, the control is returned back to the user-mode calling application, after executing another context switching.

Due to the development overhead caused by such context switching protocol, Windows offers commodity libraries that facilitate the interaction with kernel objects from within user-space. The `ntdll` library represents the lowest API entry point available within user-space. However, applications running in user-mode do not typically call ntdll functions directly; instead, they rely on Win32 routines [60]. Win32 libraries provide an higher

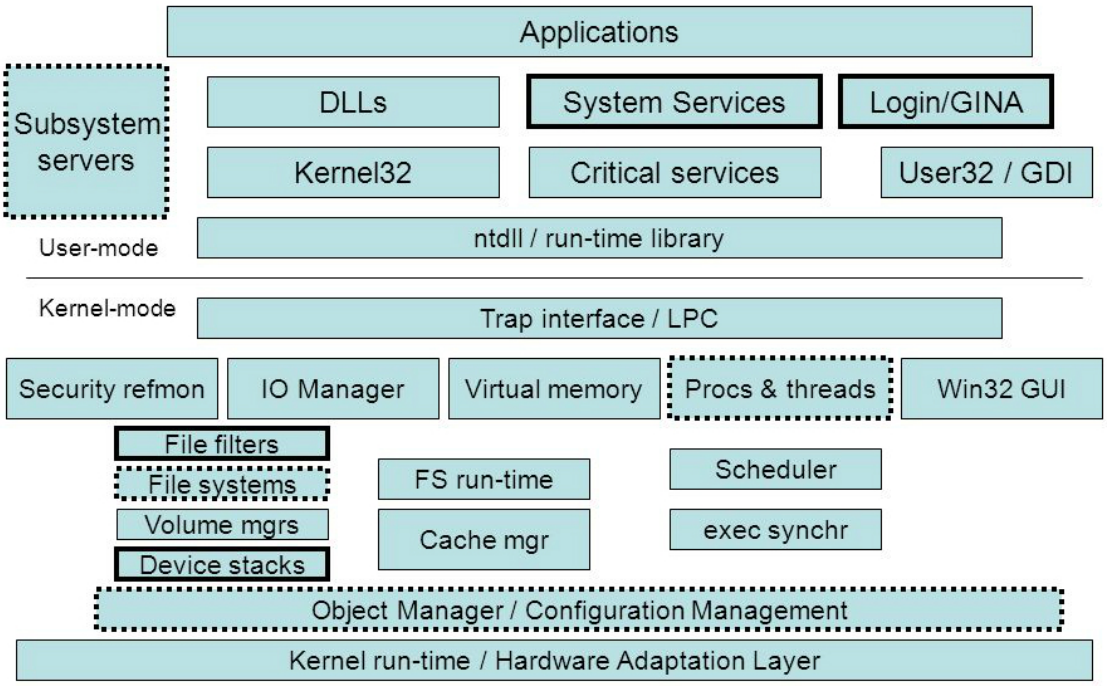


Figure 6.3: Windows NT architecture, Microsoft©

level of abstraction, implementing more generic functionality by dealing with specific ntdll implementations independently of the version of the operating system. There are a number of reasons why programs usually rely on Win32 libraries instead of linking directly with the native ntdll module. The most important regards the stability of the Win32 interface over the years: Microsoft develops Win32 carefully, putting much effort in holding the interface stable. Therefore, when a bug is fixed or a functionality is improved, the Win32 APIs tend to remain the same, but ntdll interface may change. Moreover, Win32 APIs take care of many low level aspects which are not directly addressed by ntdll functions. Lastly, many ntdll APIs are not officially documented by Microsoft [36].

Given a process running in user-space, its interaction with a resource asset happens through different layers. To summarize, from the highest abstraction layer, we find:

- User Mode
  - Application libraries
  - Win32 libraries (user32, kernel32, etc.)
  - Ntdll library
- Kernel Mode
  - Filter drivers
  - Low level drivers (hardware dependent)
  - HAL

One concrete way of monitoring resource access is to intercept all the API calls provided by one or more libraries in the given stack. In particular, the lower is the level of

APIs intercepted, the more difficult is for an application to escape the interception layer. Therefore, a drastic approach is to implement specific drivers for underlying hardware, and load them at the bottom of the driver-stack, just above the HAL. However, several problems arise when dealing with that a way. In first instance, driver implementation requires knowledge of the underlying hardware and associated communication protocols. Moreover this approach is not robust to hardware changes: whenever a component is updated, drivers might require to be updated. Another relevant drawback of intercepting resource access at this level is the lack of abstraction. Drivers are usually stacked [70] and each one is addressing a specific function set. Thus, lowest drivers in the chain only deal with low level information, while higher drivers in the chain are aware of high level information. As an example, we might consider disk drivers and the file system. Whenever an user-space process requests write access to a device, the IO Manager (a kernel object in charge of handling IO) passes the request to the file system driver. This request contains few parameters: a file pointer, an offset and the buffer to be written. The FS driver calculates the position of data to be written, resulting in volume-relative byte offset. Then it calls the next driver in stack. At this point, the disk driver will translate the volume byte offset in sector and cylinder coordinates, used for storing the buffer on the disk. As a result, the lowest driver has lost the information of *which file* is being written, while it was available to the higher ones.

A good trade-off would be to intercept high-level drivers at the top of the stack. For this purpose, Microsoft provides commodity driver classes, such as file-system and registry filter drivers [58]. A file system filter driver gets loaded just above the file system driver, therefore it can *filter* all the request before passing them to the next driver in the chain. In the same circumstances of the previous example, this approach would provide a good level of abstraction (the filter driver will be aware of the file being accessed). However, there still are a couple of drawbacks when using this strategy. First of all, in case the IOManager performs optimizations, caching or simply refuses to handle a request coming from user space, the driver does not get called. Therefore, such as system would not be able to intercept access failures in those cases. Moreover, a driver loaded in the system may represent an artifact of the system, that analyzed software might identify relatively easily.

In order to find a good trade-off among the previously described situations, we decided to adopt a third strategy. The idea is to add a software interception layer in between the user space and the kernel space. More specifically we targeted the API exposed by *ntdll* module. Every process running on windows NT is required to load this module. In fact, the system throws a hard error if a process is run without loading the ntdll module [30] in its memory space. Our approach consists in modifying the memory-space of each process in such a way that specific code is executed *before* and *after* ntdll function calls. Thus, the added code provides monitoring and logging functionality, directly from within the running processes. The technique we have used to *de-route* ntdll function calls to our custom binary code is technically defined as *API Hooking*.

6.2.2 API Hooking: Overview

API Hooking is a technique that enables the interception of function calls [84]. There are many possible ways to perform such interceptions. On Windows, we might identify the followings:

- Import/Export Address Table Hooking

- SSDT Hooking
- Inline Hooking

The first hooking mechanism is an user-space oriented approach. Its idea is to modify the *Import Address Table* (IAT) or the *Export Address Table* (EAT) of a given process. The IAT maps imported functions of an external DLL to their relative memory address. This binding is performed by the Windows Loader before the executable is run. IAT Hooking consists in modifying such a table, so that the address of a loaded function is overwritten with the address of another arbitrary function [75]. Although this hooking method is easy to apply, its efficacy is poor. In fact, IAT hooking is easily detectable [75] and more advanced hooking systems are usually preferred. Moreover, IAT Hooking is ineffective when the target process uses runtime dynamic linking via LoadLibrary() and GetProcAddr(). On the other hand, this hooking system allows to selectively hook only the functions of interests for a given process. That is because each process has its own private IAT.

The second hooking approach, performed within kernel space, aims at patching the *System Service Dispatch Table* (SSDT). The SSDT represents a kernel data structure, mapping system call routines to their relative memory addresses [80]. Whenever a system call is requested by an user-space process, the SSDT is scanned in order to find the memory address of the system call to execute. Thus, it is possible to change the address of specific system calls, obtaining the same hooking effects described previously. Since kernel memory is shared to all the kernel objects, a driver would be able to access the SSDT. Thus, a simple way of patching SSDT is to develop a driver that, at load time, allocates hooking functions and overwrite the addresses of the SSDT. Although SSDT is write protected, there are some strategies allowing to workaround that limit [75]. Eventually, this SSDT method patching methods have been tackled by Microsoft with modern 64bit operating systems [59], but still affect 32 bit OSes. The main problem of SSDT is the overhead inducted to the entire system. In fact, once the SSDT has been patched, every process calling an hooked syscall will run hooked function. A workaround is to add some logic into the hooked function, so that the standard syscall is called based on certain circumstances.

The third method is called *Inline Hooking*, and corresponds the hooking approach adopted in our system. The basic idea behind inline hooking is to overwrite the first instruction of the function to hook, so that arbitrary code is executed. To do so, a series of operations are necessary. Let `FUNC` be the hooked function and `HOOK` the hook function. In order to hook `FUNC` and execute `HOOK` before it, inline hooking requires to:

1. Copy the first bytes of `FUNC` into a new memory location, `STUB`
2. Overwrite the first bytes of `FUNC` with an unconditional JUMP to `HOOK`
3. Append a JUMP instruction to `HOOK`, bringing control back to `STUB`

Figure 6.4 graphically shows a comparison between a standard function call (A) and an inline hooked function call (B). The very first part of the function (A) constitute the *prologue*, which is in charge of preparing the stack and registers before executing the logic of the called function. In a normal context, when calling this function, the prologue is executed, then the control passes to the real logic of the function (unspecified in the image). After applying the inline hooking, the memory appearance changes like in Figure 6.4 (B). The prologue is overwritten with an unconditional JUMP instruction, which immediately

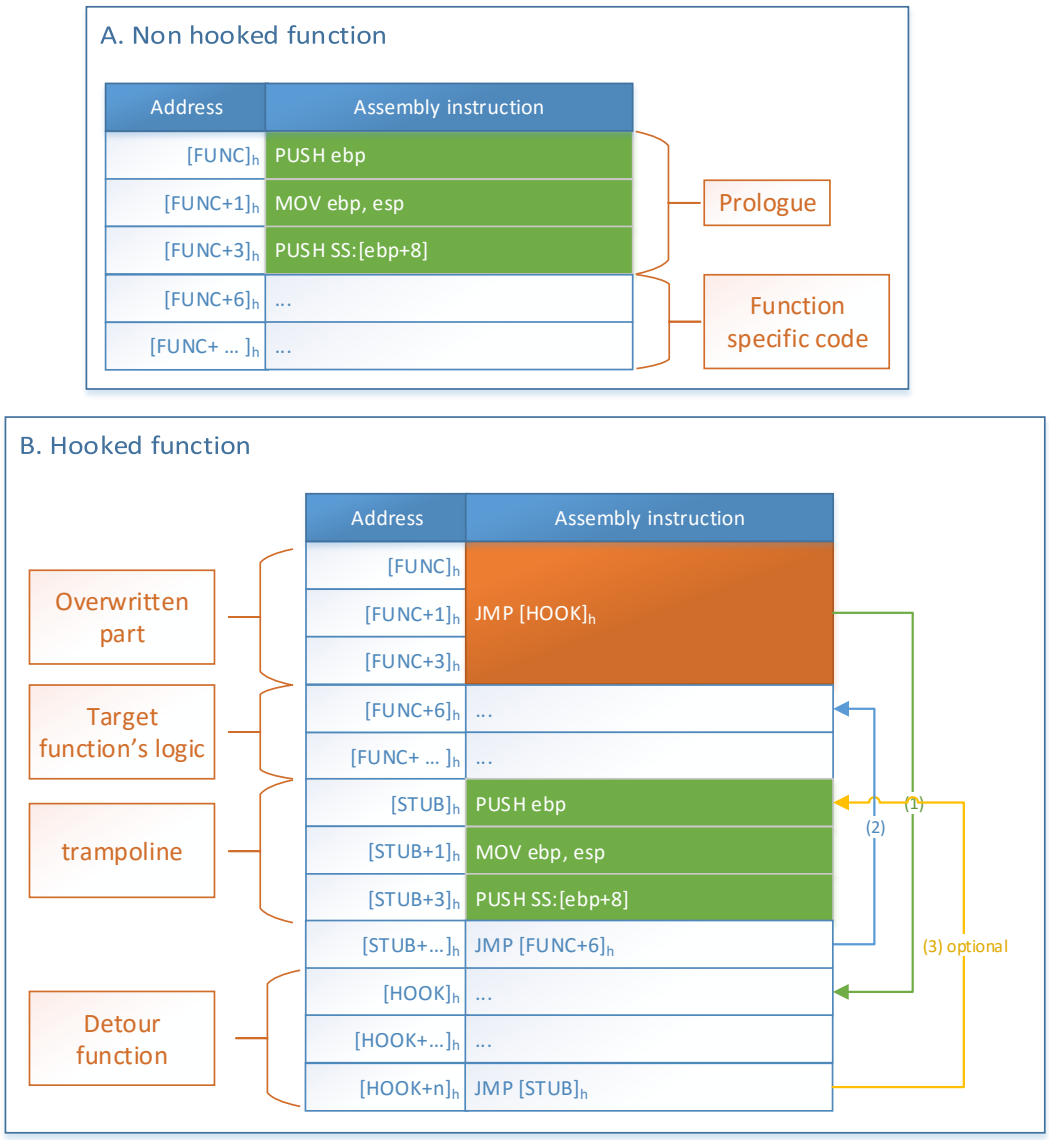


Figure 6.4: Inline hooking from memory perspective

brings the control to another user-defined function, called *detour function* (in our case `HOOK`). The detour function implements custom logic, often aimed at logging or profiling function calls. The detour code, after performing profiling or logging, might execute the original targeted function. This is done by executing the *trampoline function*, i.e. the copied version of the original prologue (in our case a copy of it, `STUB`). Hence, the trampoline prepares the stack and then invokes the unmodified target function.

Inline hooking is mostly applied in user space, and does not suffer of any drawback regarding DLL load strategies. In fact, in contrast with IAT hooking, inline hooking overwrites library functions after they have been loaded into the memory. Thus, inline hooking serves well both with static runtime and load-time dynamic linking. Moreover, in contrast with SSDT hooking, inline hooking does not affect the entire system. In fact, inline

hooks can be applied selectively to an arbitrary set of processes, chosen by the developer.

The hooking mechanism we chose for our architecture belongs to the third family: inline hooking. More specifically, we rely on the Microsoft's *Detours* library [39]. Detorus implements inline hooking by preserving the target function unaltered, patching the libraries directly in memory at runtime. Indeed, the interception mechanism used by Detours is guaranteed to work regardless of the method used by the application to locate the target function [39].

6.2.3 Dll Injection & Injector

In order to perform inline function hooking, it is necessary to inject a portion of code into the target process, so that its memory space can be modified. The idea of injecting code into another process is called *process injection* [75]. There are two different ways for performing process injection under windows: *direct injection* and *dll injection*. While direct injection consists in writing all the pre-compiled code to be injected directly into the remote processes, dll injection is more convenient in our case. In fact, dll injection represents the most used technique to perform process injection [75]. The basic idea consists in injecting only a few instructions into the target process, aimed at loading an external DLL. The external DLL contains the whole logic to be injected into the remote process. To do so, the following actions have to be taken:

1. Obtain an handle to the target process (if already spawned)
2. Allocate enough memory in the target process for storing the dll path name to load
3. Write the dll path into the allocated space
4. Make the target process to execute the LoadLibrary() function

---

```
1
2 // ...
3
4 // Perform injection:
5 if (processCreated) {
6   // Allocate enough memory on the new process
7   LPVOID baseAddress = (LPVOID)VirtualAllocEx(lpProcessInformation->↩
        hProcess, NULL, strlen(DllPath)+1, MEM_RESERVE | MEM_COMMIT, ↩
        PAGE_READWRITE);
8
9   // Copy the code to be injected
10  WriteProcessMemory(lpProcessInformation->hProcess, baseAddress, DllPath, ↩
        strlen(DllPath), NULL);
11
12  kern32dllmod = GetModuleHandle(TEXT("kernel32.dll"));
13  HANDLE loadLibraryAddress = GetProcAddress(kern32dllmod, "LoadLibraryA");
14  if (loadLibraryAddress == NULL)
15  {
16    OutputDebugStringW(_T("!!!!!LOADLIB IS NULL"));
17    return 0;
18  }
19
20  // Create a remote thread into the target process and trigger the ↩
        LoadLIbraryA execution
```

```
21   HANDLE  threadHandle = CreateRemoteThread(lpProcessInformation->hProcess,↩
         NULL, 0, (LPTHREAD_START_ROUTINE)loadLibraryAddress, baseAddress, ↩
        NULL, 0);
22   if (threadHandle == NULL) {
23     OutputDebugStringW(_T("REMTOE THREAD NOT OK"));
24   }
25
26   OutputDebugStringA("Remote thread created");
27
28   ResumeThread(lpProcessInformation->hThread);
29 }
30
31 //...
```

Listing 6.2: An example of dll injection

Listing 6.2 demonstrates a simple way of performing dll injection on Windows. At first we allocate enough memory in the target process, aimed at storing the path for the dll path to load. This operation is performed via `VirtualAllocEx`. Afterwards we write the path into the previously allocated memory. Later on we retrieve an handle to the kernel32 library, via `GetModuleHandle`. At that point, using `GetProcAddr`, we obtain the relative offset of the `LoadLibrary` routine within the kernel32 module. Then, we start a new thread in the remote process. The entry point of the thread is the LoadLibrary function, having the address of the dll to load as first parameter. Finally, we start the thread we allocated.

```
1  BOOL WINAPI MyDetourCreateProcessWithDll(LPCSTR lpApplicationName,
2    __in_z LPSTR lpCommandLine,
3    LPSECURITY_ATTRIBUTES lpProcessAttributes,
4    LPSECURITY_ATTRIBUTES lpThreadAttributes,
5    BOOL bInheritHandles,
6    DWORD dwCreationFlags,
7    LPVOID lpEnvironment,
8    LPCSTR lpCurrentDirectory,
9    LPSTARTUPINFOA lpStartupInfo,
10   LPPROCESS_INFORMATION lpProcessInformation,
11   LPCSTR lpDllName,
12   PDETOUR_CREATE_PROCESS_ROUTINEA pfCreateProcessA)
13
14   PROCESS_INFORMATION pi;
15   LPCSTR rlpDlls[2];
16   DWORD nDlls = 0;
17
18   // Configure flag mask and set SUSPENDED bit.
19   DWORD dwMyCreationFlags = (dwCreationFlags | CREATE_SUSPENDED);
20
21   // If no custom function has been passed to create the process, use ↩
         CreateProcessA
22   if (pfCreateProcessA == NULL) {
23     pfCreateProcessA = CreateProcessA;
24   }
25
26   // Invoke process creation
27   if (!pfCreateProcessA(lpApplicationName,
28     lpCommandLine,
29     lpProcessAttributes,
30     lpThreadAttributes,
31     bInheritHandles,
```

```
32      dwMyCreationFlags,
33      lpEnvironment,
34      lpCurrentDirectory,
35      lpStartupInfo,
36      &pi)) {
37        return FALSE;
38    }
39
40    // Copy the dll path string pointer into the array of dll to inject
41    if (lpDllName != NULL) {
42      rlpDlls[nDlls++] = lpDllName;
43    }
44
45    // Inject the DLL into paused process
46    if (!DetourUpdateProcessWithDll(pi.hProcess, rlpDlls, nDlls)) {
47      TerminateProcess(pi.hProcess, ~0u);
48      return FALSE;
49    }
50
51    // Copy back info about the created process
52    if (lpProcessInformation) {
53      CopyMemory(lpProcessInformation, &pi, sizeof(pi));
54    }
55
56    // Resume thread execution, so process continues.
57    if (!(dwCreationFlags & CREATE_SUSPENDED)) {
58      ResumeThread(pi.hThread);
59    }
60
61    return TRUE;
62  }
```

Listing 6.3: Dll Injection procedure

Listing 6.3 shows part of the source code characterizing the *injector* module of guest nodes, which is in charge of performing dll injection. The injector requests to the OS to prepare a process for execution (Line 27), but to leave it in suspended status(Line 19). Then, dll injection is performed indirectly, using `DetoursUpdateProcessWithDll()` (Line 46). That function basically implements some checks against the target process and performs the needed operation to load the HookingDLL into the remote process, similarly to what is shown in Listing 6.2. Afterwards, the code takes care of storing information about the new created process (such as thread ID and Process ID). Eventually, the process execution is resumed, by calling `ResumeThread` (Line 58).

6.2.4 Clooser look at HookingDLL

So far we have briefly introduced API Hooking and DLL injection techniques, used in our implementation. However, the fulcrum of the API Hooking system is implemented within the HookingDLL, which uses Microsoft Detours and is injected via the injector.

```
1  INT APIENTRY DllMain(HMODULE hDLL, DWORD Reason, LPVOID Reserved)
2  {
3    // Get the module in which there is the function to deroute, which is ↩
       NTDLL.DLL
4    ntdllmod = GetModuleHandle(TEXT("ntdll.dll"));
5    kern32dllmod = GetModuleHandle(TEXT("kernel32.dll"));
```

```
6   wsmod = LoadLibrary(TEXT("wsock32.dll"));
7   ws2mod = LoadLibrary(TEXT("ws2_32.dll"));
8
9   string tmplog;
10
11  // Now, according to the reason this method has been called, register or ↩
        unregister the DLL
12  switch (Reason)
13  {
14    case DLL_THREAD_ATTACH:
15      OutputDebugString(_T("THREAD ATTACHED TO DLL."));
16    break;
17
18    case DLL_THREAD_DETACH:
19      OutputDebugString(_T("THREAD DETACHED FROM DLL."));
20    break;
21
22    case DLL_PROCESS_ATTACH:
23
24      // Initialize the TLS index. Every allocated slot is guaranteed to be↩
            initialized to 0. So we don't need to initialize them.
25      if ((dwTlsIndex = TlsAlloc()) == TLS_OUT_OF_INDEXES)
26        return FALSE;
27
28      tmplog.append(_T("Attached to process"));
29      tmplog.append(to_string(GetCurrentProcessId()));
30      OutputDebugString(tmplog.c_str());
31
32      DisableThreadLibraryCalls(hDLL);
33
34      // Set the error mode to NONE so we do not get annoying UI
35      SetErrorMode(SEM_FAILCRITICALERRORS | SEM_NOGPFAULTERRORBOX);
36      _set_abort_behavior(0, _WRITE_ABORT_MSG);
37
38      // The following is needed for performing the lookup of opened query ↩
            keys
39      realNtQueryKey = (pNtQueryKey)(GetProcAddress(ntdllmod, "NtQueryKey")↩
            );
40
41      Hook(&realNtCreateFile, MyNtCreateFile, ntdllmod, "NtCreateFile");
42      // ...
43      // ...other hooks...
44      // ...
45
46      notifyNewPid(0, GetCurrentProcessId());
47
48    break;
49
50    case DLL_PROCESS_DETACH:
51      OutputDebugString(_T("PROCESS DETACHED FROM DLL."));
52
53      // Removing Hooks
54      UnHook(&realNtCreateFile, MyNtCreateFile, "NtCreateFile");
55      // ...
56      // ... other hooks being removed here ...
57      // ...
58
59    break;
60  }
```

```
61
62   return TRUE;
63 }
```

Listing 6.4: Hooking DLL source code, DLL entry points

Listing 6.4 points out the entry point of the HookingDLL module. Windows Loader, at loading time, invokes the `DllMain` method and passes two important arguments: the module handle and an enumerative variable. The former, represents the handle of the loading process, which is not really relevant in our case. The latter argument can assume four distinct values, depending on the reason why the loader invokes the DllMain. Therefore, this variable is used as discriminant by a switch case (Line 12). Whenever the DLL is loaded for the first time into a process, the control will b passed to `DLL_PROCESS_ATTACH` branch. On the contrary, when the Windows Loader unloads the DLL from the process, the control is passed to the `DLL_PROCESS_DETACH` branch. Similarly, `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH` constants are used by the Windows Loader for loading and unloading the module to singular threads. In our case we expressly avoid this by disabling thread library calls.

At loading time, HookingDLL takes care of initializing some memory structures (Line 25) used afterwards, then it disables DllMain invocation upon thread events (Line 31). Afterwards API hooking is actualized, by invoking a custom function `Hook`, generated by a template, as shown in Listing 6.5. The `Hook` function is invoked for each function to be hooked. The complete list of API hooked in the current version of the HookingDLL is reported in Appendix B. Eventually, a call to `notifyNewPid()` communicates to the guest controller that the DLL has been correctly injected to a new process. By doing so, the guest controller adds the relative pid to the list of *monitored processes*, and starts listening for logging message from them.

```
1  template <typename Type>
2  bool Hook(Type* realFunction, void* hookingFunction, HMODULE module, const ↩
       char* function_name) {
3    char buff[512];
4
5    // Assign the pointer to the real function
6    (*realFunction) = (Type)(GetProcAddress(module, function_name));
7
8    DetourTransactionBegin();
9    DetourUpdateThread(GetCurrentThread());
10   DetourAttach(&(PVOID&)(*realFunction), hookingFunction);
11   if (DetourTransactionCommit() != NO_ERROR) {
12     sprintf_s(buff, "[CHOOKING DLL] %s not derouted correctly", ↩
           function_name);
13     OutputDebugStringA(buff);
14     return false;
15   }
16   else {
17     sprintf_s(buff, "[CHOOKING DLL] %s atteched OK", function_name);
18     OutputDebugStringA(buff);
19     return true;
20   }
21 }
22
23 template <typename Type>
24 bool UnHook(Type* realFunction, void* hookingFunction, const char* ↩
```

```
    function_name) {
25  char buff[512];
26
27  DetourTransactionBegin();
28  DetourUpdateThread(GetCurrentThread());
29  DetourDetach(&(PVOID&)(*realFunction), hookingFunction);
30  if (DetourTransactionCommit() != NO_ERROR) {
31    sprintf_s(buff, "[CHOOKING DLL] %s not detached correctly", ↩
        function_name);
32    OutputDebugStringA(buff);
33    return false;
34  }
35  else {
36    sprintf_s(buff, "[CHOOKING DLL] %s detached OK", function_name);
37    OutputDebugStringA(buff);
38    return true;
39  }
40 }
```

Listing 6.5: Hooking DLL source code, Hooking and UnHooking templates

When a process is going to be unloaded, the operating system takes care of invoking DllMain again, passing the `DLL_PROCESS_DETACH` value as reason argument. Thus, the control reaches the relative branch in the switch-case construct (Line 57), and the hooks are removed.

A closer look to the hooking mechanism is provided by Listing 6.5. In particular, we implemented two C++ templates in charge of providing Hooking and UnHooking functionality, independently of the functions to be hooked. The Hook templates starts by retrieving a pointer to the target function to be hooked, via `GetProcAddr` (Line 6). Afterwards, it starts a *DetourTransaction* and requests to hook the target function (*realFunction*) via a detour function (*hookingFunction*). Then, if the operation succeeds, the template returns a `true` value; in case of error, `false` is returned. The `UnHook` function works similarly. At first, it starts a transaction, then it requests to unhook the target function (this also destroys any allocated trampoline), finally a boolean value is returned for discriminating success or failure.

```
1 NTSTATUS WINAPI MyNtOpenFile(PHANDLE FileHandle, ACCESS_MASK DesiredAccess,↩
     POBJECT_ATTRIBUTES ObjectAttributes, PIO_STATUS_BLOCK IoStatusBlock, ↩
    ULONG ShareAccess, ULONG OpenOptions)
2 {
3   if (!shouldIntercept())
4     return realNtOpenFile(FileHandle, DesiredAccess, ObjectAttributes, ↩
        IoStatusBlock, ShareAccess, OpenOptions);
5
6   /* If the handle has write access, it is a good idea to calculate the ↩
       hash before the attached thread changes the file itself. To do so, we↩
        send a message to the GuestController everytime we see an NtOpenFile↩
        with write access. The Guest controller will then try to open the ↩
       file and store, in its report, the hash of the file before any ↩
       modification. After that, the Guest controller will answer to the ↩
       SendMessage and this thread will continue (yeah, SendMessage blocks ↩
       until the sender eats the message from the pump). */
7   string s = GetFullPathByObjectAttributes(ObjectAttributes);
8   if (IsRequestingWriteAccess(DesiredAccess))
9     NotifyFileAccess(s, WK_FILE_OPENED);
```

```
10
11   // Call first because we want to store the result to the call too.
12   NTSTATUS res = realNtOpenFile(FileHandle, DesiredAccess, ObjectAttributes↩
         , IoStatusBlock, ShareAccess, OpenOptions);
13
14   if (res == 0)
15     handleMap.Insert(*FileHandle, s);
16
17   #ifdef SYSCALL_LOG
18     // Use a node object to create the XML string: this will contain all ↩
           information about the SysCall
19     pugi::xml_document doc;pugi::xml_node element = doc.append_child(_T("↩
           NtOpenFile"));
20
21     // >>>>>>>>>>>>>>>> File Path <<<<<<<<<<<<<<<<
22     // The objectname contains a full path to the file
23     element.addAttribute(_T("Path"), s.c_str());
24
25     // >>>>>>>>>>>>>>>> DESIRED ACCESS <<<<<<<<<<<<<<<<
26     element.addAttribute(_T("AccessMask"), StandardAccessMaskToString(↩
           DesiredAccess).c_str());
27
28     // >>>>>>>>>>>>>>>> IO STATUS BLOCK <<<<<<<<<<<<<<<<
29     s.clear();
30     IoStatusToString(IoStatusBlock, &s);
31     element.addAttribute(_T("IoStatusBlock"), s.c_str());
32
33     // >>>>>>>>>>>>>>>> SHARE ACCESS <<<<<<<<<<<<<<<<
34     s.clear();
35     ShareAccessToString(ShareAccess, &s);
36     element.addAttribute(_T("ShareAccess"), s.c_str());
37
38     // >>>>>>>>>>>>>>>> OPEN OPTIONS <<<<<<<<<<<<<<<<
39     s.clear();
40     FileCreateOptionsToString(OpenOptions, &s);
41     element.addAttribute(_T("OpenOptions"), s.c_str());
42
43     // >>>>>>>>>>>>>>>> Result <<<<<<<<<<<<<<<<
44     s.clear();
45     NtStatusToString(res, &s);
46     element.addAttribute(_T("Result"), s.c_str());
47
48
49     if (NT_SUCCESS(res))
50     {
51     wchar_t buff[32];
52     wsprintf(buff, _T("0x%p"), *FileHandle);
53     element.addAttribute(_T("Handle"), buff);
54     }
55
56     log(&element);
57
58   #endif
59
60   return res;
61 }
```

Listing 6.6: HookingDLL source code, Detour function of NtOpenFile()

An example of detour function is provided by Listing 6.6. Function `MyNtOpenFile()` represents the detour procedure attached in place of `NtOpenFile()`. All the detour procedures share the same structure. At first, the function checks whether the following logic has to be applied or not. In fact, in case recursion is detected, detour function is skipped, and the original procedure is called directly, via the trampoline (Lines 3-4). This situation is pretty common when hooking low level API such as `NtOpenFile()` and using high level API to communicate with other processes. In fact, the notification mechanism, via named pipes, internally uses `NtOpenFile()` to achieve its goal. Thus, it is necessary to avoid hooking function calls used for logging and IPC with guest controller. For this reason, the HookingDLL module makes use of *Thread Local Storage* (TLS), a simple mechanism providing private memory to each thread. Whenever a thread invokes any of the logging procedures (such as `NotifyFileAccess()`), a thread-specific flag is raised. Then, when a recursion happens, shouldIntercept() checks whether the flag is set or not. In case it has been set, the detour procedure is skipped, and the trampoline to the target function is invoked directly (Line 4).

For logging purposes, each function may require an overhead to provide contextual data. For example, in this case, `NtOpenFile()` is invoked directly using a file handle, which does not bring any path information. Thus, the path associated to the file being accessed is obtained through a specific function (`GetFullPathByObjectAttributes()`), on Line 7. After that, the function checks if the access has to be logged or not. In case the file access is performed with write permission, the action is logged, therefore the HookingDLL synchronously notifies the guest controller. In this way, the guest controller has a chance of performing preliminary checks, such as calculating a hash of the accessed file before any other modification happens. Afterwards, the function calls the trampoline of the target function (in this case `NtOpenFile()` with original arguments). Eventually the HookingDLL takes track of all accessed handles and relative paths using an in-memory hashmap.

The last part of the function is executed only when the code is compiled with `SYSCALL_LOG` switch on. If that is the case, supplementary log information is sent to the guest controller, encoded in XML format. In this case, the logging information would include all the syscall arguments and the result of the execution.

6.2.5 Process hierarchy, dll injection and API hooking

So far we have described how HookingDLL is injected into a process and how hooking is performed via Microsoft Detours, using the injected DLL. Initially, the injector starts the target installer-process by pausing it and injecting the HookingDLL at startup. However, in case the process *forks* (i.e. creates a new process), the child process will not be subjected to hooking. Hence, it is necessary to recursively inject the HookingDLL to any child process, at spawn time.

Such an objective can be easily achieved by hooking the `CreateProcess()` API belonging to the Kernel32.dll library (user space). The idea is to intercept calls to process creation routines, spawn the child process with the suspended flag (using the trampoline), then inject the HookingDLL into the child; eventually we resume the execution normally. However, Windows offers different APIs to create processes [72]:

- `CreateProcessW()` (*kernel32.dll*)
- `CreateProcessAsUserW()` (*kernel32.dll*)
- `CreateProcessWithLogonW()` (*advapi32.dll*)

- `CreateProcessWithTokenW()` (*advapi32.dll*)

Those API belong to two relevant user-space libraries, documented by Microsoft. Unfortunately, their inner implementation details are not public nor documented. Therefore, the way they use lower level APIs, such as `NtCreateUserProcessEx()` is not completely disclosed and often subjected to change on any Windows version. For these reasons, we decided not to hook native APIs for process creation; instead we targeted a common internal routine invoked by all the previous functions: `CreateProcessInternalW()`. We discovered the existence of such an undocumented function by using a windows debugger, analyzing the call stack for simple custom test executables. In particular, we observed this function among the call stacks of `CreateProcessW()` and `CreateProcessAsUserW()`, as shown by Figure 6.5. The last two functions, instead, rely on a separate Windows Service, called *Secondary Logon*, which internally uses `CreateProcessAsUserW()`. Therefore, the reader may immediately note that every function has invoked, at a certain point, the `CreateProcessInternalW()` function.

```
0:000> uf /D /c kernel32!CreateProcessW
kernel32!CreateProcessW (757e204d)
  kernel32!CreateProcessW+0x27 (757e2074):
    call to kernel32!CreateProcessInternalW (7582de78)
0:000> uf /D /c kernel32!CreateProcessAsUserW
kernel32!CreateProcessAsUserW (758158af)
  kernel32!CreateProcessAsUserW+0x28 (758158d7):
    call to kernel32!CreateProcessInternalW (7582de78)
```

Figure 6.5: Call stacks of disassembled functions CreateProcessW() and CreateProcessAsUserW(), obtained via WinDbg

Listing 6.7 shows the detour function implemented on top of the `CreateProcessInternalW()`, within the HookingDLL.

```
1  BOOL WINAPI MyCreateProcessInternalW(HANDLE hToken,
2    LPCWSTR lpApplicationName,
3    LPWSTR lpCommandLine,
4    LPSECURITY_ATTRIBUTES lpProcessAttributes,
5    LPSECURITY_ATTRIBUTES lpThreadAttributes,
6    BOOL bInheritHandles,
7    DWORD dwCreationFlags,
8    LPVOID lpEnvironment,
9    LPCWSTR lpCurrentDirectory,
10   LPSTARTUPINFOW lpStartupInfo,
11   LPPROCESS_INFORMATION lpProcessInformation,
12   PHANDLE hNewToken)
13 {
14   if (!shouldIntercept())
15     return realCreateProcessInternalW(hToken, lpApplicationName, ↩
           lpCommandLine, lpProcessAttributes, lpThreadAttributes, ↩
           bInheritHandles, dwCreationFlags, lpEnvironment, lpCurrentDirectory↩
           , lpStartupInfo, lpProcessInformation, hNewToken);
16
17   CHAR   DllPath[MAX_PATH] = { 0 };
18   GetModuleFileNameA((HINSTANCE)&__ImageBase, DllPath, _countof(DllPath));
19   BOOL processCreated;
20
21   // Save the previous value of the creation flags and make sure we add the↩
         create suspended BIT
```

```
22  DWORD originalFlags = dwCreationFlags;
23  dwCreationFlags = dwCreationFlags | CREATE_SUSPENDED;
24
25  // Call the trampoline
26  processCreated = realCreateProcessInternalW(hToken, lpApplicationName, ↩
        lpCommandLine, lpProcessAttributes, lpThreadAttributes, ↩
        bInheritHandles, dwCreationFlags, lpEnvironment, lpCurrentDirectory, ↩
        lpStartupInfo, lpProcessInformation, hNewToken);
27
28  // Perform injection: we cannot use directly DetoursCreateProcessWithDll ↩
        due to the low level functionality being hooked.
29  if (processCreated) {
30    // Allocate enough memory on the new process
31    LPVOID baseAddress = (LPVOID)VirtualAllocEx(lpProcessInformation->↩
          hProcess, NULL, strlen(DllPath)+1, MEM_RESERVE | MEM_COMMIT, ↩
          PAGE_READWRITE);
32
33    // Copy the code to be injected
34    WriteProcessMemory(lpProcessInformation->hProcess, baseAddress, DllPath↩
          , strlen(DllPath), NULL);
35
36    OutputDebugStringA("HookingDLL: DLL copied into host process memory ↩
          space");
37
38    // Notify the HostController that a new process has been created
39    notifyNewPid(GetCurrentProcessId(), lpProcessInformation->dwProcessId);
40    kern32dllmod = GetModuleHandle(TEXT("kernel32.dll"));
41    HANDLE loadLibraryAddress = GetProcAddress(kern32dllmod, "LoadLibraryA"↩
          );
42    if (loadLibraryAddress == NULL)
43    {
44      OutputDebugStringW(_T("!!!!!LOADLIB IS NULL"));
45      //error
46      return 0;
47    }
48    else {
49      OutputDebugStringW(_T("LOAD LIB OK"));
50    }
51
52    // Create a remote thread into the target process and trigger the ↩
          LoadLIbraryA execution
53    HANDLE  threadHandle = CreateRemoteThread(lpProcessInformation->↩
          hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)loadLibraryAddress, ↩
          baseAddress, NULL, 0);
54    if (threadHandle == NULL) {
55      OutputDebugStringW(_T("REMTOE THREAD NOT OK"));
56    }
57    else {
58      OutputDebugStringW(_T("REMTOE OK"));
59    }
60
61    OutputDebugStringA("HookingDLL: Remote thread created");
62
63    // Check if the process was meant to be stopped. If not, resume it now
64    if ((originalFlags & CREATE_SUSPENDED) != CREATE_SUSPENDED) {
65      // need to start it right away
66      ResumeThread(lpProcessInformation->hThread);
67      OutputDebugStringA("HookingDLL: Thread resumed");
68    }
```

```
69    }
70
71    #ifdef SYSCALL_LOG
72    // Use a node object to create the XML string: this will contain all ↩
          information about the SysCall
73    pugi::xml_document doc; pugi::xml_node element = doc.append_child(_T("↩
          CreateProcessInternalW"));
74
75
76    // >>>>>>>>>>>>>>>> Result <<<<<< <<<<<<<<<<
77    string w = string();
78    NtStatusToString(processCreated, &w);
79    element.addAttribute(_T("Result"), w.c_str());
80
81
82    log(&element);
83
84    #endif
85
86    return processCreated;
87  }
```

Listing 6.7: Detour function for CreateProcessInternalW()

Similarly to the other detour functions, `MyCreateProcessInternalW()` begins with recursion checks. In case recursion is detected, the trampoline is invoked directly, without executing any other operation of the detoured function (Lines 14-15). If no recursion is detected, the `CREATE_SUSPEDED` flag is set, saving its previous value apart. Thus the trampoline to the original `CreateProcessInternalW()` is executed with that flag set (Line 26). Afterwards, if process creation succeeds, DLL injection is performed into the child process. In this case we could not use directly the `DetourCreateProcessWithDll()` utility: in fact that utility internally uses `CreateProcessInternalW()`, and that would cause recursion. Thus, we decided to manually implement DLL injection in this case. Therefore, we first allocate enough memory into the child process in order to store the path to the HookingDll to be loaded (Line 31). Then, we write the path into that memory space, via `WriteProcessMemory()`. Later on, we notify the guest controller about the new child process being spawned (Line 30) and finally allocate a remote thread into the child process, setting its entry point to *LoadLibraryA()* and specifying the dll path previously allocated as argument. Eventually we decide whether to start the main thread of the process directly or not, by looking at the original value of `CREATE_SUSPEDED` bit (Lines 64-70).

6.2.6 Hooking Windows services

Microsoft Windows implements several *services*, each one offering a specific set of functionality to other applications. Some of these services are generally used by installers. Particularly relevant in our context are two services: *Distributed Component Object Model* (DCOM) and *Microsoft Installer* (MSI).

The DCOM Service provides mechanisms to distribute applications' functionality over a network. It uses the Microsoft *Component Object Model* (COM) system, by implementing location transparency and relies on Remote Procedure Call (RPC) [71]. The DCOM service is critical for us due to its ability of spawning a process remotely. In fact, even though the application resides on the same machine, it may request to spawn a process via RPC, by relying on the local DCOM server.

The Microsoft Installer Service implements some background functionality related to MSI-compilant installers. Similarly to the DCOMLaunch service, also MSIExec uses RPC to communicate with client applications. Moreover, the MSIExec services implements a wider range of capabilities: from registry modification, file manipulation and process creation.

Any process in the system may, sooner or later, communicate with those services and delegate to them some operations. For instance, InstallShield may use DCOM in order to start some COM objects and deal with them later on. Another important issue regards the capability of a process to install a new service. In such a case the new service needs to be monitored as well. For this reason, we decided to apply the same hooking system to the following system services: *dcomlaunch*, *msiexec* and *services*. As a consequence, the injector takes care of checking whether those services are alive, and injects the HookingDLL into them before spawning the target installer. The relative code that addresses this task is exposed in Listing 6.8.

```
1
2 int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,LPSTR ↩
    lpCmdLine,int nCmdShow)
3 {
4   // Variables
5   [...]
6
7   // Arg check
8   [...]
9
10  // Some processes will use DCOMLAUNCHER. Patch it!
11  if (!HookAndInjectService(DLLPATH, DCOM_LAUNCH_SERVICE_NAME)) {
12    LogError("XXX INJECTOR ERROR XXX: Cannot hook DCOMLauncher");
13  }
14  else {
15    Log("[INJECTOR] DCOM Injection performed! :)");
16  }
17
18  // Some processes will use Windows Installer, so we need to path it
19  if (!HookAndInjectService(DLLPATH, WINDOWS_INSTALLER_SERVICE_NAME)) {
20    LogError("XXX INJECTOR ERROR XXX: Cannot hook WindowsInstaller service"↩
        );
21  }
22  else {
23    Log("[INJECTOR] WindowsInstaller Injection performed! :)");
24  }
25
26  // Also take care of patching services.exe
27  if (!HookAndInjectServicesExe(DLL_SERVICES_PATH)) {
28    LogError("XXX INJECTOR ERROR XXX: Cannot hook Services.exe");
29
30  } else {
31    Log("[INJECTOR] Services.exe Injection performed! :)");
32  }
33
34  // Rest of logic
35  [...]
36 }
37
38 BOOL HookAndInjectService(const char* dllPath, const char* serviceName){
```

```
39
40  // Declarations
41  [...]
42
43  // Look for the DcomLaunch process.
44  // Connect to service manager first, then open the service.
45  schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
46  if (schSCManager == NULL) {
47    DWORD err = GetLastError();
48    sprintf_s(strbuff, sizeof(strbuff), "XXXXXX Injector Error: Cannot open↩
          service manager error: %d", err);
49    LogError(strbuff);
50    return FALSE;
51  }
52
53  service = OpenService(
54    schSCManager,            // SCM database
55    serviceName,            // name of service
56    SERVICE_QUERY_STATUS);         // Query Status
57
58  if (service == NULL) {
59    DWORD err = GetLastError();
60    sprintf_s(strbuff, sizeof(strbuff), "XXXXXX Injector Error: Cannot open↩
          service, error: %d", err);
61    LogError(strbuff);
62    CloseServiceHandle(schSCManager);
63    return FALSE;
64  }
65
66  ZeroMemory(&srvStatus, sizeof(srvStatus));
67
68  // At this point we want to know which is the PID of the process running ↩
      this service.
69  if (!QueryServiceStatusEx(service, SC_STATUS_PROCESS_INFO, (LPBYTE)&↩
      srvStatus, sizeof(SERVICE_STATUS_PROCESS), &dwBytesNeeded)){
70    DWORD err = GetLastError();
71    sprintf_s(strbuff, sizeof(strbuff), "XXXXXX Injector Error: Cannot ↩
        retrieve status info about service %s, error: %d",serviceName, err)↩
        ;
72    LogError(strbuff);
73    CloseServiceHandle(schSCManager);
74    return FALSE;
75  }
76
77  // Check if the service is started. If not, start it now.
78  if (srvStatus.dwCurrentState != SERVICE_RUNNING) {
79    sprintf_s(strbuff, sizeof(strbuff), "Service %s was not running. I'll ↩
        start it now.", serviceName);
80    Log(strbuff);
81
82    StartSampleService(schSCManager, serviceName, &dwProcessId);
83  }
84  else {
85    // We finally have the pid now. Copy into a local variable.
86    dwProcessId = srvStatus.dwProcessId;
87  }
88
89  // We don't need this anymore.
90  CloseServiceHandle(schSCManager);
```

```
91
92    sprintf_s(strbuff, sizeof(strbuff), "Process ID of %s is %d", serviceName↩
          , dwProcessId);
93    Log(strbuff);
94
95    // Get the handle of the running dcomlauncher process
96    return injectIntoPID(dwProcessId, dllPath);
97  }
98
99  BOOL HookAndInjectServicesExe(const char* dllPath) {
100
101   HANDLE hProcess = NULL;
102   DWORD dwProcessId;
103   char strbuff[512];
104
105   // Check the pid of services.exe and proceed with the injection
106   PROCESSENTRY32 entry;
107   entry.dwSize = sizeof(PROCESSENTRY32);
108
109   HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
110
111   if (Process32First(snapshot, &entry) == TRUE)
112   {
113     while (Process32Next(snapshot, &entry) == TRUE)
114     {
115       if (_stricmp(entry.szExeFile, "services.exe") == 0)
116       {
117         hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, entry.↩
              th32ProcessID);
118         dwProcessId = entry.th32ProcessID;
119         sprintf_s(strbuff, sizeof(strbuff), "Process ID of services.exe is ↩
              %d", dwProcessId);
120         Log(strbuff);
121         CloseHandle(hProcess);
122         break;
123       }
124     }
125   }
126
127   CloseHandle(snapshot);
128
129   // Get the handle of the running dcomlauncher process
130   return injectIntoPID(dwProcessId, dllPath);
131 }
```

Listing 6.8: Injector's source code: injecting HookingDLL into dcom and msiexec

The source code is self explanatory. At the very beginning of its execution, the injector immediately injects the HookingDLL into the relevant services. A specific function takes care of locating the service, retrieving its PID and injecting the given DLL into it: `HookAndInjectService()` (Line 38). Specific injection logic into the process is added by another function: `injectIntoPID()`. The injection logic is the same we applied for `CreateProcessInternalW()`. The only difference between the two approaches regards the call to `CreateRemoteThread`. In fact, starting from Windows Vista, Microsoft protects system services against the usage of `CreateRemoteThread`. In order to workaround this limitation, we relied on another function, called `RtlCreateUserThread`, which does not perform this security check and allows us to create a remote thread into a system service.

6.3 Sniffer

Our specific implementation of the gateway sniffer is completely software, based on a Linux Ubuntu 14.10 operating system (64 bit, Kernel 3.19), equipped with a specific set of networking services. Linux offers a wide set of networking capabilities, some of them provided directly by its kernel module [68]. Moreover, various open source software packages have been developed over the years, targeting specific networking functions. Although performances are not comparable to high-end proprietary hardware accelerated solutions, they remain more than acceptable for our scope.

An overview of the general architecture of the sniffer is provided in Section 5.2.6. The complete list of services to be implemented by the network sniffer are the following:

- Networking functions
  - IP Routing
  - DHCP
  - DNS
  - NAT
  - Traffic sniffing
- Host Controller interaction
  - Handling sniffing sessions
  - Serving captured files
- Capture file synthesis

The first set of services represents the basic networking functions that guest machines need to operate on the network. As we will discuss later in this chapter, those services are provided by a combination of open-source Linux software and native Linux kernel modules. The second subset of services addresses the problem of providing a network interface for the host controller, which will command the sniffer. As we will see, those services have been implemented via a simple python daemon, offering convenient HTTP APIs accessible to the Host Controller. Finally, the last service provided by the sniffer gateway aims at elaborating capture files, in order to summarize relevant data out of redundant and large capture logs.

6.3.1 Networking services

Table 6.3.1 describes which software packages have been adopted in order to implement relative networking features.

The most basic network service for a router is the ip forwarding service. Linux implements such a feature directly within its kernel. However packet forwarding among different NICs is disabled by default, but can be enabled by simply setting the `net.ipv4.ip_forward` value to 1 in `/proc/sys/net/ipv4/ip_forward`. By doing so, the router will automatically forward incoming IP packets according to its routing table, passing them to the default gateway if no matching is obtained among the connected networks.

The second most important networking service in our configuration is represented by DHCP. The sniffer gateway uses a specific software package to implement such a feature:

Network Service	Software Package
IP Routing	Linux kernel
DHCP	DNSMASQ 2.68
DNS	DNSMASQ 2.68
NAT	Iptables (Linux kernel)
Firewall	Iptable (Linux kernel)
Network sniffing	Tcpdump (Libpcap)

Table 6.1: Sniffer Gateway: Mapping between software relative networking functions provided

dnsmasq. Dnsmasq is a lightweight software component, providing both dhcp and dns services, targeting small network infrastructures. Dnsmasq is configured through a text file, usually located in `/etc/dnsmasq.conf`. Its configuration is really simple and requires just a few modification to the standard configuration file provided by default. Listing 6.9 shows the necessary configuration entries needed for dnsmasq in order to work correctly in our infrastructure.

```
# If you want dnsmasq to listen for DHCP and DNS requests only on
# specified interfaces (and the loopback) give the name of the
# interface (eg eth0) here.
# Repeat the line for more than one interface.
interface=eth1

# Uncomment this to enable the integrated DHCP server, you need
# to supply the range of addresses available for lease and optionally
# a lease time. If you have more than one network, you will need to
# repeat this for each network on which you want to supply DHCP
# service.
dhcp-range=eth1,192.168.0.2,192.168.0.150,12h
```

Listing 6.9: Configuration entries needed by dnsmasq to operate as transparent DNS proxy and simple DHCP server

In summary, the configuration file tells to dnsmasq daemon to listen on interface *eth0*, by answering both to DNS and DHCP requests. In particular, DHCP service is configured to lease addresses from 192.168.0.2 up to 192.168.0.150, with a lease period of 12 hours. The rest of the settings are taken by defaults and are perfectly aligned to our needs. Specifically, DNS service is exposed locally to the network, and requests are proxied though the system's DNS configuration. Therefore dnsmasq acts exactly like a transparent DNS proxy by default.

Another important service needed by the guests is NAT. Network address translation enables multiple connections though a single public IP, shared among many machines, each of them using a private IP. Due to the possible high amount of guest machines running, natting enables Internet access through a single public IP, rather than requiring many dedicated public IPs. On Linux, NAT is implemented directly within the kernel and can be configured via the iptables module. The configuration applied in our case is shown in Listing 6.10.

```
# Basic iptables configuration for natting
# eth0 => Internet
# eth1 => Local LAN 192.168.0.0/24

# Masquerade private source lan addresses with external IP when directed to↩
    eth0
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

# Accept packets incoming from public network and directed to local nodes ↩
    only if connections status is ACCEPTED
iptables -A FORWARD -i eth0 -o eth1 -m conntrack --ctstate ESTABLISHED,↩
    RELATED -j ACCEPT

# Allow all outgoing connections from local LAN to external network
iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

Listing 6.10: Configuration commands issued to iptables in order to enable NAT

The last network service offered by the sniffer gateway is traffic sniffing. The tool used for this purpose is *tcpdump*[3], an open source protocol analyzer [68]. This tool is based on the *libpcap* C/C++ library, offering network traffic capture capabilities on many platforms, Linux included. Tcpdump is capable of sniffing network traffic from layer 2 in the TCP/IP stack. Before starting a new job on a guest machine, the host controller requests to start a new sniffing session to the associated sniffer gateway. As a consequence, a new tcpdump process is spawned, with specific arguments aimed at filtering only the traffic relevant for that capture session.

```
1 tcpdump -w <capturefile> -C 2048 -W 1 -i eth1 ip and ether host <mac>
```

Listing 6.11: Linux command to start a capture session with specific paramters

Listing 6.11 shows how capture sessions are spawned by the sniffer gateway in response to host controller requests. The reader may notice several arguments being specified. First of all, the *-w* switch tells to tcpdump where to store the captured data on the disk. The following arguments, *-C 2048*, limits the capture file size to 2 Gb, avoiding to fill up the gateway disk in case of network angry jobs. The *-W 1* argument limits the number of captured files to 1, so that the total amount of captured data is 1 * 2048 Mb. After that, the *-i eth1* argument instructs tcpdump to sniff traffic coming from the eth1 interface, matching the following filters *ip and ether host <addr>*. Those filters have two effects: to hold only packets going from/to the relative guest (having <addr> as mac address), and to filter information lower than network level.

6.3.2 Host controller interaction

The host controller interacts with the gateway sniffer thanks to a web application, implemented by a custom Python module, spawned on the gateway at boot time. This component is called *MiddleRouter*, and is in charge of:

- Offering web APIs for controlling sniffing sessions
- Implementing a simple web-ui for manually checking spawning sessions
- Exposing a network service for synthesizing captured data

The web service has been implemented using the *Flask*[4] Python library. Table 6.2 points out supported methods by the web service.

When performing a GET request to the root of the gateway sniffer, a simple html page is returned, which dynamically updates itself. This page offers an handy way for manually controlling the sniffing status of the system. A screen shot of the returned web page is offered by Figure 6.6.

[3] http://www.tcpdump.org/
[4] http://flask.pocoo.org/

Url	Method	Parameters	Description
/sniffers	Get	None	List all the sniffer instances available.
/sniffers	Post	{'mac':'...'}	Creates a new sniffer instance, aimed at sniffing network data from/to the specified mac address.
/sniffers/*MAC*	Get	None	Returns details about the sniffer attached to the mac specified in the url
/sniffers/*MAC*	Delete	None	Deletes the specified sniffer, if stopped
/manager/*MAC*/start	Get	None	Starts the sniffing instance associated to
/manager/*MAC*/stop	Get	None	Stops the sniffing instance associated to
/manager/*MAC*/collect	Get	None	Download the captured data file for instance
/manager/*MAC*/delete_log	Get	None	Removes the captured data file for instance

Table 6.2: MiddleRouter http API specification

6.3.3 Capture file synthesis

The gateway sniffer captures raw network data from MAC level (OSI 2), collecting a very large amount of information for each session. Maintaining such that amount of data for each analysis is possible, but would require very large storage systems. Moreover, a relevant part of the raw data exchanged at level 2 and level 3 corresponds to overhead. Classical examples are the header checksums calculated at mac layer as well as packet re-transmissions at network layer, and so on. Moreover, storing all the capture data in raw format does not provide any facilitation for querying and correlating captured data. For this reason it is necessary to post-process the capture files for each job analysis, reducing the amount of data to be stored in the system.

The MiddleRouter implements a network service aimed at elaborating network-capture data files, called *synthetizer*, by extracting relevant information out of capture files. This synthesis process takes as input the raw capture file, obtained previously by a sniffing session, and returns a report in json format, containing the following data:

- List of hosts and relative packet statistics
- List of tcp network flows and relative statistics
- List of udp network flows and relative statistics
- List of observed network protocols
- List of http requests
- List of http downloads

The first five items of the list are calculated via *tshark*[5], a protocol analyzer utility able of parsing and processing raw tcpdump outputs. Regarding contacted hosts, tshark extracts hostnames and associated ip(s). Concerning udp and tcp network flows, tshark calculates the number of trasmitted/received frames and bytes for each endpoint (source/destination ips, ports). Tshark also extracts the list of observed protocols appearing in the capture

[5]https://www.wireshark.org/docs/man-pages/tshark.html

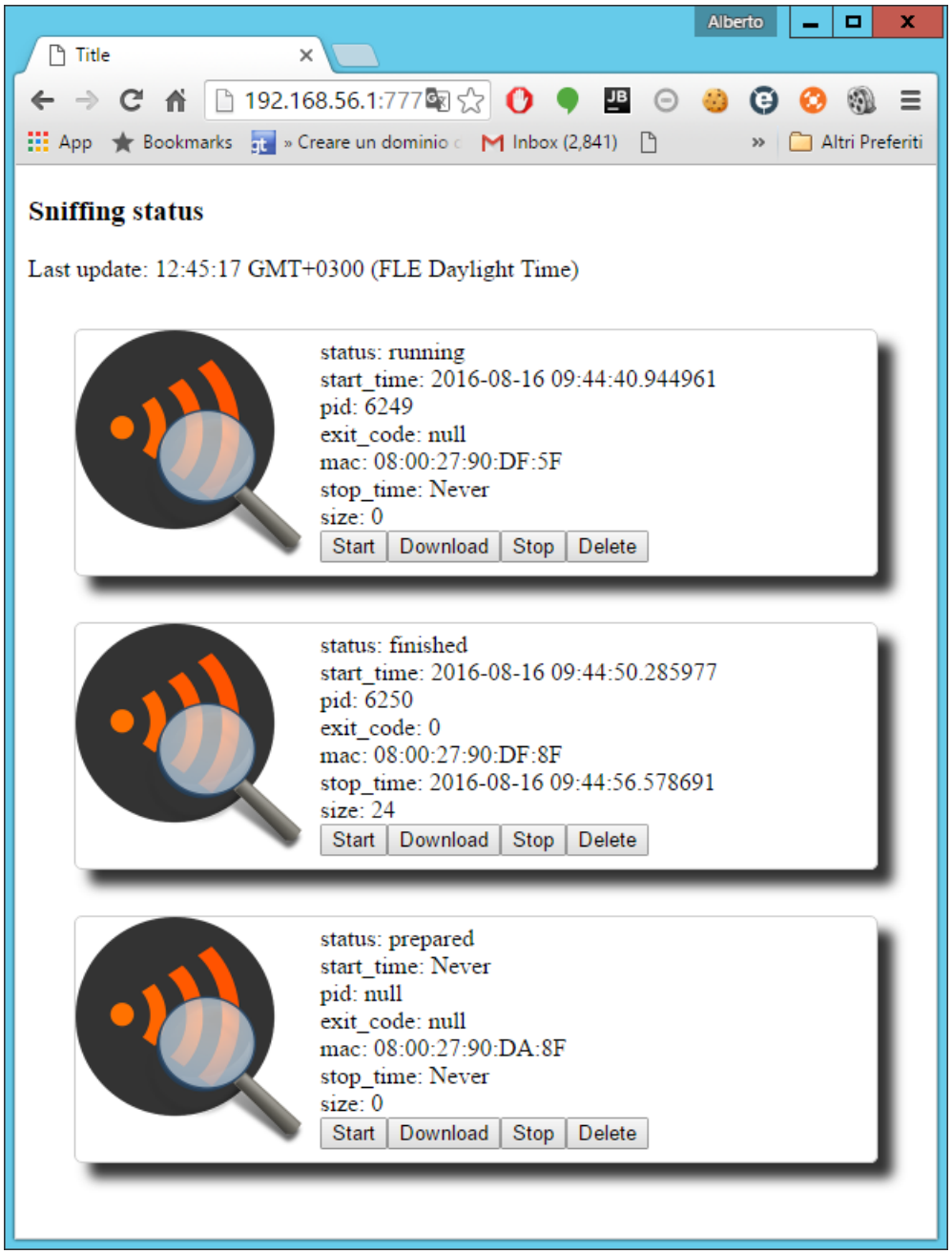


Figure 6.6: Screenshot of the sniffer gateway monitoring page implemented by MiddleRouter

file, alongside with the number of exchanged frames and bytes. For instance, if a process uses some P2P protocol as bittorrent, this would result in a specific entry in the observed protocols array. Eventually, tshark is also able of calculating http requests statistics: for each destination host it collects contacted url paths.

Unfortunately, tshark is unable to extract downloaded files out of a capture file. To do so, other tools are required. In particular we need to perform the following steps in order to obtain http downloads:

- Reassemble tcp streams from L2/L3

- Interpret HTTP protocol
- Analyze HTTP responses and extract binary files
- Recursively analyze downloaded compressed files

In order to reassemble the tcp streams and perform simple http processing, the synthetizer relies on *tcpflow*[6]. This utility is able of extracting all the TCP flows by parsing raw capture files. As a result, tcpflow provides a bounch of files, each one corresponding to a particular tcp flow. By specifying the *-e http* parameter, tcpflow also performs basic http processing, providing more metadata for every flow. Each flow is described by its file name, which is formatted as follows: *<SOURCE ADDRESS>.<SOURCE PORT>-<DESTINATION ADDRESS>.<DESTINATION PORT>-<PROTOCOL>*. For example, an http response obtained by server 8.8.8.8:80 for client 192.168.1.2:54635 would produce the file *008.008.008.008:00080-192.168.001.002:54635-HTTPBODY*. Once extracted http body responses, the synthetizer opens each file and analyzes its binary contents, guessing the mime type and calculating md5/sha1 hashes. If the mime type corresponds to a compressed file, the synthetizer extracts its contents and recursively applies the same analysis to extracted files. Eventually, the synthetizer builds up the downloads section of the analysis. As a consequence, this section will contain a set of entries, one for each downloaded file, describing the source ip, source port, the mime type, the file size and calculated hashes.

[6] http://manpages.ubuntu.com/manpages/xenial/man1/tcpflow.1.html

Chapter 7

Test and evaluation of results

In this chapter we present the results obtained by experimenting our PUP analysis infrastructure. In particular, the chapter begins with a detailed description of the test. We first justify how we performed job collection, then we comment the composition of input binaries, in terms of binary types, sizes and sources. After that, we describe the hardware configuration used for hosting the analysis system and to run the tests. Later on, results are presented to the reader. In that context, we evaluate how good was the UI interaction mechanism, by manually double checking how many interactions led to successful installations. Later on, focus is given to performance and scalability capabilities of the system. Finally we introduce a series of heuristics that could be used in order to detect PUPs and discuss their goodness.

7.1 Test configuration

Our analysis system necessitates to be tuned before being operative. In fact, each node of the system requires specific settings. For instance, crawlers need URLs where to search for binary installers, while host controllers necessitate of hypervisor-dependent configurations. Moreover, in case virtual machines are used, each VM needs to be configured accordingly to the resource available on the hosting system. *Job selection* and *resource allocation policies* represent crucial aspects of the system, which are discussed in the next paragraphs.

7.1.1 Job selection

In order to run the analysis and test the performance of our prototype, the database needs to be filled with some inputs. Inputs are provided by crawlers, which scan arbitrary websites, looking for executable binaries to analyze. The current version of the prototype is equipped with three different crawlers, each one targeting a specific software distributor website. Targets for our test are reported in Table 7.1. The underlying criterion we applied in the selection of targets is *popularity*. Thus, chosen websites represent the top-visited freeware aggregators at the time of writing, in accordance with Alexa's global ranking[1] (updated in September 2016). We also decided to avoid explicit PPI distributors, as they would possibly bias the analysis with a very high number of PUPs.

We noticed that all of those websites offer internal freeware rankings, according to the most downloaded applications. Therefore, our crawlers target those rankings and download

[1] http://www.alexa.com

	cnet.com	**softonic.com**	**filehippo.com**	**total**
Alexa's Global Rank	160	297	705	/
# .exe binaries	196	195	81	**472**
# .msi jobs	4	5	3	**12**
# collected jobs	200	200	84	**484**

Table 7.1: Jobs' sources with relative Alexa's global rank, divided by format. Crawling session of 17th September 2016

most popular software. Some websites also offered advanced filtering mechanisms, such type of license (freeware, shareware, etc), supported operating system (Windows Xp, Windows 7, etc) and system architecture (32 bit or 64 bit). In those cases, rankings have been filtered by Windows 7 supported OS, freeware license and 32 bit architectures.

All the crawlers operate in a similar way: they scan the list of most popular downloads, sorted by popularity, and accept the job only if the download file is either a 32bit executable or a Windows Installer archive. Then, the binary file is accepted only if its cryptographic hash and source domain is not already present into the database. Moreover, crawlers inspect the mime type of source page, the magic number and the extension of the downloaded file, only accepting 32bit Windows compatible executables or MSI.

In order to limit the amount of files to be collected, each crawler has been configured to download up to 200 jobs. Therefore, the test has been limited to the first 200 top ranked executable files for each websites, for a maximum of 600 jobs. However, by applying these criteria, crawlers were able to collect 484 different installers in total. Table 7.1 shows that the FileHippo only offered 84 valid jobs. The reason why this happened depends on the limitation of its ranking system: FileHippo only shows its 100 hottest downloads, and 16 of them are not 32 bit executables nor MSI files.

The reader may observe that the majority of collected binary files is in .exe format, while just a little subset of them uses the windows installer format. This consideration represents valuable information: sadly, the majority of the top-downloaded utilities freely available on the web do not comply with the msi file format. Indeed, the total number of collected msi file is as low as 12, which represents barely 2.48% of the input jobs.

It is also interesting to consider how many executable files are shared among those sources. For this reason, we identified the common binaries by grouping jobs with same binary hashes and different sources. As a consequence, Table 7.2 was produced.

	cnet.com		**filehippo.com**		**softonic.com**	
	#	%	#	%	#	%
cnet.com	200	100	13	15.47	10	5
filehippo.com	13	6.5	84	100	2	1
softonic.com	10	5	2	1	200	100

Table 7.2: Percentage of identical binaries crawled across different sources

Interestingly, Filehippo and Cnet share a noticeable number of binary files. In particular Filehippo shares 13 binaries over 84 with Cnet. Similarly, Softonic shares 10 binary files with Cnet and only 2 with filehippo.

	Unit 1	Unit 2
Brand	HP	HP
Model	ProLiantBL280c G6	ProLiant BL280c G6
CPUs	2 x Intel Xeon E5640	2x Intel Xeon E5640
Memory	64 Gb DDR3 ECC	64 Gb DDR3 ECC
Storage	136 Gb Raid 1 on Smart Array P512m Samsung EVO 850 500Gb SSD	136 Gb Raid 1 on Smart Array P512m
Network	2 x Intel 82576 Dual Gigabit	2x Intel 82576 Dual Gigabit
Operating System	Windows Server 2012 R2 (64 bit)	Ubuntu Server 14.10 (64bit)

Table 7.3: Hardware specifications for testing infrastructure

7.1.2 Infrastructure configuration

Our PUP analysis system has been deployed in a private production environment offered by the Aalto university. We were entitled a couple of rack-mount enterprise-grade servers, i.e. two *HP ProLiant BL280c G6*[2]. The given hardware does not rely on latest CPU architectures, yet it offers an acceptable starting point for our needs. Table 7.3 reports the specific hardware configuration for both of the rack units.

The reader may notice there are little differences between the two hardware configurations. Both the units are based on a couple of Intel Xeon E5640 processors, each one counting 4 physical cores running at 2.66 GHz. Thanks to the Intel Hyper Threading (HT) technology, each processor runs 8 threads. This means each Unit is theoretically capable of running 16 threads nearly simultaneously. Each unit also has 64 Gb of total memory, running at 1333 MHz. Regarding the storage options, both the units rely on hardware raid controllers, each one providing 136 Gb of total redundant storage (RAID 1). Unit 1 also has a dedicated 500 Gb consumer level solid state drive. Networking capabilities are identical for both of the units: each one is connected to a gigabit public network, with external Internet access and protected by a firewall. The second and last difference between the two units regards the installed operating system. We expressly asked to be given two heterogeneous systems in order to prove platform-independence of our analysis system. As a consequence, the first unit has been equipped with Windows Server 2012 R2 (64bit), while the other unit runs a plain Ubuntu Server 14.10 (64bit) operating system.

Given the modular and distributed nature of the analysis system, it is necessary to decide how to deploy the various nodes among the two units. In particular, a very basic configuration of the analysis system requires:

- One central database
- One or more host controllers
- One or more guest machines

Figure 7.1 shows how the distributed nodes have been deployed between the two server units.

In order to balance the IO on the disks, we decided to deploy the central database on Unit 1. In fact, this unit has more storage space to be used (combining the SSD and the

[2]http://h20564.www2.hpe.com/hpsc/doc/public/display?docId=emr_na-c01740643

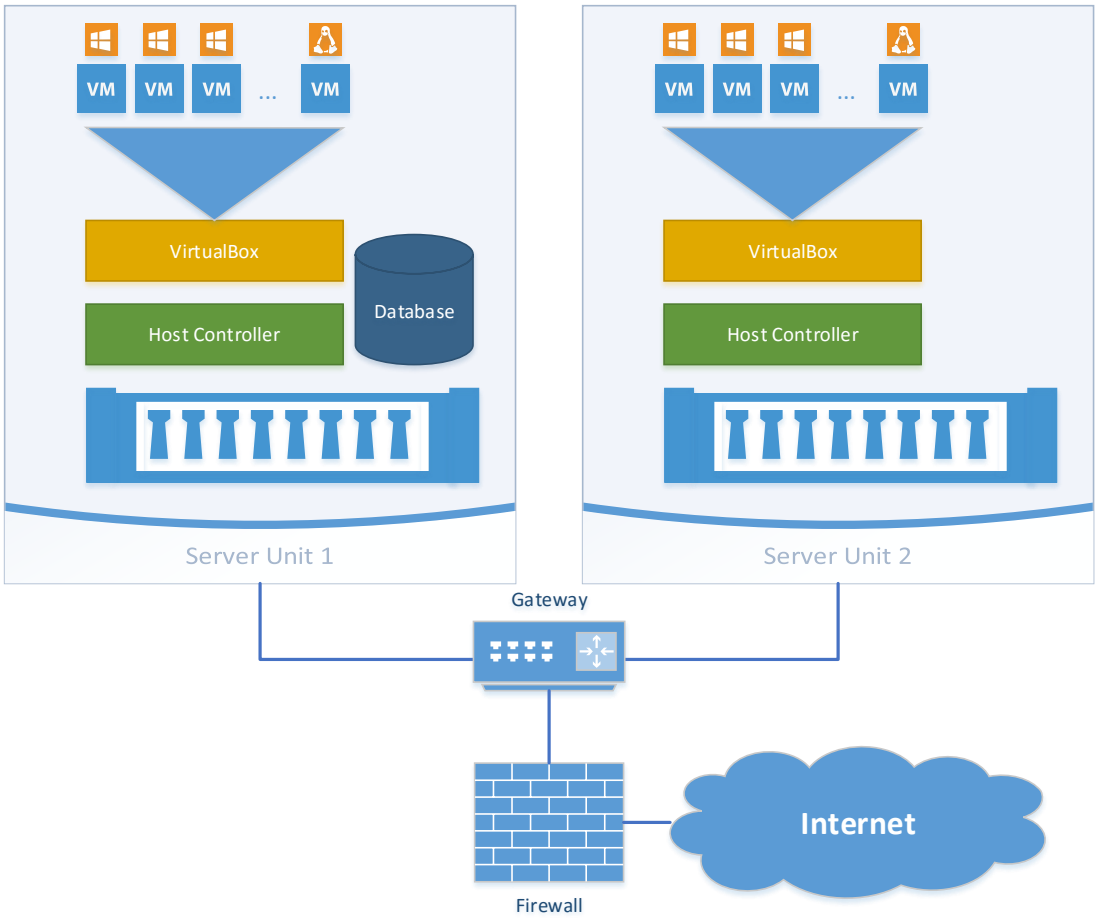


Figure 7.1: Deployment of distributed nodes among the server units

RAID 1 logic array), therefore it makes sense to install the database on this unit. More specifically, the database resides on the RAID 1 logic array. The reason why we decided in favor of such an option mainly regards reliability: RAID 1 offers good protection grade in case of disk failure. Moreover, the database is not stressed much during PUP analysis: host controllers just requests jobs and save results at the beginning and end of each analysis. Therefore, there would be no advantage in deploying the solid state drive.

Both the servers units have sufficient computational power to serve as host controllers. In fact, both of them support hardware accelerated virtualization (thanks to Intel VT-d technology) and have enough CPU cores and memory to ensure sufficient performance for a discrete number of small virtual machines. Thus, we decided to install a dedicated host controller instance on each server unit.

Guests are virtualized though VirtualBox, intsalled on both the server units. Each host controller has been configured to use the virtualbox driver for handling machine states.

Two network sniffers have been deployed, one per server unit, virtualized alongside the guests. Therefore, each sniffer handles the traffic generated by guest machines handled by the same host controller. This solution is particularly convenient because allows to completely virtualize the network infrastructure among guests and sniffers. Moreover the traffic is routed locally to the virtualization engine (in this case VirtualBox) without

actually passing over wires: this reduces latency and ensure high bandwidth.

Both the units communicate with the external Internet via a gateway and a firewall, protecting the system from unwanted inbound connections, yet allowing outbound connections. The host controller deployed on the second unit also uses the same network in order to communicate with the central database. On the other hand, communication in between host controller and database happens locally for server unit 1.

The reader may notice that Unit 1 can run the analysis on its own. In fact it hosts the db, a host controller and a series of guests. Therefore, server unit 2 is not crucial for the system. That is correct. In fact, this configuration allows us to switch on and off a second node, whenever high load is detected. However, no automatic system has been developed for turning on or off the secondary unit, yet.

Specific server units configurations

Each server unit has a limited amount of resources to be used. In particular, those resources are:

- Cpu cores / threads
- Primary memory
- Secondary memory (storage)

In order to ensure good performances, it is crucial to balance correctly those resources among the software modules, ensuring no bottleneck is present. Even though both the server units are similar, their configurations are different. This difference is due to the asymmetric distribution of the database and the unavailability of a solid state drive in server unit 2. Infact, as it generally happens in modern systems, mechanical disks represent the main performance bottleneck [94]. On the contrary, solid state drives are now quickly filling the gap, fixing that bottleneck. For these reasons, we will discuss allocation of resource separately, pointing out the policies we applied in order to balance the system.

CPU Cores. Each server unit is capable of running up to 16 threads simultaneously. Both the HostController and the database are not meant to perform high CPU-bound operations; instead they mainly require IO resources. Thus they only require little CPU time to work correctly. On the contrary, guests are more cpu-hungry. This happens because of the large amount of hashing and image processing performed by the guest controllers, happening in parallel with the installations. As a consequence, we decided to assign all the CPU cores to the virtualized system. In particular 2 virtual cpu cores (vCPU units) have been assigned to each VM and 4 vCPU units have been assigned to the sniffer gateway. Considering that server unit 1 is running 8 VMs and a gateway sniffer, the total number of vCPU requested would be 20. As a consequence, the cpu power results under-dimensioned by a 25% factor: 16 threads are available but 20 would be required. However, the usage of such a dimensioned system increases the CPU usage efficiency and still does not represent a bottleneck for the entire system.

The second unit has been dimensioned differently. As we will discuss later, storaging and IO capabilities impact heavily on the system. So, the number of running VMs has been reduced to 6. Therefore the second server uses 12 vCPU for all the virtual machines, plus a sniffer gateway, counting 4 more vCPU. As a consequence, the second unit uses 16 threads for 16 vCPU, with a 1:1 dimensioning approach.

Ram Memory. Ram memory is a crucial resource for host controllers. Memory allocation has is shown in Figure 7.2.

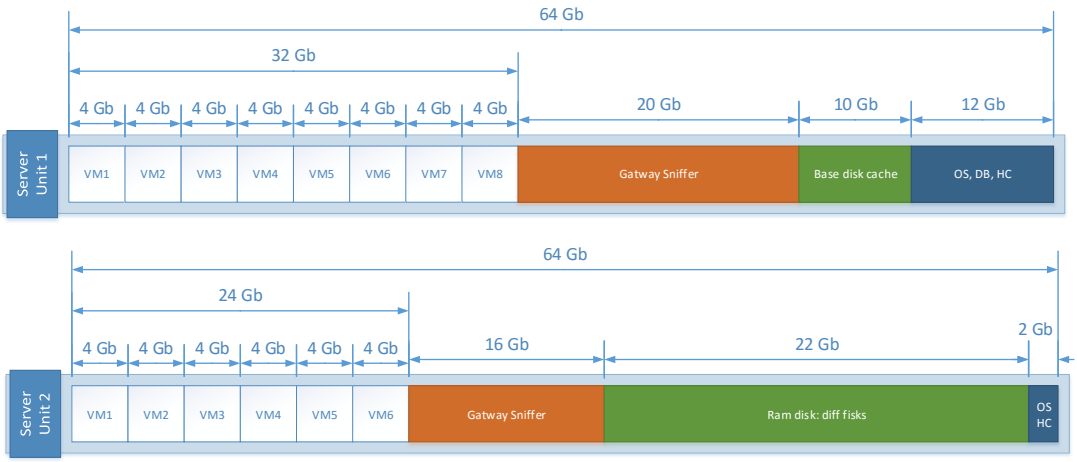


Figure 7.2: Memory allocation comparison between server unit 1 and 2

The virtualization system absorbs the majority of the available memory. Server unit 1 allocates half of the total memory for VMs: each vm is assigned 4 Gb of dedicated memory, for a total of 32 Gb. Similarly, Unit 2 uses 24 Gb of memory for virtual machines.

The gateway sniffer is assigned a quantity of memory calculated in relation with the maximum number of concurrent sniffing session. Indeed, gateway sniffers allocate an internal ramdisk of 2 Gb for possible concurrent sniffing sessions, plus 4 Gb of memory for virtualized operating system. As a consequence, unit 1 allocates 20 Gb for the sniffer, while unit 2 allocates 16 Gb.

A noticeable difference between memory allocation of unit 1 and unit 2 regards the green and blue areas, as shown in Figure 7.2. Unit 1 reserves 10 Gb of memory used as ramdisk cache for the base virtual disk shared among its virtual machines. In this case it amounts to 10 Gb. At the same time, differential disks are stored directly on the solid state drive of the unit. The same approach is inapplicable for the second unit, hence a portion of the 22 Gb is reserved to be used as ramdisk for differential disks.

Finally, unit 1 reserves 12 Gb of memory for host operating system, database and host controller daemon. On the contrary, the second unit only reserves 2 Gb for the operating system and host controller daemon. This difference is mainly due to the different operating system memory requirements (Ubuntu 14.10 is less memory-hungry than Windows Server 2012) and to the memory requirements of the database.

Storage. Both the server units have limited storaging capabilities. While Unit 1 can count on a 136 Gb volume and a 500 Gb SSD, the second unit only has a 136 Gb volume.

For each job analysis, a discrete amount of data is generated: network sniffed data can grow up to 2 Gb per job, while the IO log might hit tens of megabytes. Therefore, analyzing up to 484 jobs can led up to 1 Tb of temporary capture files. Such an amount of data cannot be held by neither of the server units. For this reason, temporary logs and capture files are stored on an external NAS, offering 2 Tb of space. The NAS only serves as *temporary buffer* for raw analysis data. Each time a job analysis is done, the

sniffer performs the synthesis of the sniffed data, extracting only a few Megabytes of data. However, due to debugging reasons, the current implementation of the system does not delete temporary capture data, which is maintained for further analysis on the NAS.

Another important aspect to consider is the IO capabilities of the storaging system. While unit 1 can count on a recent consumer-grade solid state drive, Unit 2 only relies on mechanical disks, which are several times slower than solid state drives. In particular disk IO capabilities heavily affects performances of the system. Each job aims at deploying software on the machine, therefore it is reasonable to assume that those process are *IO-bound*. Hence, neither Unit 1 nor Unit 2 use the mechanical disk for storing guest machine images. In fact, server unit 1 relies on the SSD, while server unit 2 uses a ramdisk for storing the differential guest images.

7.2 Evaluation of results

Once the infrastructure was ready, and crawlers finished the input collection process, the real automated analysis started. At first, we ran the entire analysis process just using a single server unit, i.e. server unit 1, running 8 VMs. Afterwards, the same test has been executed with both the server units, with a total of 14 VMs.

Measure	**Single server results**	**Dual server results**	**Improvement**
# Input Jobs	484	484	N/A
# Guests	8	14	75 %
Total Test Duration (hh:mm:ss)	19:24:20	11:13:42	72.83 %
Average Throughput (Jobs/Minute)	0.42	0.72	72.83 %
Average Job Analysis Duration (hh:mm:ss)	00:13:20	00:12:11	9.44 %

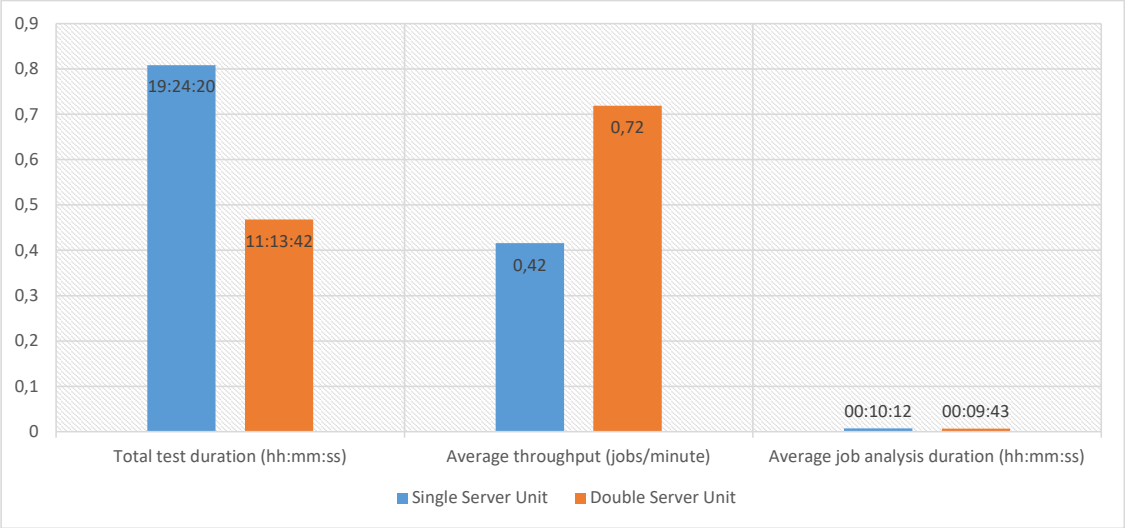
Table 7.4: Results of tests: test duration, throughput and average analysis duration

First column of Table 7.4 summarizes results about the first test. A single server offering 8 VMs is able to analyze the total input set in roughly 19.5 hours. The resulting throughput of the analysis system is 0.42 jobs per minute, with an average analysis time of about 13 minutes. The same results are plotted on Figure 7.2 for visual comparison.

When taking advantage of both server units, performance globally grows. In particular the total analysis time drops to 11 hours 13 minutes 42 seconds, while the throughput of the systems raises up to 0.72 jobs per minute, which means we obtain a 72.83% improvement by adding 6 guests to the analysis system. This result clearly demonstrates scalability capabilities of the system. On the other hand, the average time for a single analysis is reduced by a 10% factor. This improvement is justified by the better performance obtained by guests belonging to the second server. In fact, they rely on ramdisk instead of a solid state disk. Beside the second server unit dedicates a single CPU core to each vCPU, while the first server unit is underdimensioned from CPU perspective.

It is worth to noticing that average analysis time mainly depends on the hardware configuration of guests. The more resources are dedicated to each guest (memory, fast disks or ramdisk, cpu cores), the faster will be the analysis. Memory and disks are the most stressed devices, due to the IO-bound nature of analyzed jobs. At the same time,

Figure 7.3: Results of tests: comparison of single server and dual server configurations



hash calculations and image recognition absorb cpu power as well. However, over a certain threshold, guest's hardware capabilities eventually hit a bottleneck: network bandwidth. In fact, most of the analyzed jobs exchanged data with the network. More specifically, the average of downstram tcp traffic per installer is about 103 Mb, with a minimum of 56 mb and a maximum of 1780 Mb. As a consequence, the time spent on completing network IO becomes relevant. Beside, some servers may intentionally limit the reserved bandwidth for each client, therefore the IO time cannot be reduced on the MSASs, even increasing the network bandwidth reserved to each guest.

7.2.1 Install automation results

Guest controllers are in charge of automating UI interaction, in order to step through the installation process. This part of the analysis is crucial: in case guest controllers fail to correctly install jobs, results are biased. In fact, in case of failed installations, resource accesses may be limited to some initial or pre-loading stage of the installer, which may not trigger any PUP installation at that time.

In order to evaluate the quality of the results obtained in our test, it is necessary to inspect first the quality of the UI automation process. To facilitate this task, guest controllers collect internal information about their interaction status. In particular, identifies the following result statuses:

Success — The installer ran correctly and all spawned processes exited with result code 0. At least one new entry is found among installed programs in the control panel. Log report is reported back to host controller.

Partial Success — The installer ran correctly, but not all of the spawned processes exited when timeout was hit. However the control panel shows at least a new item in the list of installed programs. Log report is reported back to host controller.

UI Stuck — The installer ran correctly, but the GUI analyzer detected a UI loop or a

window offering no interaction possibilities. Log report is reported back to host controller.

Timeout — The installation process times out and the guest controller gives up. Guest controller reports failure. Log report is reported back to host controller.

Failure — The host controller was forced to reboot the guest after an error or a timeout. No log report is reported back, data cannot be processed.

Each analysis will be described by one of the statuses introduced above. Yet, we cannot accurately rely on such information, since it just describes the results of UI interaction, which may differ from the epilogue of the installation. In fact, there are situations in which the UI interaction status describes a timeout/failure, but the software installation has been performed correctly. That is the case of installers running the deployed executable at the end of the installation: the gui engine tries to follow the interaction, eventually stalling, giving up with a *timeout* or *UI Stuck* result.

In order to give an accurate ground of truth, we manually checked all the installations performed by this first run. This operation was performed with the help of an ad-hoc Android applicaiton, developed by us for this purpose. The application showed information regarding every job analysis in an intuitive form, so that we could immediately recognize if the program was successfully installed or not. In fact, for each analysis, screenshots were automatically taken by the interaction engine, before performing any interaction. As a result, we were able to review all the interactions performed by the engine. Figure 7.2.1 shows how the ad-hoc Android application facilitates the result checks.

Figure 7.4: Screenshot of the Android application developed for quick report visualization. In this screen only few of the total information is shown, such as detected installed applications, assigned scores to each control and the job's id.

The review process marked each analysis as follows:

Successful — The installer procedure finished correctly. In this case the interaction with the UI simulated closely the approach of *lazy user*.

Failed — The interaction engine was unable to correctly accomplish software installation. As a result the software was only partially installed or not installed at all. Failed installations usually happen when the UI requires advanced user's interaction, such as textual inputs with no predefined default value (e.g. serial numbers, destination paths). Also this case is verified when the interaction engine detects a loop in the UI or some controls are not detected on the UI.

Impossible — The analysis did not run correctly because the input job required unpredictable interactions or because of incompatibilities with the test environment. Binaries for different architectures that x86 fall in this category, as well installers checking for particular hardware or software dependencies. In this case, it would be incorrect to mark those analysis as failed, because failure of this analysis is not addressable to the analysis system.

Once inspected all the reports, we were able to identify the success ratio of the GUI interaction engine, as shown by Figure 7.2.1. The chart shows that 354 jobs over 484 (73%) were correctly handled by the GUI engine. Those installers are confirmed to be executed in a human-like manner, therefore they provide the most accurate data for our analysis. On the other hand, 87 jobs did not complete the installation in a satisfactory way. In such cases, the interaction stopped in the middle of the installation, or did not trigger the installation in the first place. Analysis falling under this category provide the same amount of information that classic sandboxes would collect.

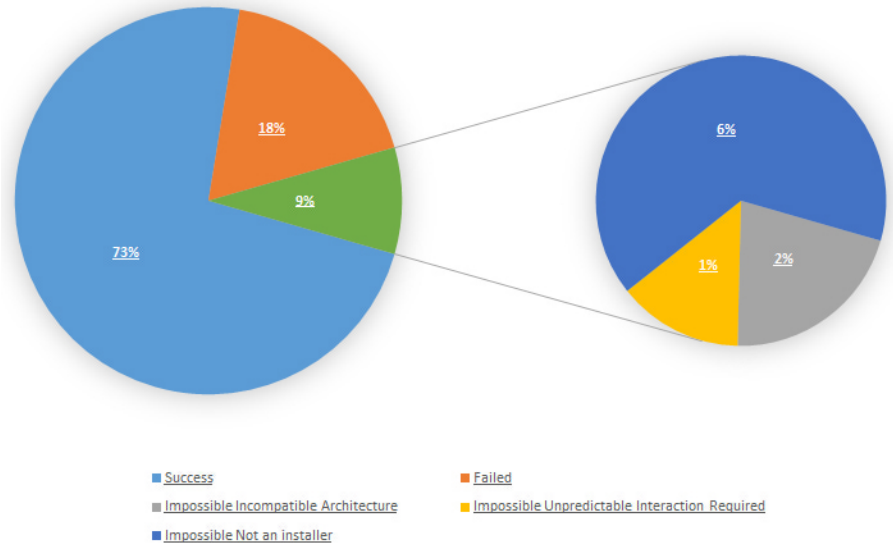


Figure 7.5: Outcomes of the UI interaction engine.

Finally, we identified 43 (9 % of the total inputs) jobs within the *impossible* category. In particular, 9 of them corresponded to different hardware/software architectures (i.e. 64 bits), 6 required advanced interactions and 28 corresponded to uninstallable applications (such as stand-alone applications).

By leaving apart the third category, the success ratio of installations raises to 80.27 %, while failure ratio becomes 19.73 %.

7.2.2 Looking for PUP installers

Identifying PUPs is a hard task. Issues and countermeasures, applied by antimalware vendors, are discussed in Section 2.3. In general, antimalwares analyze each execution separately, looking for a set of suspicious behaviors. Thus, antimalwares defined a set of suspicious patterns that are inspected during software execution. This approach enables immediate reactions: whenever a suspicious behavior is detected, the antimalware may block the installation or warn the user.

Our context is different: unknown software is running in a safe environment, thus immediate reaction is not strictly necessary. Beside, behavioral information collected by our sandboxing system is much wider than what is collected by antimalware products, installed on users' systems. Our system is capable of analyzing suspicious behaviors analogously to the antimalware products (by limiting the scope to each single analysis). At the same time, it also enables data correlation among different analysis.

Common installed applications

A first very simple approach for analyzing results is to check the most common installed applications, as listed in the control panel. Similarly to software libraries, PUPs should appear among the installed applications in specific registry keys, alongside the legitimate applications desired by the user.

In order to prove the effectiveness of this criterion, we executed query shown in Listing 7.1. At first, the query selects experiments that report the installed *product name* among the newly detected applications (Lines 8-12). Consequently, we count the number of distinct installed applications (Line 2) for each product name (Line 13) different from the intaller's one (Lines 5-6). The product name is extracted by the metadata of the analyzed binary and is usually used as human readable identification for the installed application.

```
1 -- Select the program name and the number of distinct experiments ↩
     characterized by the same program name
2 SELECT TRIM(LOWER(name)), COUNT(DISTINCT e.id)
3 FROM cp_new_apps AS apps
4 JOIN experiments AS e ON e.id = apps.experiment_id
5 WHERE e.product_name NOT LIKE CONCAT('%',apps.name,'%')
6 AND apps.name NOT LIKE CONCAT('%',e.product_name,'%')
7 -- Limit this analysis only to installers which show themself among the ↩
     control panel.
8 AND e.id IN ( SELECT id
9     FROM cp_new_apps AS apps
10    JOIN experiments AS e ON e.id = apps.experiment_id
11    WHERE e.product_name LIKE CONCAT('%',apps.name,'%')
12    OR apps.name LIKE CONCAT('%',e.product_name,'%'))
13 GROUP BY TRIM(LOWER(name))
14 ORDER BY COUNT(*) DESC
```

Listing 7.1: Query for selecting the most common installed applications, as listed in the control panel

The execution of such a query on the result set we collected is summarized by Table 7.5

The most common installed application is *Google Chrome*, explicitly listed as installed software by 5 distinct binaries (other 5 executable installers are shared among aggregators, therefore results overlap). Similarly, the Google Toolbar for Internet Explorer is reported as

installed program by 4 different installers. The rest of the records are biasing: most of the listed entries represent legitimate libraries or dependencies for the installing applications. Yet, we inspected the case of Google Chrome. Table 7.6 drills down the results obtained by previous analysis, by focusing on Google Chrome product name. That data is obteined by executing Listing 7.2.

```
select c.name as PUP_name, e.product_name as installer_name, e.product_version, e.id as ExperimentId, a.name
from experiments as e
join cp_new_apps as c on e.id = c.experiment_id
join jobs as j on j.id = e.job_id
join aggregators as a on j.aggregator_id=a.id
where lower(trim(c.name)) not like CONCAT('%',lower(trim(e.product_name)),'%')
and lower(trim(e.product_name)) not like CONCAT('%',lower(trim(c.name)),'%')
and lower(trim(c.name)) = 'google chrome'
order by a.name
```

Listing 7.2: Query for selecting all the installers which explicitly installed Google Chrome.

As we may observe from Table 7.6, Google Chrome is installed as part of 6 different installers: CCleaner, Avast Antivirus, Recuva, PhotoScape, Defraggler, Speeccy. Those applications do not depend on Google Chrome for implementing their functionality, exception made for Avast Antivirus, which may take advantage of SLL key exports offered by Google Chrome browser. From this perspective, Google Chrome configures as a PUP.

Suspicious registry accesses

Analysis of registry accesses represent a valid way of looking for malicious or unwanted applications. In fact, antimalware products heavily rely on registry inspection when to identify potential dangerous behaviors, such the ones listed in 2.3.2. Therefore, we scanned the results looking for the followings:

- Browser extensions / toolbars / plugins
- Autorun and startup configurations
- Installed Root Certificate Authorities

```

-- IE Toolbar settings
select CASE experiments.product_name WHEN '' THEN experiments.description ELSE experiments.product_name END as ProductName, experiments.id as experiment_id, aggregators.name, concat(count(*) over (PARTITION BY AGGREGATORS.name)*100 / count(*) over (), '%') as Aggregator_Percentige
from registry_changes
join experiments on experiments.id = registry_changes.experiment_id
join jobs on experiments.job_id = jobs.id
join aggregators on aggregators.id = jobs.aggregator_id
where lower(full_path) like lower('%Software\\Microsoft\\Internet Explorer\\Toolbar%') and is_new=true and key_value_name is not null
order by aggregators.name;

```

```
-- Chrome Extensions
select distinct 'Chrome', CASE experiments.product_name WHEN '' THEN experiments.description ELSE experiments.product_name END as ProductName, experiments.id as experiment_id, aggregators.name, concat(count(*) over (PARTITION BY AGGREGATORS.name)*100 / count(*) over (), '%') as Aggregator_Percentige
from fs_changes
join experiments on experiments.id = fs_changes.experiment_id
join jobs on experiments.job_id = jobs.id
join aggregators on aggregators.id = jobs.aggregator_id
where lower(fs_changes.path) like lower('%AppData\\Local\\Google\\Chrome\\User Data\\Default\\Extensions%');
```

Listing 7.3: Query for selecting all the installers which installed/modified any toolbar settings for Internet Explorer amd Google Chrome

As we can observe from Table 7.7, 16 jobs caused modifications to registry keys regarding the toolbar configurations of Internet Explorer. In terms of unique binaries (i.e. excluding identical binaries among aggregators), we count 10 different installers. Regarding Google Chrome's extensions, we only detected one positive: PhotoScape, collected from *cnet.com*. However, those results are affected by the fact that Google Chrome was not installed on the SandBox environment. Thus, PUP installers might have refused to install extensions because of the missing browser.

```
select distinct experiments.id as Experiment_id, CASE experiments.product_name WHEN '' THEN (select fname from jobs where id = experiments.job_id) ELSE  experiments.product_name END as product_name, aggregators.name
from registry_changes
join experiments on experiments.id = registry_changes.experiment_id
join jobs on jobs.id = experiments.job_id
join aggregators on aggregators.id = jobs.aggregator_id
where (lower(full_path) like lower('%Software\\Microsoft\\Windows\\CurrentVersion\\Run%')
or lower(full_path) like lower('%Software\\Microsoft\\Windows\\CurrentVersion\\Run%')
or lower(full_path) like lower('%Software\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon\\Userinit%')
or lower(full_path) like lower('%Software\\Microsoft\\Windows NT\\CurrentVersion\Windows%')
) and (is_new = true or is_modified=true)
order by aggregators.name
```

Listing 7.4: Query for inspecting accesses to startup configuration of the operating system.

Another delicate zone of the Windows' Registry regards the autostart management. Listing 7.4 shows how did we query the central database in order to retrieve all the experiments affecting autostart management. Results are listed in Table 7.8. We counted 31 different experiments, with 25 unique product names (i.e. excluding duplicates among different aggregators). It is worth noticing that the number of results provided by this query is higher than the previous one. Many of the listed products may have valid reasons for doing so, such as antimalware applications. The rest of programs may take advantage of update checking capabilities, that justify the autostart requirement. As a consequence, we need to consider the autostart indicator very carefully, because it may affect heavily the false positive detection rate of PUPs.

Eventually we focused our attention to the area of the Windows' Registry regarding system's certificates. In particular we looked for installer that have installed new root certificates into the system, which are therefore trusted by the entire system. To do so, we ran the query shown in Listing 7.5, that produced results exposed in Table 7.9.

```
select distinct experiments.id, product_name, aggregators.name from registry_changes
join experiments on experiments.id = registry_changes.experiment_id
join jobs on jobs.id=experiments.job_id
join aggregators on aggregators.id = jobs.aggregator_id
where (lower(full_path) like lower('%Software\\Microsoft\\SystemCertificates\\root%')
or lower(full_path) like lower('%SOFTWARE\\Policies\\Microsoft\\EnterpriseCertificates%')
or lower(full_path) like lower('%SOFTWARE\\Policies\\Microsoft\\SystemCertificates%')
or lower(full_path) like lower('%SYSTEM\\CurrentControlSet\\Services\\CertSvc %')
) and (is_new=true)
order by aggregators.name
```

Listing 7.5: Query for selecting jobs that installed new root certificates on the system.

Excluding duplicates among different aggregators, we only identify one product which installs a root certificate: Avast Antivirus. The reason why Avast installs a Root CA is explained in [4]. In summary, the antivirus scans HTTPS traffic by performing a man-in-the-middle attack. This is possible thanks to the registration of the Root Certificate Authority in the system.

Network and HTTP traffic

Previous researches have shown that many PUP installers contact remote services, in order to retrieve the most promising PUPs for the system they are running on. After testing particular conditions [86], they might download PUPs and show associated EULAs during the installation process, eventually installing the downloaded binary. As a consequence, we might assume that a relevant part of the PUPs are downloaded on demand at installation time. By implementing such an on-demand feature, PPI distributors can promptly avoid antimalware or antispyware detection, once they recognize an antimalware product being installed on the target system. Moreover, they might avoid re-installing PUPs already available on the target system. Thus, many PPI distributors started to build tiny application downloaders, which query a centralized HTTP server in order to choose the best PUPs to install on the local system. This interaction usually happens over HTTP or HTTPs. The reason why those protocols are usually preferred is mainly related to firewall issues: HTTP traffic can usually propagate through firewalls better than other traffic type.

Once an offer is selected, the relative installer is downloaded, then executed. Assuming those executable files are downloaded via HTTP, then it is logic to analyze HTTP traffic, looking for HTTP downloaded files.

```
select http_downloads.sha1, count(distinct jobs.sha1)
from http_downloads
join experiments on http_downloads.experiment_id = experiments.id
join jobs on experiments.job_id = jobs.id
```

```
5 join file_accesses on file_accesses.file_id = http_downloads.sha1
6 where file_accesses.file_id not in (
7   select file_id from file_accesses where lower(file_accesses.path) like ↩
      lower('%CryptnetUrlCache%') or lower(file_accesses.path) like lower('↩
      %Temporary Internet Files%'))
8 group by http_downloads.sha1
9 having(count(distinct jobs.sha1)>3)
10 order by count(distinct jobs.sha1) desc
```

Listing 7.6: Query for selecting common donwloaded files via HTTP by at least 4 different binaries

Listing 7.6 represents the query executed on the collected data, in order to identify most common downloads among all the analysis. The logic behind the query is straightforward: it lists all the file hashes (downloaded over HTTP) shared by at least 4 different installers, excluding temporary files and ssl-encrypted ones. The minimum number of distinct experiment is arbitrary, and may affect the quantity of returned records. Logically, by lowering this number we might spot more PUP candidates, while increasing the chances of false positives. For instance, different versions of the same product may rely on common modules, downloaded from the internet. Our query does not take into account this possibility, because it groups results by installer's binary. For this reason, we adopted a high threshold: if the same file is downloaded by more than 3 different binaries, we list it. The implicit assumption is that there are no more than 3 versions for each possible product being analyzed.

The execution of Listing 7.6 produced 11 results (as listed in Table 7.10). For each of the obtained files, we listed associated installers, with relative product names and vendors. Then, we elaborated results and we extracted all the combinations of installer's file name sharing the same downloaded file hash. By doing so, we identified 6 different groups of installers. Table 7.11 summarizes the results we eventually obtained. Alongside dependency relationships among applications and files, Table 7.11 also shows the number of expected PUP applications to be installed. This number is manually derived by looking at the interaction screenshots. The number of *offers* shown by the installer are summed up in the *Expected PUPs* column.

Interestingly, we found 6 distinct groups of related applications. Particularly important are groups 1, 2, 4, 5. In fact, those groups clearly enlighten common *installation patterns*: GUIs are similar, installed offers also. As a confirmation, we checked what are the file names associated to the common hashes, in order to inspect what kind of files are they depending on. By looking at Table 7.12, we identify Google Chrome, Google Toolbar and Google Update installers, falling in tha same group. This clearly confirms our previous results: Google Chrome and its toolbar is being installed as part of at least 5 distinct products (group 1). The second group of installers download and execute a couple of files, respectively named *rkverify.exe* and *rkinstaller.exe*, which led to the installation of a PUP. Similarly, *Fences 3* is being installed by applications belonging to group 4 (7 distinct applications, 6 of which install the PUP). Lastly, we identify a single binary file being downloaded by applications belonging to group 5.

It is worth noticing that the 4 groups of interest (1,2,4,5) depend on binary files shared among them and downloaded via HTTP. The other two groups downloaded textual data (the .gz files contained textual files) in form of XML configuration files.

The analysis we applied so far can be easily automated, being based on a bunch of DB queries. By doing so, we define a first indicator for spotting PUPs. The indicator is given

by the execution of the following steps:

- Identify all the HTTP downloaded files shared by at least n distinct applications
- Define groups of applications depending on the same shared files
- Discriminate groups downloading binary files from groups downloading textual files.
- Groups downloading the same binary files most probably install PUPs.

We can evaluate the goodness of the derived indicator by referring to Table 7.11. Group 1 is composed by 3 executable files, which led to the installation of Google Chrome and its toolbar. Similarly, group 2 identifies 2 executable files, which led to the installation of *rkinstaller.exe*, which is a malicious PUP, as according to the majority of antimalware programs [1]. Group 3 is composed by two files: a textual xml configuration file and a compressed .gzip configuration file. Here, we miss one positive: Free Studio was expected to install a PUP. On the other hand, we do not get any false positive. Analogously, group 6 misses one positive, but provides not false positive. Groups 4 and 5 detect all positives (as they should), and we get one false positive on YTD Video Downloader. Therefore, we may conclude that this indicator provides a good detection rate: it correctly spotted both PUP installers and installed PUPs, with a null false positive detection rate. Concerning the successful detection rate, the indicator identified 21 pups over a total of 43 jobs that showed PUP offers during installation, i.e. 48.84 %.

Suspicious file accesses

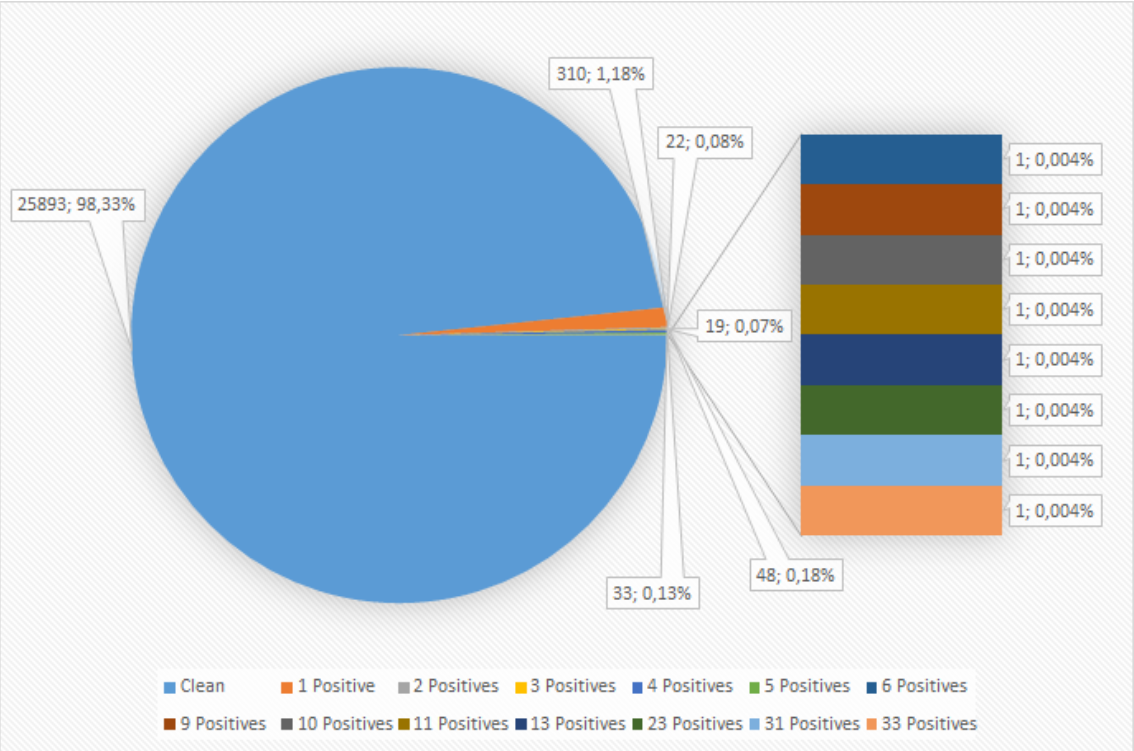
Accesses to the file system provide another valuable source of information for our research. In particular, we wanted to identify all the installers that ended up installing known malicious third party modules. However, data collected from the sandbox does not include any binary that landed on the analysis system. On the contrary, our analysis system takes track of file modifications, through files' hashes. In our test, we identified 194.800 different file hashes. All the hashes were queries against VirusTotal[3].

Virus total recognized 26.333 hashes, 440 of which were positives to at least one scanner. As shown by Figure 7.2.2, about 98.33 % of scanned hashes are considered clean or unknown, while the remaning 2.67 % is positive to at least one scanner.

Particularly relevant are hashes *7dca11208de954c55e352a1ca266b223ece286fa* and *5cafb0702e89b7a0982e33e8a* corresponding to *rtkinstaller.exe* and *rtkverify.exe*. Those hashes resulted malicious to 33 and 31 different scanners, respectively. Most of the scanners classified these files as PUP or adware threats. In accordance with our records, these two files were manipulated by 7 different job analysis, corresponding to 5 distinct product names (2 of the analysis were duplicated hashes regarding different aggregator). More specifically, product names affected by those two files are:

- Free Video Cutter Joiner;
- Free Video Editor;
- FreeMp3VideoConverter;
- Free MP3 Cutter Joiner;

[3] http://www.virustotal.com

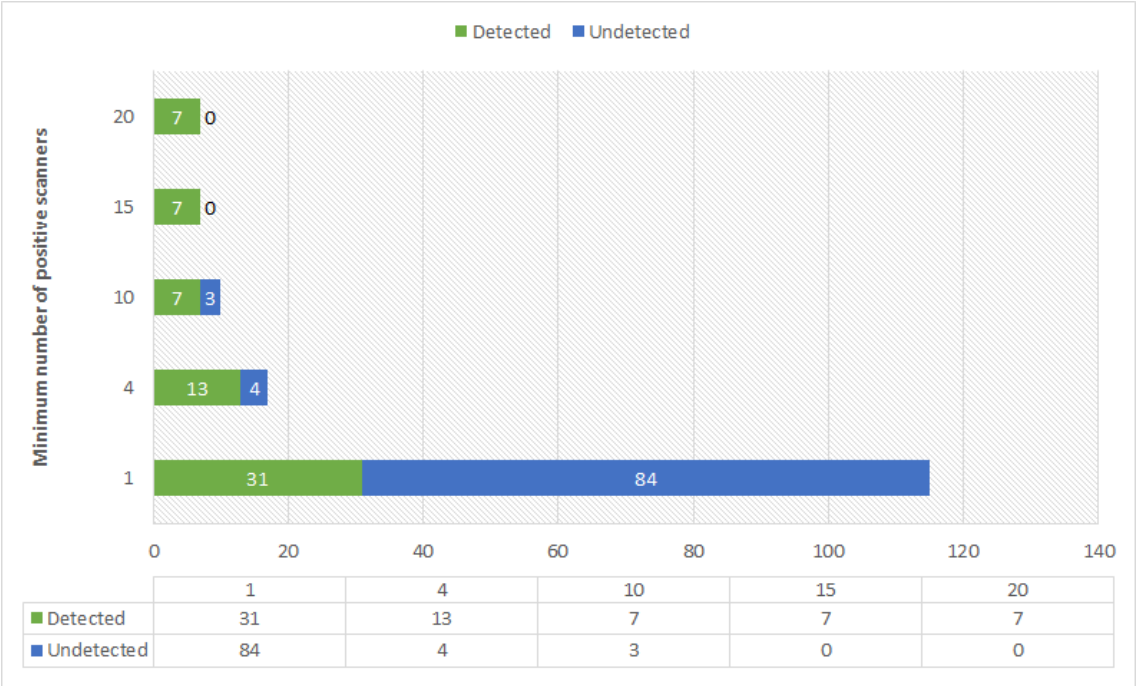


- Free Video To Audio Converter 2016.

The reader may recognize those product names, because they appeared in previous analysis, while inspecting HTTP traffic. Thus, this second analysis confirms the presence of PUPs that we were able to spot by simply inspecting http traffic.

In order to confirm previous results, we extended the analysis to the rest of the hashes considered dangerous by VirusTotal. We studied the detection rate of our indicator in relationship with the number of positive scanners reported by VirusTotal, for each file hash in our database. A summary view of this study is shown in Figure 7.2.2.

As clearly shown by the proposed chart, the quality of HTTP detection strictly depends on the minimum number of positives scanner considered as lower threshold. When taking into account a very low threshold, i.e. 1, the HTTP analysis is capable of detecting just 26.96 % of the potential threats. However, such a low threshold affects the quality of results, causing possibly high false positives. By increasing the threshold to 5, we obtain a substantial improvement, i.e. 76.47 % of detected potential threats. Raising the threshold to 13, the indicator perfectly aligns to the VirusTotal detections.



	1	4	10	15	20
Detected	31	13	7	7	7
Undetected	84	4	3	0	0

Figure 7.6: Relationship between minimum numbers of positive scanners and our HTTP analysis detection rate

Control panel entry	Per experiment	Per product name
google chrome	10	5
google toolbar for internet explorer	8	4
google update helper	5	4
auslogics boostspeed 9	1	1
avg zen	1	1
avira antivirus	1	1
avisynth 2.5	1	1
bonjour	1	1
easeus todo backup free 9.2	1	1
fast browser cleaner 2.0.0.7	1	1
ffdshow [rev 2583] [2009-01-05]	1	1
fmw 1	1	1
haali media splitter	1	1
iobit uninstaller	1	1
jasob 4.1.2	1	1
microsoft .net framework 4 multi-targeting pack	1	1
microsoft application error reporting	1	1
microsoft security essentials	2	1
microsoft sql server compact 3.5 enu	1	1
microsoft visual c++ 2005 redistributable	1	1
microsoft visual c++ 2008 redistributable - x86 9.0.30729.17	1	1
microsoft visual c++ 2008 redistributable - x86 9.0.30729.4974	1	1
microsoft windows sdk for visual studio 2008 express tools for .net framework	1	1
microsoft windows sdk for visual studio 2008 express tools for win32	1	1
mozilla maintenance service	3	1
orange defender antivirus 2	1	1
save-o-gram instagram downloader 3.9	1	1
spyhunter 4	1	1
vc runtimes msi	1	1
visual studio 2012 x86 redistributables	1	1
web companion	1	1
winpcap 4.1.2	1	1
wondershare helper compact 2.5.0	1	1
4videosoft dvd copy 3.2.22	1	1
ziiosoft total video converter version 2.1.4	1	1
apple application support	1	1
apple software update	1	1

Table 7.5: Results regarding the execution of query shown in Listing 7.1

Installer's product name	Installer's version	Experiment id	Aggregator
CCleaner		2992	cnet
Avast Antivirus	12.1.3076.0	2995	cnet
Recuva		3420	cnet
Avast Antivirus	12.2.3126.0	3464	cnet
PhotoScape		3005	cnet
Defraggler		3018	file hippo
Recuva		3014	file hippo
CCleaner		2987	file hippo
Speccy		3179	file hippo
CCleaner		3191	softonic
Recuva		3218	softonic
PhotoScape		3052	softonic
Avast Antivirus	12.1.3076.0	3001	softonic

Table 7.6: Installers which installed Google Chrome as part of their default installation process.

Job Product Name	Experiment id	Aggregator
CCleaner	2992	cnet
RoboForm	3006	cnet
Yahoo! Messenger Suite Install Bootstrapper Setup	3166	cnet
Recuva	3420	cnet
GetGo Download Manager	3417	cnet
Speccy	3179	file hippo
CCleaner	2987	file hippo
AIM for Windows	3176	file hippo
Recuva	3014	file hippo
Defraggler	3018	file hippo
Check Point Install Utility	3321	softonic
CCleaner	3191	softonic
Microsoft Office	3152	softonic
Check Point Install Utility	3321	softonic
Yahoo! Messenger Suite Install Bootstrapper Setup	3103	softonic
Recuva	3218	softonic

Table 7.7: Installers that created registry keys affecting Internet Explorer toolbars

Experiment Id	Product Name	Aggregator
2995	Avast Antivirus	cnet
3017	AVG Internet Security System	cnet
3148	Spybot - Search & Destroy	cnet
3155	Avira Launcher	cnet
3214	Setup Factory 6.0 Runtime	cnet
3242	ACD_WebInstaller(EN-CNET)	cnet
3373	EaseUS Partition Master	cnet
3391	SetupWinCalendarV4.exe	cnet
3393	Freemake Video Converter	cnet
3399	Freemake Video Downloader	cnet
3403	KeyScrambler	cnet
3434	360 Total Security Online Installer	cnet
3442	Windows Media Component Setup Application	cnet
3464	Avast Antivirus	cnet
3091	Spybot - Search & Destroy	file hippo
3117	Microsoft Security Client	file hippo
3129	ZoneAlarm	file hippo
3171	Microsoft® Windows® Operating System	file hippo
3175	Microsoft® Windows® Operating System	file hippo
3001	Avast Antivirus	softonic
3206	QuickTime	softonic
3210	360 Total Security Online Installer	softonic
3215	WhatsappTime-9-1-1dp3.exe	softonic
3228	Connectify 2016	softonic
3259	Filmora	softonic
3274	Microsoft Security Client	softonic
3294	Baidu Antivirus	softonic
3297	Video Converter Ultimate	softonic
3298	Baidu PC Faster	softonic
3321	Check Point Install Utility	softonic
3334	Free USB Disk Security	softonic

Table 7.8: Binaries that affected Windows autostart management

Experiment Id	Product Name	Aggregator
2995	Avast Antivirus	cnet
3001	Avast Antivirus	softonic

Table 7.9: Binaries that affected Windows autostart management

Shared downloaded file	# of jobs	# unique jobs
38dfccb749f98ceaded3a03380eefdc9c752ba18	6	4
543dee8cf997bf0ea7e74b372b33de71dd4648ad	6	6
5cafb0702e89b7a0982e33e8a5c5d52d0e17a1c2	8	6
7dca11208de954c55e352a1ca266b223ece286fa	7	5
8f2ffce11be02356fb37abc2bef0f4d2163e26ff	7	7
91d5182ff719543affb1fe4a4a809cf646a1ea95	6	6
a0fd82fd911f4136ca2ac23c7379cef41170b0ea	3	3
a4a5e3d6cb3921cf2af221559e8b1dafc2fc38b9	6	6
ae323deec0921fff142ea91926e1670e4732c2ff	8	5
b15989b9c3f7bce2d39426d826abfa5683b4dab0	4	4
fb3c1dfcd979099138c592c4d043a7b53ae1b010	7	5

Table 7.10: Hashes of downloaded files shared by more than 3 unique jobs

#	Shared downloaded files	Jobs	PUPs
1	38dfccb749f98ceaded3a03380eefdc9c752ba18 ae323deec0921fff142ea91926e1670e4732c2ff fb3c1dfcd979099138c592c4d043a7b53ae1b010	CCleaner	2
		Recuva	2
		Speccy	2
		Defraggler	2
		PhotoScape	2
2	5cafb0702e89b7a0982e33e8a5c5d52d0e17a1c2 7dca11208de954c55e352a1ca266b223ece286fa	Free Video To Audio Converter 2016	1
		Free MP3 Cutter Joiner	1
		FreeMp3VideoConverter	1
		Free Video Editor	1
		Free Video Cutter Joiner	1
		MKV Player	1
3	91d5182ff719543affb1fe4a4a809cf646a1ea95 a4a5e3d6cb3921cf2af221559e8b1dafc2fc38b9	Free Studio	1
		Free Audio Converter	0
		Free 3GP Video Converter	0
		Free YouTube To MP3 Converter	0
		Free YouTube Download	0
		Free Video to MP3 Converter	0
4	543dee8cf997bf0ea7e74b372b33de71dd4648ad 8f2ffce11be02356fb37abc2bef0f4d2163e26ff	WindowBlinds_setup.exe	1
		Fences3-cnet-setup.exe	1
		WindowBlinds_cnet_setup.exe	1
		ObjectDock.exe	1
		DeskScapes8_setup.exe	1
		ObjectDock-cnet-setup.exe	1
		YTD Video Downloader	1
5	b15989b9c3f7bce2d39426d826abfa5683b4dab0	Freemake Video Converter	1
		iPadian	1
		The KMPlayer	1
		Freemake Video Downloader	1
6	a0fd82fd911f4136ca2ac23c7379cef41170b0ea	IObit Uninstaller	0
		Start Menu 8	1
		IObit Malware Fighter 4	0

Table 7.11: Groups of installers depending on same dowloaded files

#	File name	Mime type	Source host
1	googletoolbarinstaller....exe	application/x-dosexec	dl.l.google.com
	..._chrome_installer.exe	application/x-dosexec	r1.sn-ovgq0oxu-5goe.gvt1.com
	GoogleUpdateSetup.exe	application/x-dosexec	r4.sn-ovgq0oxu-5goe.gvt1.com
2	rkinstaller.exe	application/x-dosexec	post.securestudies.com
	rkverify.exe	application/x-dosexec	post.securestudies.com
3	versions_tmp.xml.gz	application/gzip	tools.dvdvideosoft.netdna-cdn.com
	versions_tmp.xml	application/xml	tools.dvdvideosoft.netdna-cdn.com
4	Fences 3_setup.exe	application/x-dosexec	stardock.cachefly.net
	getCountry	text/plain	ytd01.greentreeapps.ro
5	70D9F0F8_stp.CIS	application/octet-stream	cdnus.meyepey.com
6	F4$oCvbABEqJXwRF.tmp	text/plain	gs1.wpc.edgecastcdn.net

Table 7.12: File type and network source for shared http downloads regarding Table 7.11

Chapter 8

Conclusions and future work

In this work we presented a distributed architecture of MSAS aimed at automating analysis of software installers, mainly targeting PUP installers. We also implemented a prototype reflecting the architectural structure provided, in order to evaluate the real possibilities of such a tool. The system implemented so far combines classic techniques already adopted in the malware fighting, yet it discloses new angles from which to study the PUP threat. In this chapter we will summarize what are the contributions that our work brings to the PUP fight, comparing pros and cons with state-of-the-art competing technologies. Furthermore, current limitations and possible issues are discussed, hence possible future improvement strategies are introduced.

8.1 Contributions

The structure of the MSAS we designed and developed in this document is, at best of our knowledge, the first kind of sandboxed environment specifically addressing PUPs. As such, it attempts to solve a number of limitations that current malware sandbox anslysis systems do not consider. In general, we might identify two major contributions that differentiate our design from the other sandboxes: automation of UI interfaces and data correlations on server side. Secondary objectives included scalability capabilities and MSAS detection avoidance.

Our UI interaction system was able to automatically handle 80.27 % of the possible installations in a human-like way, based on the interaction policies currently implemented. The remaining 20 % of the analysis provide results comparable to MSASs currently freely accessible on the Internet (i.e. resource auditing with no UI interaction). The interaction mechanism adopted proved to be general enough for interacting with the majority of software installers, independently from their installer framework. This represent a strong advantage in respect with other sandboxing systems, which are capable of automating none or a strict number of installation interfaces, depending on the underlying installer frameworks. The UI system also collects basic UI data during analysis and takes advantage of the background auditing system in order to decide when to interact with the installer's GUI.

Thanks to the analysis of collected data, we were able to correlate results obtained by distinct installers. HTTP traffic analysis, together with native file system access monitoring, proved to be crucial when dealing with PUP detection. We were able to define a simple indicator which correctly identified 48.84 % of the expected PUPs, without relying on any

signature or blacklisted hashes database. The quality of the indicator has been confirmed by the analysis of the detected files via VirusTotal. Moreover, the indicator was able to detect 76 % of the PUPs rated as malicious by at least 4 scanners of VirusTotal. The detection rate became 100 % when the number of minimum scanners considered raises to 15.

Differently from the other MSAS platform we are aware of, ours collects information on the UI interface presented, such as recognized buttons, checkboxes, radio buttons and labels. Although this data has not been analyzed yet, it is our believe that it might disclose new information useful for PUP installer analysis. This aspect represents another important contribution that our MSAS brings to the PUP detection research area.

Another relevant characteristic distinguishing our work regards the bare-metal anaysis support. The modular - network driven - architecture, combined with platform independence, allows guest controllers to run virtually on any hardware supported by the Guest OS (a the moment Windows 7). Moreover, the structure of the analysis system is modular and is designed to support multiple hypervisors. At the time of writing, only virtual box support has been implemented, however little effort is required in order to support other virtualization engines, such as Openstack or VMWare.

From the perspective of scalability, the infrastructure reacted as expected. Our tests registered a significant improvement of throughput corresponding to the increment of guest nodes added to the distributed system. In fact, raising the number of guests from 8 to 14 (75 %), the total average throughput grew by a 72.83 % factor, alongside with the reduction of duration of test. When running with a total of 14 guests (handled by 2 host controllers) the current prototype was able to collect and analyze 484 distinct binaries in less than 11.5 hours, running on a relatively old server infrastructure (6 years old processors). The average time for a single jobs analysis is around 12 minutes. Thanks to the scalable architecture of the system, multiple analysis can happen simultaneously, on different host controllers running distinct virtualization engines. As a result, with two 6-years-old server blades, running heterogeneous operating systems(one Linux server and one Windows server), we reached an average throughput of 0.72 jobs per minute.

8.2 Limitations

The analysis system we presented is affected by some limitations, some of which may be easily overcome in future versions. At first, the current implementation of API hooking is based on *Microsoft Detours x86* and on a custom DLL we implemented for 32 bit operating systems. Thus, the current prototype is only capable of analyzing 32 bit installers. However, there are several more recent libraries providing API hooking capabilities, such as *Deviare* [7] or *EasyHook* [2]. Similarly, we can recompile the injected DLL for 64 bit systems and code the injector in such a way it decides dynamically which versions of the DLL to inject in each process.

A second limitation is represented by encrypted http traffic not being analyzed. The current version of the analysis system does not enable SLL/TLS inspection. To solve this problem, we might install a HTTPS proxy on the sniffers, configuring all the VMs to use that proxy while accessing HTTPS traffic. There are open source products that address exactly this problem, such as *mitmproxy* [6].

Other constraints of the actual prototype regard the GUI interaction engine. While the current state of the engine is sufficient to interact with the majority of installers we gathered, it still requires to be fine tuned and optimized. The prototype we implemented

and tested in this document correctly interacted with 80 % of the installers. It is our believe that this result can be increased considerably.

8.3 Future work

Our MSAS implementation demonstrates that a great part of the PUP analysis can be automated. However, we introduced a number of limitations that should be overcome in the future. In particular it is crucial to add support for 64 bit binaries, by developing a 64bit version of the injected DLL, taking advantage of a API Hooking library supporting 64 bit applications. Secondly, the prototype lacks of HTTPS traffic inspection capabilities. Lastly, the GUI interaction mechanism requires tuning and improvements in order to be more effective.

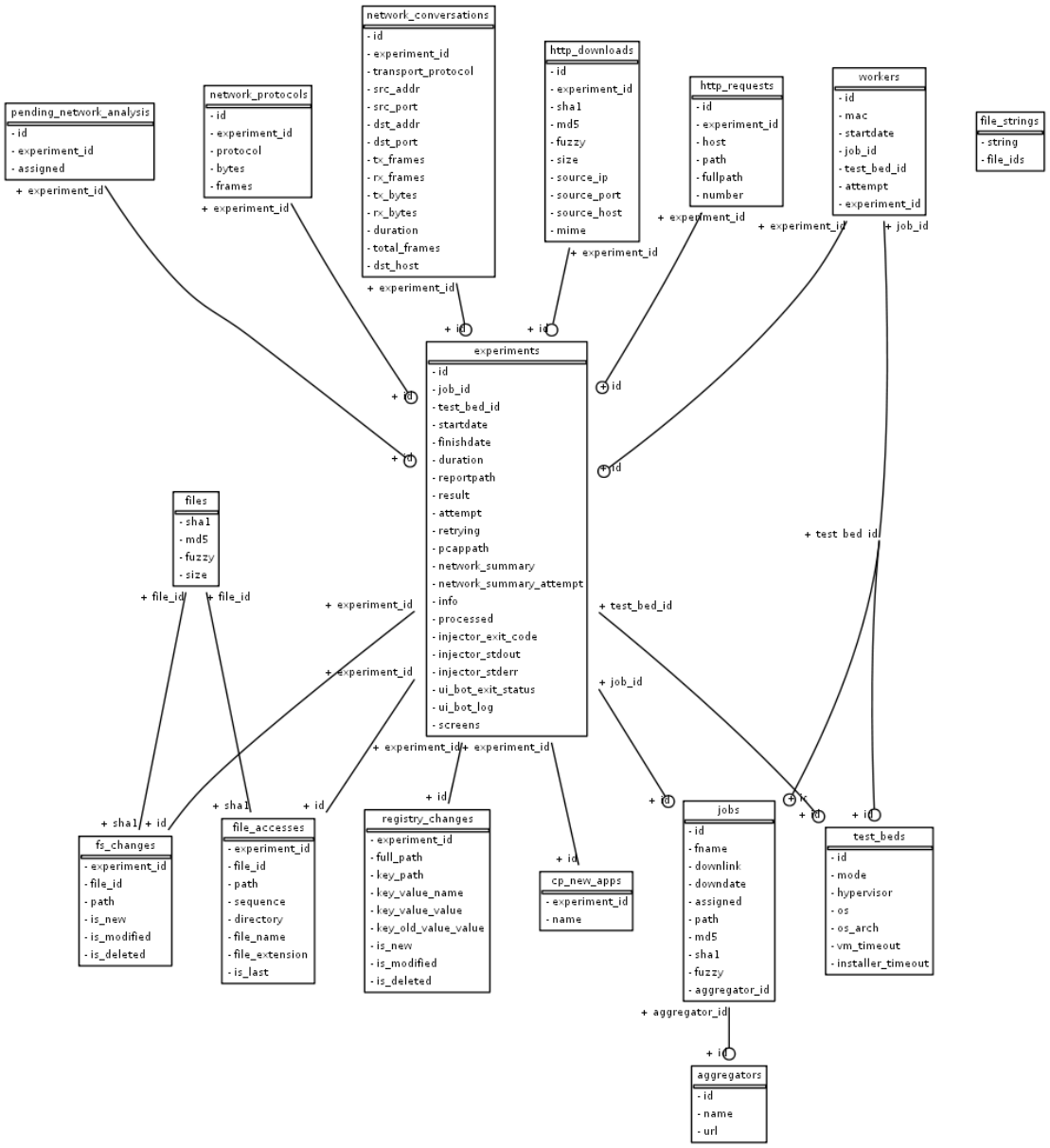
Once fixed the current limitations of the developed MSAS, it is our intention to focus on data analysis, in order to synthesize new possible PUP indicators. In fact, in this work we just scratched the surface of data analysis, yet we were able to identify more than 48 % of expected PUPs, just considering HTTP traffic, without relying on any external PUP database. We believe that new PUP identification techniques can be disclosed by analyzing PUP installer's GUIs. Machine learning technology can be used in order to classify PUP installers from legitimate clean software.

Thanks to the ability of running the same job on multiple guest configurations, it might be interesting to compare behaviors of installers when running on bare metal and virtual environment. This study may reveal which are, if any, sandbox detection mechanisms applied by PUP installers.

Lastly, we plan to generalize crawlers, in order to collect installers from all over the web automatically, pushing jobs into the central DB any time a new product is identified. To do so, crawlers need to evolve in such a way they recognize installer products, discarding stand-alone executable files.

Appendix A

Database schema



Appendix B

Hooked APIs

Table B.1 shows the list of APIs which are hooked by the HookingDLL. For each hook it is also reported the original Windows module and a short description.

Method	Module	Detour Function	Description
NtCreateFile	ntdll.dll	MyNtCreateFile	The ZwCreateFile routine creates a new file or opens an existing file.
NtOpenFile	ntdll.dll	MyNtOpenFile	The NtOpenFile routine opens an existing file, directory, device, or volume.
NtDeleteFile	ntdll.dll	MyNtDeleteFile	The ZwDeleteFile routine deletes the specified file.
NtCreateKey	ntdll.dll	MyNtCreateKey	The ZwCreateKey routine creates a new registry key or opens an existing one
NtOpenKey	ntdll.dll	MyNtOpenKey	The ZwOpenKey routine opens an existing registry key.
NtSetInformationFile	ntdll.dll	MyNtSetInformationFile	The ZwSetInformationFile routine changes various kinds of information about a file object.
NtClose	ntdll.dll	MyNtClose	The ZwClose routine closes an object handle.
CreateProcessInternalW	kernel32.dll	MyCreateProcessInternalW	Internal function that creates a new process.
ExitProcess	kernel32.dll	MyExitProcess	Ends the calling process and all its threads.

Table B.1: API hooked in the current version

File monitorng is obtained by hooking three low level file access APIs: NtCreateFile(), NtOpenFile(), NtDeleteFile and NtClose(). Whenever one of those functions is called by the target process, the HookingDLL notifies the guest controller, which calculates the hash of the file being processed and updates the logging data structures accordingly.

The Windows registry is monitored in a different way. Instead of hooking all the API providing writing capabilities on the registry, only two methods are monitored: NtOpenKey() and NtClose(). When a registry key is being created or opened, the HookingDLL notifies the guest controller, which saves the relative registry path and its original values into

a hash-table. When target process finishes, the guest controller scans that table and calculates all the differences with the original contained values.

Bibliography

[1] Antivirus scan for f8d11b1e3e027355a11163049b530de4fd67183abd08a691d5d18744653ef575 at utc - virustotal. https://www.virustotal.com/it/file/f8d11b1e3e027355a11163049b530de4fd67183abd08a691d5d18744653ef575/analysis/1474630548/. (Accessed on 09/23/2016).

[2] Easyhook. https://easyhook.github.io/. (Accessed on 09/24/2016).

[3] Embedding principles - sciter. http://sciter.com/developers/embedding-principles/. (Accessed on 07/19/2016).

[4] Explaining avastś https scanning feature. https://blog.avast.com/2015/05/25/explaining-avasts-https-scanning-feature/. (Accessed on 09/22/2016).

[5] Ibm - rational functional tester. http://www-03.ibm.com/software/products/it/functional. (Accessed on 07/20/2016).

[6] Mitmproxy. https://mitmproxy.org/. (Accessed on 09/24/2016).

[7] Nektra - custom software development company. http://www.nektra.com/. (Accessed on 09/24/2016).

[8] Pay-per-install: The new malware distribution network. (Accessed on 06/03/2016).

[9] Silent install builder. http://www.silentinstall.org/. (Accessed on 09/04/2016).

[10] Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010.

[11] Kujawa A. Potentially unwanted miners, malwarebytes unpacked. https://blog.malwarebytes.org/cybercrime/2013/11/potentially-unwanted-miners-toolbar-peddlers-use-your-system-to-make-btc/. (Accessed on 06/03/2016).

[12] AForge. Dilatation class. http://www.aforgenet.com/framework/docs/html/88f713d4-a469-30d2-dc57-5ceb33210723.htm. (Accessed on 07/25/2016).

[13] AForge. Fillholes class. http://www.aforgenet.com/framework/docs/html/68bd57bd-1fd6-6c4e-4500-ed4726bc836e.htm. (Accessed on 07/25/2016).

[14] AForge. Sisthreshold class. http://www.aforgenet.com/framework/docs/html/39e861e0-e4bb-7e09-c067-6cbda5d646f3.htm. (Accessed on 07/25/2016).

[15] A. Ahmed. Test automation for graphical user interfaces: A review. In *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, pages 1–6, Jan 2014.

[16] P. Aho, M. Suarez, T. Kanstren, and A. M. Memon. Murphy tools: Utilizing extracted gui models for industrial software testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 343–348, March 2014.

[17] Jason B. Defining rules for acceptable adware. In *Proceedings of the Fifteenth Virus Bulletin Conference*, 2005.

[18] J. Blanchette and M. Summerfield. *C++ GUI Programming with Qt4*. Pearson Education, 2008.

[19] Rainer Böhme and Stefan Köpsell. Trained to accept? a field experiment on consent dialogs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2403–2406, New York, NY, USA, 2010. ACM.

[20] M. Boldt and B. Carlsson. Privacy-invasive software and preventive mechanisms. In *Systems and Networks Communications, 2006. ICSNC '06. International Conference on*, pages 21–21, Oct 2006.

[21] M. Boldt, A. Jacobsson, N. Lavesson, and P. Davidsson. Automated spyware detection using end user license agreements. In *Information Security and Assurance, 2008. ISA 2008. International Conference on*, pages 445–452, April 2008.

[22] Pat Brenner. Spy++ internals | visual c++ team blog. https://blogs.msdn.microsoft.com/vcblog/2007/01/16/spy-internals/, January 2007. (Accessed on 07/19/2016).

[23] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[24] Wei Hoo Chong. Loader cart automation enhancement. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, pages 1096–1101, Feb 2012.

[25] R.S. Choraś. *Image Processing & Communications Challenges 3*. Advances in Intelligent and Soft Computing. Springer Berlin Heidelberg, 2011.

[26] Symantec Corp. Risks. https://www.symantec.com/security_response/landing/risks/. (Accessed on 06/13/2016).

[27] C. D. Curran. Combating spam, spyware, and other desktop intrusions: legal considerations in operating trusted intermediary technologies. *IEEE Security Privacy*, 4(3):45–51, May 2006.

[28] Benjamin E. Spyware, adware, and malware– research, testing, legislation, and suits. http://www.benedelman.org/spyware/. (Accessed on 06/03/2016).

[29] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014.

[30] J. Faircloth, J. Beale, R. Temmingh, H. Meer, C. van der Walt, and HD Moore. *Penetration Tester's Open Source Toolkit*. Elsevier Science, 2006.

[31] David French and William Casey. 2 fuzzy hashing techniques in applied malware analysis. *Results of SEI Line-Funded Exploratory New Starts Projects*, page 2, 2012.

[32] T.S. Garfinkel, M. Rosenblum, D. Boneh, J. Mitchell, and Stanford University. Computer Science Department. *Paradigms for Virtualization Based Host Security*. Stanford University, 2010.

[33] Nathaniel Good, Rachna Dhamija, Jens Grossklags, David Thaw, Steven Aronowitz, Deirdre Mulligan, and Joseph Konstan. Stopping spyware at the gate: A user study of privacy, notice and spyware. In *Proceedings of the 2005 Symposium on Usable Privacy and Security*, SOUPS '05, pages 43–52, New York, NY, USA, 2005. ACM.

[34] Aryeh Goretsky. Problematic-unloved-argumentative.pdf. http://go.eset.com/us/resources/white-papers/Problematic-Unloved-Argumentative.pdf. (Accessed on 06/03/2016).

[35] A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria. Reverse engineering of gui models for testing. In *5th Iberian Conference on Information Systems and Technologies*, pages 1–6, June 2010.

[36] R.A. Grimes. *Honeypots for Windows*. Apresspod Series. Apress, 2005.

[37] A. Gupta and P. Anand. Focused web crawlers and its approaches. In *Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 2015 International Conference on*, pages 619–622, Feb 2015.

[38] D. Harley. *AVIEN Malware Defense Guide for the Enterprise*. Elsevier Science, 2011.

[39] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[40] P. Joshi, D.M. Escriva, and V. Godoy. *OpenCV By Example*. Packt Publishing, 2016.

[41] Kaspersky. Participate in whitelist. http://whitelist.kaspersky.com/whitelist_program. (Accessed on 06/13/2016).

[42] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplement:91 – 97, 2006. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

[43] Jesse Kornblum. Fuzzy hashing, 2012.

[44] D. Kusnetzky. *Virtualization: A Manager's Guide*. Real Time Bks. O'Reilly Media, Incorporated, 2011.

[45] F-Secure Labs. Submit a sample. https://www.f-secure.com/en/web/labs_global/submit-a-sample. (Accessed on 06/13/2016).

[46] Malwarebytes Labs. Malwarebytes adopts aggressive pup policy. https://blog.malwarebytes.com/malwarebytes-news/2013/07/malwarebytes-adopts-aggressive-pup-policy/. (Accessed on 09/04/2016).

[47] Malwarebytes labs. Pup reconsideration information. https://it.malwarebytes.org/pup/. (Accessed on 06/03/2016).

[48] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[49] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *ACSAC*, 2014.

[50] Malwarebytes. What are pup detections. https://support.malwarebytes.com/customer/portal/articles/1834873-what-are-pup-detections-are-they-threats-and-should-they-be-deleted-?b_id=6438. (Accessed on 09/04/2016).

[51] P. McFedries. Technically speaking: The spyware nightmare. *IEEE Spectrum*, 42(8):72–72, Aug 2005.

[52] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Computer Security Applications Conference, 1997. Proceedings., 13th Annual*, pages 175–181, Dec 1997.

[53] Jianpeng Mo. how_to_identify_pua.pdf. file:///C:/Users/webking/Downloads/how_to_identify_pua.pdf. (Accessed on 06/13/2016).

[54] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do windows users follow the principle of least privilege?: investigating user account control practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, page 1. ACM, 2010.

[55] MSDN. About messages and message queues (windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms644927(v=vs.85).aspx. (Accessed on 07/18/2016).

[56] MSDN. About window procedures (windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms633569(v=vs.85).aspx. (Accessed on 07/18/2016).

[57] MSDN. Enumchildwindows function (windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms633494(v=vs.85).aspx. (Accessed on 07/19/2016).

[58] MSDN. Filtering registry calls (windows drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff545879(v=vs.85).aspx. (Accessed on 07/26/2016).

[59] MSDN. Kernel patch protection: frequently asked questions - windows 10 hardware dev. https://msdn.microsoft.com/en-us/library/windows/hardware/dn613955(v=vs.85).aspx. (Accessed on 07/27/2016).

[60] MSDN. Libraries and headers (windows drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff554288(v=vs.85).aspx. (Accessed on 07/25/2016).

[61] MSDN. Windows forms. https://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx. (Accessed on 07/23/2016).

[62] Charles Petzold. *Programming Windows, Fifth Edition*. Microsoft Press, Redmond, WA, USA, 5th edition, 1998.

[63] C. Pickard and S. Miladinov. Rogue software: Protection against potentially unwanted applications. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 1–8, Oct 2012.

[64] Matt Pietrek. Peering inside the pe: A tour of the win32 portable executable file format. https://msdn.microsoft.com/en-us/library/ms809762.aspx. (Accessed on 06/22/2016).

[65] Realtimepublishers.com and D.M.J. Sanoy. *The Definitive Guide to Windows Installer Technology for System Administrators*. Realtimepublishers.com, 2002.

[66] Paul Robichaux. *Managing the Windows 2000 Registry*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[67] D.J. Rogers. *Broadband Quantum Cryptography*. Synthesis lectures on quantum computing. Morgan & Claypool Publishers, 2010.

[68] R. Rosen. *Linux Kernel Networking: Implementation and Theory*. Books for professionals by professionals. Apress, 2014.

[69] Mark Russinovich. Security: Inside windows vista user account control. https://technet.microsoft.com/en-us/magazine/2007.06.uac.aspx. (Accessed on 06/09/2016).

[70] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7 (Windows Internals)*. Microsoft Press, 2012.

[71] M.E. Russinovich, D.A. Solomon, and A. Ionescu. *Windows Internals*. Number pt. 1 in Developer Reference. Pearson Education, 2012.

[72] M.E. Russinovich, D.A. Solomon, and A. Ionescu. *Windows Internals*. Number pt. 1 in Developer Reference. Pearson Education, 2012.

[73] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226, April 2010.

[74] Mark B. Schmidt and Kirk P. Arnett. Spyware: A little knowledge is a wonderful thing. *Commun. ACM*, 48(8):67–70, August 2005.

[75] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition, 2012.

[76] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[77] N.P. Smart. *Cryptography Made Simple*. Information Security and Cryptography. Springer International Publishing, 2015.

[78] Sophos. Pua protection and adware security | pua threat detection and removal. https://www.sophos.com/it-it/threat-center/threat-analyses/adware-and-puas.aspx. (Accessed on 06/13/2016).

[79] Sophos. Submitting samples of suspicious files to sophos. https://www.sophos.com/en-us/support/knowledgebase/11490.aspx. (Accessed on 06/13/2016).

[80] T. Soulami. *Inside Windows Debugging*. Developer Reference. Pearson Education, 2012.

[81] Chad Spensky, Hongyi Hu, and Kevin Leach. Lo-phi: Low-observable physical host instrumentation for malware analysis. 2016.

[82] Andreas Stamminger, Christopher Kruegel, Giovanni Vigna, and Engin Kirda. Automated spyware collection and analysis. In *International Conference on Information Security*, pages 202–217. Springer, 2009.

[83] Vlasta Stavova, Vashek Matyas, and Mike Just. On the impact of warning interfaces for enabling the detection of potentially unwanted applications. 2016.

[84] L. Stevenson and N. Altholz. *Rootkits For Dummies*. –For dummies. Wiley, 2006.

[85] Symantec. Sonar.pua!gen5 | symantec. https://www.symantec.com/security_response/writeup.jsp?docid=2015-061218-1537-99. (Accessed on 06/13/2016).

[86] Kurt Thomas, Juan A. Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, Lucas Ballard, Robert Shield, Nav Jagpal, Moheeb Abu Rajab, Panayiotis Mavrommatis, Niels Provos, Elie Bursztein, and Damon McCoy. Investigating commercial pay-per-install and the distribution of unwanted software. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 721–739, Austin, TX, August 2016. USENIX Association.

[87] R. Tian, R. Islam, L. Batten, and S. Versteeg. Differentiating malware from cleanware using behavioural analysis. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 23–30, Oct 2010.

[88] Andrew Tridgell. Spamsum algorithm. https://www.samba.org/ftp/unpacked/junkcode/spamsum/README. (Accessed on 06/15/2016).

[89] A.B. Tucker. *Computer Science Handbook, Second Edition*. CRC Press, 2004.

[90] S.E. Umbaugh. *Digital Image Processing and Analysis: Human and Computer Vision Applications with CVIPtools, Second Edition*. CRC Press, 2016.

[91] VirusTotal. Antivirus scan for 865b480f7ec90ffb8738f581658f24b6f81cbc6fefb0f540f01644a51e51f167 at 2015-10-27 01:23:55 utc. https://www.virustotal.com/en/file/865b480f7ec90ffb8738f581658f24b6f81cbc6fefb0f540f01644a51e51f167/analysis/. (Accessed on 06/03/2016).

[92] W3C.org. Operating system statistics. http://www.w3schools.com/browsers/browsers_os.asp. (Accessed on 06/07/2016).

[93] Phil Wilson. *The Definitive Guide to Windows Installer*. APress, 2004.

[94] Y. Wiseman. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies: Techniques and Technologies*. Premier reference source. Information Science Reference, 2009.

[95] K. Yoshioka, Y. Hosobuchi, T. Orii, and T. Matsumoto. Vulnerability in public malware sandbox analysis systems. In *Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on*, pages 265–268, July 2010.