

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



Prometheus: A Web-based Platform for Analyzing Banking Trojans

Relatore: Prof. Federico Maggi
Correlatore: Ing. Michele Carminati

Andrea Braschi, matricola 797136
Andrea Continella, matricola 797581

Anno Accademico 2013-2014

Contents

Sommario	17
1 Introduction	19
2 State of the art	23
2.1 Overview of Banking Trojans	23
2.1.1 The underground economy	24
2.1.2 The fraud scheme	25
2.1.3 Man in the Browser attacks and WebInject	26
2.1.4 Hooking mechanism	29
2.1.5 The Automatic Transfer System	31
2.1.6 Man in the Mobile	31
2.1.7 A new Target: Bitcoin	32
2.2 The Zeus Crimeware Toolkit	33
2.2.1 Keylogging, formgrabbing, screenshotting and click-grabbing	34
2.2.2 Components	35
2.3 Banking Trojan detection	37
2.3.1 Antivirus detection	37
2.3.2 Reverse Engineering	38
2.3.3 Other related work	39
2.4 Challenges and Goals	39
3 Prometheus: the approach	43
3.1 Proposed Approach	43
3.2 Prometheus Analyses	44
3.2.1 URL analysis	45
3.2.2 Sample analysis	45
3.3 System Overview	46
3.3.1 Phase 1: Data Collection	47

3.3.2	Phase 2: Data Processing	48
3.3.3	Phase 3: Results Elaboration	49
3.4	The Crawler	52
4	Prometheus: implementation	55
4.1	The Overall Architecture	55
4.2	Libraries and Tools	57
4.3	The Configuration file	58
4.4	Cuckoo	59
4.4.1	Quick Overview	60
4.5	Back-end	61
4.5.1	Cuckoo analysis packages	62
4.5.2	Executor	63
4.5.3	Submit Server	64
4.5.4	Scheduler	65
4.5.5	VMs Server	66
4.5.6	Comparisons Manager	67
4.5.7	Comparer.jar	68
4.5.8	Memory Analysis	69
4.6	Back-end functioning	70
4.6.1	Phase 1: Data Collection	70
4.6.2	Phase 2: Data Processing	71
4.6.3	Phase 3: Results elaboration	74
4.7	Web Service	76
4.7.1	Analysis Submission and Results Retrieval	78
4.8	The Crawler	80
5	Experimental Evaluation	85
5.1	Deployment	85
5.2	Challenges	86
5.3	Datasets construction	86
5.3.1	Samples dataset	86
5.3.2	URLs list creation	86
5.3.3	Ground truth	88
5.4	Experiments	89
5.4.1	False positives discussion	89
5.4.2	False negatives discussion	91
5.4.3	Memory analysis discussion	91
5.4.4	Performance	92

6	Conclusions	95
6.1	Limitations	96
6.2	Future Works	97
	Bibliography	99
A	The Database	103
B	Determine Malware Activation Time	105

List of Figures

2.1	Number of infected computers in 2013	24
2.2	Number of infected computers in 2013 by country	24
2.3	The fraud scheme	27
2.4	Example of a real injection	28
2.5	WebInject Hooking mechanism	28
2.6	Man in the Mobile attack scheme	31
2.7	Example of virtual keyboard	34
2.8	Zeus control panel interface	36
3.1	I/O Scheme of the URL analysis	45
3.2	I/O Scheme of the sample analysis	46
3.3	Overview of the sample analysis	47
3.4	Graphical explanation of differences comparison and filtering	51
3.5	Zeus tracker monitor page	52
3.6	SpyEye tracker monitor page	53
4.1	Prometheus architecture	56
4.2	Prometheus architecture, crawler interaction	56
4.3	Comparisons Manager data structure	69
4.4	Link relations between web pages.	77
4.5	Snapshot of the DOM visualization page (no injections). . . .	78
4.6	Snapshot of the DOM visualization page (with injections). . .	79
4.7	Snapshot of the search through hashes page.	80
4.8	Snapshot of the Top Ten targeted URLs page.	81
4.9	Snapshot of the sample submission page.	81
4.10	Snapshot of the URL submission page.	82
4.11	Particular of the results page.	82
4.12	Loading bar in results page.	82
4.13	Keys extraction in results page.	83
4.14	Regular expressions matching in results page.	83
4.15	Regular expressions extraction in results page.	83

5.1	False Positive Rate depending on the ε threshold	90
5.2	False Positive Rate depending on the number of VMs used . .	91
5.3	Speed and Scalability of Prometheus	94
5.4	Trade-off between Performance and False Positive Rate . . .	94
A.1	EER schema of the database.	104
B.1	Table of API hooks in cuckoo results.	106
B.2	WebInject hook.	106

List of Tables

2.1	Typical APIs hooked by ZeuS	30
5.1	Samples dataset	87
5.2	Most injected websites	89
5.3	Most found regular expressions	92

List of code and log excerpts

2.1	Portion of a real leaked Zeus webinject.txt	35
4.1	Prometheus configuration file	59
4.2	Webinject Cuckoo package	62
4.3	Scheduler run method	66
4.4	Scheduler submit_sample_analysis method	66
4.5	Comparisons Manager checking new comparisons to do after the arrival of a new DOM	72
4.6	YARA rule defined to extract WebInject targets	74
4.7	Example of differences JSON file	75
4.8	Crawler log example	80
A.1	Example Query	104

Abbreviations

API	Application Programming Interface
AJAJ	Asynchronous Javascript and JSON
C&C	Command and Control
DNS	Domain Name System
DOM	Document Object Model
DLL	Dynamic-Link Library
HTML	Hypertext Markup Language
HTTP(S)	Hypertext Transfer Protocol (Secure)
IP	Internet Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
MaaS	Malware-as-a-Service
OS	Operating System
OTP	One-Time Password
P2P	Peer-To-Peer
PIN	Personal Identification Number
REGEX	Regular Expression
URL	Uniform Resource Locator
VM	Virtual Machine
XML	eXtensible Markup Language

Abstract

Nowadays, banking trojans are reaching alarming levels of sophistication. New variants (the most famous examples are ZeuS, SpyEye and Citadel) are constantly being introduced to avoid detection by antivirus software on the victim's PC and to make it difficult for banks and account holders to spot fraud attempts as they occur.

Furthermore, these trojans are sold on underground markets as "toolkits" that include development kits, web-based administration panels, builders and easy-to-use customization procedures. The main consequence is that anyone, independently on the skill level, can buy a malware builder and create a customized sample.

Banking trojans exploit API hooking techniques to be able to intercept all the data going through the browser even when the connection is encrypted. This kind of malware contains also a WebInject module able to modify web pages. This module is used by the attackers to add new fields in forms in order to steal the target information.

We propose a web based platform, Prometheus, that analyzes banking trojans exploiting the visible DOM modifications in the HTML page that they cause. Prometheus is able, independently on the trojan's family or version, to detect the injections performed by the malware and, through memory forensic techniques, to extract the targets (the URLs of the web pages monitored and modified by the malware). In conclusion, Prometheus is useful to malware analysts because it significantly reduces the need of manual reverse engineering.

We evaluated Prometheus on 53 ZeuS samples and 62 real banking websites showing that our system correctly detects the injections performed by trojans and successfully extracts the WebInject targets.

Sommario

Attualmente i *banking trojan* stanno raggiungendo allarmanti livelli di sofisticazione. Nuove varianti (gli esempi più famosi sono Zeus, SpyEye e Citadel) sono costantemente rilasciate dai cybercriminali allo scopo di evadere gli antivirus e rendere più difficile per vittime e amministratori bancari individuare le frodi che essi portano a termine. Questi trojan, inoltre, sono venduti online, nei cosiddetti *underground markets*, in “pacchetti” che includono pannelli di amministrazione, builders e procedure di personalizzazione. La conseguenza più pericolosa è rappresentata dal fatto che chiunque, indipendentemente dalle abilità possedute, può acquistare un costruttore di malware e creare la sua versione personalizzata.

I banking trojan sfruttano tecniche di “hooking” per agganciarsi alle API usate dal browser e riuscire ad eseguire codice malevolo che intercetta tutti i dati che fluiscono attraverso il browser, anche quando la connessione utilizzata è criptata. Questo tipo di malware contiene anche un modulo chiamato WebInject in grado di modificare le pagine web. Questo modulo è usato dai cybercriminali per lo più per inserire nuovi campi all’interno dei forms delle pagine web con lo scopo di rubare ulteriori informazioni.

Il nostro lavoro propone un piattaforma web, Prometheus, che analizza i banking trojan sfruttando le modifiche che essi causano ai DOM delle pagine web. Prometheus è in grado, evitando lo sforzo di reversare gli eseguibili analizzati, di individuare le modifiche effettuate dai malware sui DOM e, attraverso delle tecniche forensi di ispezione della memoria, di estrarre gli obbiettivi del modulo WebInject ovvero gli URL (e le espressioni regolari che generano gli URL) delle pagine web monitorate e modificate dal malware.

L’approccio base che Prometheus sfrutta per individuare le iniezioni causate dai banking trojan consiste nell’avviare due macchine virtuali, infettare una delle due con il malware da analizzare, visitare le pagine web da monitorare da entrambe le macchine, scaricare i DOM delle pagine e confrontarli. Inoltre avviando un ulteriore macchine virtuale, infettandola e acquisendo

il dump della memoria Prometheus riesce a combinare i risultati ottenuti dal processo di confronto dei DOM con le informazioni estratte dall'analisi della memoria.

Nonostante questo approccio sia semplice ed abbia il grande vantaggio di essere indipendentemente dalla famiglia o della versione del trojan analizzato esso è allo stesso tempo molto approssimativo e può portare ad avere molti falsi positivi. Infatti ci sono tantissimi casi in cui il contenuto di una pagina web può variare legittimamente, questo potrebbe essere dovuto per esempio a script lato server, inclusioni pubblicitarie, o inclusioni di contenuti dinamici. Per questo motivo ogni analisi necessita di diverse macchine virtuali, alcune delle quali infette e altre lasciate pulite. Questo permette di eliminare le differenze legittime che si verificano tra due o più macchine pulite. Inoltre abbiamo progettato e sviluppato degli ulteriori filtri basati su euristiche per ridurre i falsi positivi.

Per concludere, Prometheus è utile agli analisti di malware perché esso riduce significativamente il bisogno di reversare manualmente i malware in questione.

Abbiamo testato e valutato i risultati prodotti dalle analisi di Prometheus su 53 diversi campioni di Zeus e 62 reali siti web mostrando che il nostro sistema è in grado di individuare correttamente le modifiche inserite dai banking trojan e di estrarre con successo gli obiettivi del modulo WebInject.

Chapter 1

Introduction

In mid 1990s, financial institutions started providing online banking services to their customers. Using a Web browser, clients could log into their bank's secure website to view statements and perform financial transactions. Since then, online banking has grown in popularity and today most financial institutions evolved the service further to reach mobile devices. In parallel to the diffusion of online banking services we witnessed an enormous rise of cybercrime. What changed in the past twenty years are the motivations of the cybercriminals. No longer searching only for notoriety and fame, cybercriminals have turned their attention to financial gain. Online frauds have become a way to earn a living. Meanwhile, banks upgraded their security measures to protect online transactions from fraud. However, attackers adapted to these countermeasures and sophisticated banking trojans, that kind of malicious software that aims at stealing banking credentials, began to emerge. While initial attacks involved simple phishing emails and keylogging trojans, which steal each keystroke the user inputs, over the years the sophistication of malware targeting financial institutions has increased dramatically and banking trojans have become one of the most prevalent categories of malware today.

Banking trojans are the main cause of billions of dollars stolen by cybercriminals. The purpose of these trojans, of which ZeuS and SpyEye are the most representative families, is stealing banking credentials and any other kind of private information. Essentially, they load code in memory and hook the network-related Windows APIs used by the browser. For this reason, they are often called "Man in the Browser" or MitB. With this technique, they can execute code that intercepts all the data going through the browser even when the connection is encrypted. This kind of malware contain also

a WebInject module that allows the cybercriminals to write scriptable procedures that modify a web page right before rendering. This module can be used to add new form fields in order to steal the target information (e.g., One-Time Passwords). Each WebInject module relies on an encrypted configuration file containing a list of targeted URLs, or regular expressions against which URLs are matched, and some HTML/JavaScript code to be injected for each URL.

The development process of banking trojans, and modern malware in general, is very refined and mature as shown in [23] [16] [8]. Banking trojans are sold on the underground markets as “toolkit” that includes builders, web-based administration panels and easy to customize configuration files. This makes everyone, even without technical skills, able to purchase a kit and start spreading a customized sample.

Antivirus software are continuously updated to counteract banking trojans but, while they offer an acceptable detection rate after having some time to issue new signatures, the detection of fresh malware samples is poor and if a cybercriminal updates the sample executable regularly he has a good chance to evade signature detection. Other works have been done regarding analysis and detection of banking trojans [7] [5] [26] but, as explained in Section 2.3, most of them are too dependent on a specific malware family or version and require a big effort to be adapted to new samples. These limitations are mainly due to the fact that the signatures are based on features such as the injection of a DLL, the hooking of some specific API or the update of a field in the Windows registry. These signatures can be bypassed if the cybercriminals release new variants of their samples hooking different API or exploiting a different type of hooking, installing browser extensions, injecting DLL into a different process.

Based on the findings of the exploratory work by Criscione et al., 2014, [10], which demonstrated the feasibility of web-page differential analysis, we propose Prometheus, a web-based platform that automatizes and facilitates the analysis of banking trojans. The proposed system works at high level of abstraction, completely independent from the implementation details of the malware. The key idea, proposed in Criscione et al., 2014, [10], is to analyze banking trojans exploiting the visible DOM modifications that they cause inside the HTML pages. Comparing DOMs downloaded in clean machines with those downloaded in infected machines we can generate signatures and we can extract the modifications reconstructing the WebInject configuration file. However, the exploratory work presented in [10] did not fully leverage the potential of differential analysis. First, it used only one infected virtual machine, whereas we show the benefits of using multiple ones.

Secondly, they did not perform a detailed analysis of the runtime behavior of the trojan, and in particular they did not analyze the time required by the sample to activate. Third, they did not explore the possibility of using memory forensics to recover, at least partially, the valuable content of the encrypted configuration file. Our system is able to perform memory forensic analyses. Inspecting the memory dumps generated from infected machines we can reconstruct part of the list of the WebInject targets, which are the URLs of the websites monitored by the trojans. This is really helpful because not all the webinject rules become visible DOM modifications. In some cases banking trojans just steal what the victim submits in web forms without injecting new fields. Moreover Prometheus, as explained in Section 3.2.2, exploits the knowledge base gained by memory forensic analyses to rank the most targeted URLs and improve the results of future analyses.

We implemented Prometheus to make it available to users and fellow researchers through a web application. Through the web interface users can easily submit samples or URLs, obtain the results of their analyses and inspect also the results of previous analyses. Indeed, we were able to implement a crawler that feeds our system continuously with new samples.

We evaluated Prometheus on a dataset of 53 distinct samples of ZeuS analyzing 62 real, live URLs of banking websites. The results show that Prometheus correctly detects the injections performed by the analyzed trojans with a low fraction of false positives (0.52%).

We evaluated Prometheus speed performances. Prometheus is able to process a single URL in about 6 seconds and the analysis of a sample processing 62 URLs require about 6 minutes. Since Prometheus has been designed to be asynchronous and to parallelize all the computations, its execution time depends only on the time required to download the DOMs of all the web page and it is able to scale directly with the available resources.

In conclusion, Prometheus gives a contribution in the analysis and detection of WebInject-based banking trojans. Prometheus complements existing antivirus by offering a helpful tool for malware analysts and bank security experts that can easily analyze samples, extracting their WebInject configuration file, or submit the URLs of their website to check if they are targeted by some previous analyzed sample.

Chapter 2

State of the art

In this chapter we start describing modern banking trojans in general showing the techniques that these trojans exploit, their characterizing features and the environment in which they are created and diffused (Section 2.1).

Then we describe in details the ZeuS crimeware toolkit as a study case (Section 2.2). We consider ZeuS as an example because it is one of the most diffused banking trojan and the majority of its peculiarities are present, in a similar way, in all the banking trojan families.

After this, in Section 2.3, we report the state of the art of banking trojans detection commenting the limitations of each presented approach.

At the end of the chapter we set the goals of our work announcing the challenges we faced during the development process (Section 2.4)

2.1 Overview of Banking Trojans

Banking Trojans are malicious programs that aim at stealing banking credentials in order to perform online financial frauds. Modern banking trojans are very complex and the process behind their development and diffusion reached a very sophisticated state. Banking Trojans live in a convoluted environment that includes development kits, web-based administration panels, builders, automated distribution networks, and easy-to-use customization procedures. The most alarming thing is that anyone can buy a malware builder from underground marketplaces and create a customized sample. New variants (the most famous examples are ZeuS and SpyEye) are constantly being introduced to avoid detection by antivirus software on the victim's PC and to make it difficult for banks and account holders to spot fraud attempts as they occur. This has been shown by Lindorfer et al. [23],

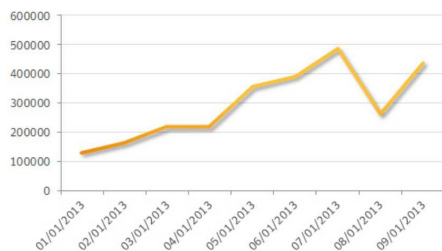


Figure 2.1: Number of computers compromised by banking trojans in 2013 (source [11]).

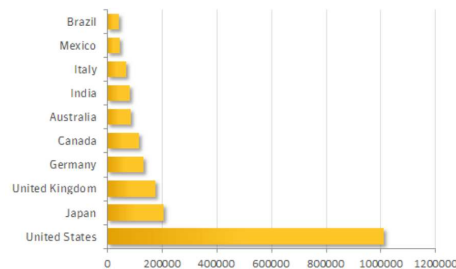


Figure 2.2: Number of computers compromised by banking trojans by country in 2013 (source [11]).

that measured how these trojans are developed and maintained by the cybercriminals.

Figure 2.1 and Figure 2.2 show the number of computers infected by banking trojans in 2013. These numbers describe significantly how much the banking trojans are diffused.

As we said the purpose of these trojans is stealing banking credentials and any other kind of private information performing MitB (Man in the Browser) attacks. This type of trojans can intercept data that the victim types into website forms and for this reason they are also called “Information Stealers” and they are the main cause of millions of dollars stolen by cybercriminals.

In October 2010, the FBI investigation called “Operation Trident Breach” [21] led up to the arrest of a group of criminals that stole a total of more than \$70 million from bank accounts. The malware used by these criminals was a version of ZeusS.

The architecture on which information stealers are based is the well-known centralized botnet architecture. There is a C&C (Command and Control) server through which the cybercriminal, also called botmaster, manages all the infected machines that belong to his botnet. All the information stolen on an infected machine are then sent to the C&C server.

During the years the functionalities of these trojans increased aiming at evading modern defense mechanisms.

2.1.1 The underground economy

One of the most worrying aspect of the banking trojans problem is that anyone independently on the skill level can perform financial frauds as the underground marketplace provides an abundance of resources like a service industry. Even those that do not have the expertise and the ability to write

an own malware, can simply purchase what they need. The trojans and services available on the underground markets are different and vary. The price depends on the features of the trojans, typically starts from 100\$ for an older leaked version to about 3000\$ for a new complete version. Furthermore, cybercriminals also offer paid support and customizations, or sell advanced configuration files that the end users can include in their custom builds. Custom WebInjects can be purchased for between 30\$ and 100\$.

Goncharov [15] studied the Russian underground economy of cybercriminals and the results of his investigation demonstrate how much this system is active and dangerous. He estimated a 2.3 billion dollars market. This [16] is alarming and represents one of the main threats to financial frauds and cybersecurity in general.

2.1.2 The fraud scheme

The fraud scheme behind the money stealing process is quite sophisticated (Figure 2.3). The first step is the malware implementation. Malware writers implement the malware toolkit and put it on sale on the underground markets. When a cybercriminal buys the toolkit and creates a customized sample (or the malware writers themselves create the executable) he starts spreading the trojan to infect victims. The infection happens mainly in three ways:

- The first way is called “drive-by download”. The user visits or is brought to visit a webpage that contains malicious code. The malicious code hosted in the visited webpage often exploit some browser’s vulnerabilities, or some vulnerabilities of its plugins or third-party extension, and downloads and executes a malware sample on the victim’s machine. The user may be brought to a malicious website in many ways. For example thanks to short links and redirection chains that, starting from a website bring the user to the final infected website. In addition most of these malware can be easily spread through social network and just clicking on them the user can become infected without being aware of what happened.
- The second way to infect users uses phishing emails. In this scenario the user receives a fake email that pretends to be from trusted institutions or websites. The email usually contains an executable attachment and the message invites the user to download and install the executable to increase protection measures, update a software or try a new “cool” service.

- Another way banking trojans infect computers is using fake tools. A fake tool is an executable that is presented to be a benign application while it contains malicious code. When a user runs a fake tool he sees the benign part of the executable that implement the actual functions the user downloaded it for but, in the background, it executes malicious code infecting the machine.

While the botmaster keeps spreading his sample infecting more and more machines he starts performing criminal actions on his bots, that means stealing money from victim's bank accounts. The stolen money are kept on bank accounts that are not in the criminals' name, but they are property of another actor, called "money mule". Money mules receive the stolen money, keep part of the sum for themselves and move the rest to the criminals' real accounts. In this way the criminals add another layer between themselves and victims making it very hard to identify the real responsible behind the fraud. Furthermore, controlling botnets that have thousands of infected machines located in different countries and continents makes even more difficult to trace botmasters.

2.1.3 Man in the Browser attacks and WebInject

Financial fraud Trojans use Man in the Browser (MitB) technique to perform attacks. This technique exploits API hooking and, as the name suggest, allows the malware to be logically executed inside the web browser and to intercept all data flowing through it.

Since the last years almost all the banking trojan families have also a module called WebInject. This module is able to manipulate and modify data transmitted between an HTTP(S) server and the browser. Once the victim is infected the WebInject module is placed between the browser's rendering engine and the API functions that allow to send and receive HTTP(S) data. In this way this module is effective even in case of an HTTPS connection, because it can access data after decryption. Exploiting this module a cybercriminal can inject HTML code to add further fields in forms and steal the target information. Each WebInject module has an encrypted configuration file. It contains a list of webinject rules composed by the targeted URL, or the regular expression from which URLs are generated, and the HTML/JavaScript code to be injected.

Recently a new variant of ZeuS was found in the wild. The particularity of this variant is that it uses images as a decoy to retrieve its configuration file. It exploits steganography techniques to hide the encrypted configuration file inside apparently normal jpg images [27].

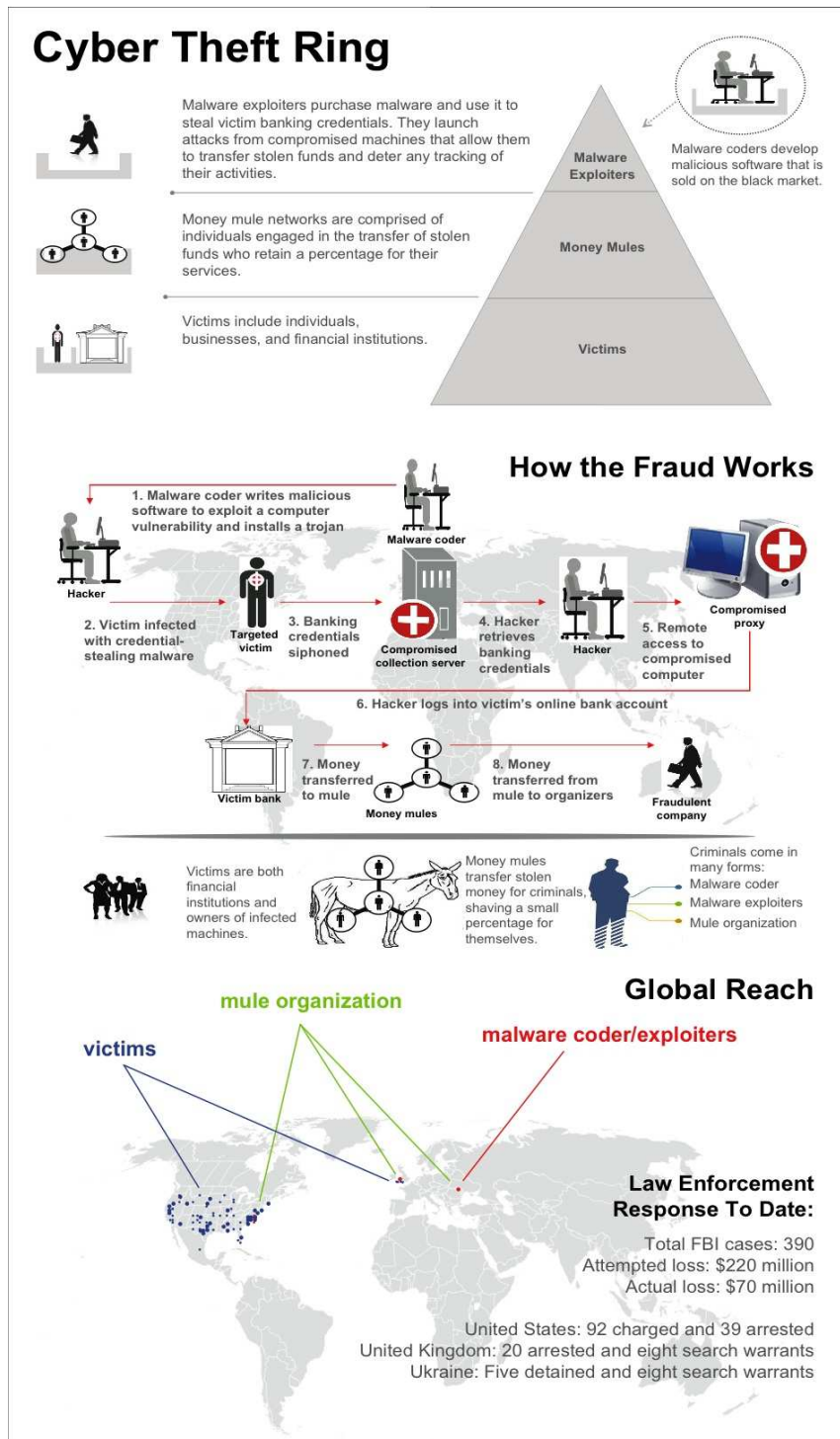


Figure 2.3: The fraud scheme (source [13]).



Figure 2.4: Example of a real injection on the login form in a web page of online-offshore.lloydstsb.com.

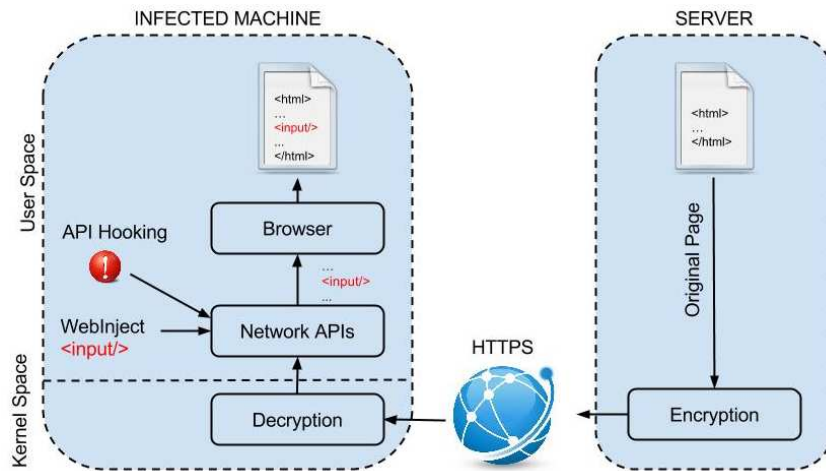


Figure 2.5: Scheme of the WebInject Hooking mechanism.

Man in the Browser attack scheme reminds the phishing attacks but in this case there is no fake website and the manipulation happens on the victim's machine at the presentation layer. The attacker does not need to host and maintain a fake website. This guarantee to the attacker much more effectiveness and flexibility than phishing.

In Figure 2.4 it is shown a case in which the WebInject module injected additional input fields in the login form of a banking website.

The goal of the attacker is to inject new fields in forms, in order to steal the target information, but without altering the main aspect of the web page so that the victim does not suspect that a fraud is happening. For this reason the webinject rules often target only a small portion of a web page. The syntax to define WebInjects follows simple rules, as shown in Section 2.2.2.

2.1.4 Hooking mechanism

Information stealers use userland rootkits techniques to intercept and manipulate web traffic. The malware injects code into the web browser process on start up and installs code hooks for API functions of the system libraries loaded by the process. An example of API functions hooked by the Trojan Zeus version 2 inside the web browser Microsoft Internet Explorer is shown in Table 2.1.

By hooking high-level API communication functions in user-mode code, the trojan can more conveniently intercept data than traditional kernel-rootkit with keyloggers and is able to intercept web data after it gets decrypted and before it gets encrypted again (Figure 2.5).

There are different ways to hook API functions in a Windows operating system: inline hooks, import address table (IAT) hooks, export address table (EAT) hooks and hook techniques to manipulate the windows loader mechanism.

Inline hooks

This is the most common method and it is based on overwriting code bytes of an API function with a jump instruction that point to a code section controlled by the trojan. Typically the first five bytes are overwritten (Bruescher et al. [7]).

Import address table hooks

This technique modifies the import address table (IAT), which is used by processes to obtain the position of functions or variables that are imported from dynamically loaded libraries. IAT hooks overwrite the original destination of an imported API function and point it to code controlled by the malware (Bruescher et al. [7]).

Export address table hooks

The export address table (EAT) of a module contains the addresses of all API functions exported by that module. As for IAT hooks this technique overwrites the corresponding function address in the table (Bruescher et al. [7]).

LIBRARY	API
wininet.dll	HttpQueryInfo
wininet.dll	HttpSendRequest
wininet.dll	HttpSendRequestEx
wininet.dll	HttpSendRequestExW
wininet.dll	HttpSendRequestW
wininet.dll	InternetCloseHandle
wininet.dll	InternetQueryDataAvailable
wininet.dll	InternetReadFile
wininet.dll	InternetReadFileEx
ntdll.dll	NtCreateThread
ntdll.dll	NtCreateUserProcess
ntdll.dll	LdrLoadDll
ws2_32.dll	closesocket
ws2_32.dll	send
ws2_32.dll	WSASend
kernel32.dll	GetFileAttributesExW
user32.dll	GetCursorPos
user32.dll	OpenInputDesktop
user32.dll	SwitchDesktop
user32.dll	DefWindowProc
user32.dll	DefDlgProc
user32.dll	DefFrameProc
user32.dll	DefMDIChildProc
user32.dll	CallWindowProc
user32.dll	RegisterClass
user32.dll	BeginPaint
user32.dll	EndPaint
user32.dll	GetDCEx
user32.dll	GetDC
user32.dll	GetWindowDC
user32.dll	GetUpdateRect
user32.dll	GetUpdateRgn
user32.dll	GetMessagePos
user32.dll	SetCursorPos
user32.dll	SetCapture
user32.dll	ReleaseCapture
user32.dll	GetCapture
user32.dll	GetMessage
user32.dll	PeekMessage
user32.dll	GetCapture
user32.dll	TranslateMessage
user32.dll	GetClipboardData
nspr4.dll	PR_OpenTCPSocket
nspr4.dll	PR_Close
nspr4.dll	PR_Read
nspr4.dll	PR_Write
crypt32.dll	PFXImportCertStore

Table 2.1: APIs hooked by ZeuS inside the web browser Microsoft Internet Explorer

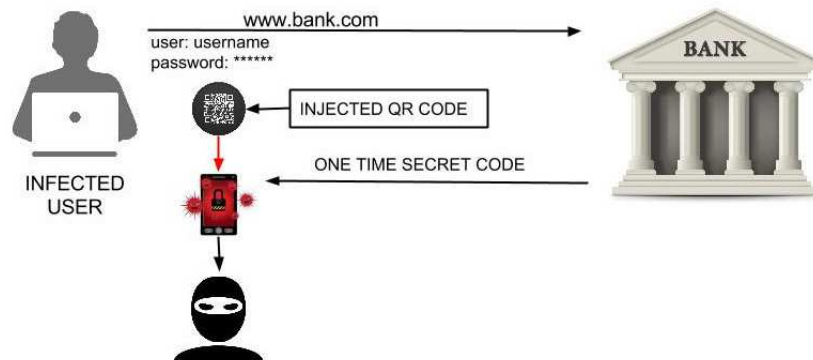


Figure 2.6: Scheme of the common attacks that involve a malicious mobile component to steal OTP codes from mobile phones.

Other hooking techniques

Other examples of rarely used hooking techniques are exploited when an infected parent creates a child process. It is also possible to use several combinations of any of the above mentioned hooking methods.

2.1.5 The Automatic Transfer System

Another dangerous feature that some banking trojans present is the so called Automatic Transfer System (ATS). The ATS emerged since 2012 and it consists in a more sophisticate WebInject module. In fact it is able to automatically perform money transaction or modify user's transaction changing the amount and the recipient on the fly. Unlike WebInject files that show pop-ups or insert form fields to steal victims' credentials, ATS remains invisible. It does not display any visible modification but performs several tasks such as checking account balances and conducting wire transfers using the victims' credentials without alerting them and hiding traces of its presence.

2.1.6 Man in the Mobile

In order to fight the credential stealing problem most of the financial institutions and banks proposed two-factor authentication. This kind of authentication requires users to provide a one time password (OTP) in addition to the usual username password pair. This one time password is sent to the user's mobile phone by SMS and it is valid once and just for a short limited amount of time. In the case an attacker steal the user's credential he cannot complete transaction without the OTP. Even if an attacker gets an OTP

from an infected machine after that the user has submitted it in a form it will be useless because expired.

Since this authentication mechanism is by now quite common, in the last two years most of the banking trojans toolkit have been diffused including a mobile component. This mobile component works in tandem with the PC versions and can access all the information in the user's phone, including SMS, and send it to its C&C server. This attack scheme is called Man in the Mobile (MitMo). The usual scheme to perform this attacks defeating the two-factor authentication is shown in Figure 2.6 and follows these steps:

- The first step is to infect the victim's PC according to the process described above.
- Once the victim's PC is infected, when the victim visits his online banking website the trojan steals his credentials and inserts a message in the web page that invites the user to download and install a new mobile application to be able to access his account even from mobile phone. This step is usually performed inserting in the web page a QR code that points to the malicious application's download.
- When the victim downloads and installs the mobile malware his phone is compromised. The mobile malware can now intercept all the SMS, silently avoid the system notification and remove them after they have been sent to the C&C. In this way the attacker manages to steal the OTP hiding the SMS arrival to the victim.

2.1.7 A new Target: Bitcoin

During the last years new versions of banking trojans targeted Bitcoin wallets [9]. The interest in Bitcoin has grown substantially in 2013, particularly since the exchange rate for one Bitcoin rose to over 1000\$ in November 2013. Over the last few years, malware authors have developed trojans that compromise Bitcoin wallets stealing the local stored files of the offline wallet and/or passwords to access online Bitcoins.

We expect a further growing interest from attackers in this digital currency, especially because Bitcoin's value is currently increasing.

2.2 The ZeuS Crimeware Toolkit

ZeuS, also known as Zbot, is the most diffused family of banking trojans. It was detected for the first time in 2007 and during the last years it spreaded massively and captured media attention especially when police arrested several botmasters that had stolen millions of dollars from bank account.

In January 2013, a 24-years old Algerian man, Hamza Bendelladj, known as “bx1” was arrested for financial frauds running a ZeuS botnet [22]. Bendelladj admitted that with just one transaction he could earn 10 to 20 million dollars.

Since the ZeuS 2.0.8.9 source code¹ was stolen and leaked to the underground community in May 2011, nearly every banking trojan contains ZeuS features. The relative maturity and broad success of ZeuS has provided a model in the weaponization and development of other families of banking trojans.

Current versions of ZeuS can infect almost all Microsoft’s operating systems targeting principally Internet Explorer and Mozilla Firefox browsers. However recent variants of ZeuS and other trojans are able to perform web injections also in Google Chrome and Opera.

In December 2013 a 64-bit version of ZeuS was found in the wild [25]. This demonstrate one more time that the malware authors constantly maintain and update their malware adapting them to the evolving technologies.

The ZeuS toolkit includes also a mobile component, Zitmo (ZeuS in the mobile), which targets Android mobile phones and acts as explained in Section 2.1.6.

When a victim machine gets infected ZeuS performs a sequence of actions in order to take control of the machine.

1. ZeuS executable first of all creates a directory and copies itself in there. Both the directory and the file name are generated randomly.
2. It then insert keys in the system registry so winlogon.exe spawns the process at startup time.
3. It injects malicious code into other process (e.g., winlogon.exe, explorer.exe, svchost.exe) and starts new thread executing its malicious code, so the main process can terminate.
4. It creates a folder to store configuration files and stolen data.

¹<https://bitbucket.org/davaeron/zeus/>



Figure 2.7: Login page on the website of SBTOnline advises to use the virtual keyboard.

5. It injects new malicious code inside the processes responsible for network communications and hooks Internet-related APIs.
6. It steals saved credential (e.g., cookies, certificates, browser's password)
7. It downloads the configuration file from the C&C server.

2.2.1 Keylogging, formgrabbing, screenshotting and clickgrabbing

The basic feature Zeus presented in his first version was keylogging. This technique consists in storing in a file each keystroke the user inputs and sending the file containing all the recorded keystroke to the botmaster. Even though this technique is very effective, there are situations in which input data can not be intercepted, for example when the user copies and pastes data from a file or when the user select an option from a menu that does not require any data typed on keyboard.

Another feature is the form-grabbing. The “FormGrabber” is a module that can be configured to intercept and steal data that the user submits into the form fields of websites.

As a defense mechanism against form-grabbing some banks provided an on-screen keyboard written in JavaScript in the login page (Figure 2.7). In this way during the login phase the user does not have to type his credential on the physical keyboard but he needs just to click on the letters of the virtual keyboard that is shown on the bank's web page.

The malware authors answered to this mechanism implementing two more features: screenshotting and clickgrabbing. The first one allows the attacker to take screenshots of the victim's PC during the login phase. The second one is able to track and record the mouse position when the user clicks.

2.2.2 Components

The ZeuS toolkit is released and sold in the underground markets with all its components and with a user manual that explains how to set up each component. Cybercriminals also offer paid support for update and personal customization. The main components of the toolkit are the following:

- **The builder** is the program that generate the customized sample executable. It is written in C++ and it has a GUI that allows to generate the executable with just a couple of clicks. During the generation of the sample, the botmaster set an encryption key and other parameters that are hardcoded and obfuscated in the bot executable. The builder encrypts also the configuration file that is uploaded to the C&C server and distributed to the bots.
- **The configuration file** contains the information needed to communicate with the C&C server, the URL where to download the encrypted configuration file, the key to encrypt and decrypt the configuration file and the traffic to and from the server.
- **The `webinject.txt` file** contains the rules to perform injections as explained in Section 2.1.3. Each rule is defined by:
 - *set_url*; it specifies the URL the webinject refers to. It can be a regular expression that generates more URL of the same host and it can also contain special characters and parameters that specify the conditions under which the injection is enabled;
 - *data_before*; it specifies the hooking point inside the page where the HTML code is injected. As the previous field also this one allows special characters and regular expressions in order to increase chances for a successful injection;
 - *data_inject*; it specifies the actual code to inject. It can contain scripting code;
 - *data_after*; it specifies in another way a hooking point;
 - *data_end*; it is used to close each code part.

Listing 2.1 shows a portion of a real leaked ZeuS `webinject.txt`.

```
set_url https://www.bbvanetoffice.com/local_bdno/  
login_bbvanetoffice.html GP  
data_before  
name="password"*<tr>  
data_end
```

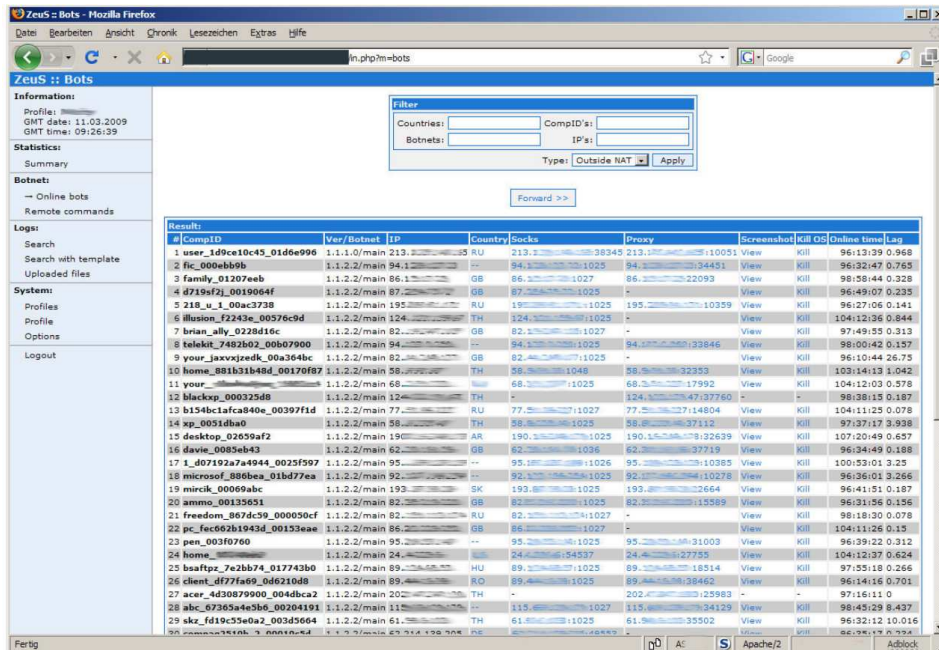


Figure 2.8: A screenshot of the control panel of ZeuS (source: abuse.ch).

```

data_inject
</tr>
<tr><td height="20" class="c"><span class="txtbc">Firma
: </span></td></tr>
<tr><td height="20" class="c"><input type="password" name
="ESpass" size="16" maxlength="9" onKeyUp="javascript
: cuenta('pwd');" tabIndex="2"></td></tr>
<tr>
data_end
data_after
data_end
data_before
name="btnEntrar"
data_end
data_inject
OnClick="javascript: if (document.forms[0].ESpass.value.
length < 3) {
alert('Debe introducir la Firma');return false;
}"
data_end
data_after
data_end
set_url https://www.wellsfargo.com/* G
data_before
<span class="mozcloak"><input type="password"*</span>
data_end
data_inject
<br><strong><label for="atmpin">ATM PIN</label>:</strong>

```

```
>&nbsp;<br />
<span class="mozcloak"><input type="password" accesskey="
A" id="atmpin" name="USpass" size="13" maxlength="14"
style="width:147px" tabindex="2" /></span>
data_end
data_after
data_end
```

Listing 2.1: Portion of a real leaked ZeuS webinject.txt

- **The C&C control panel** is the management component, written in PHP, installed on the C&C (Figure 2.8). It is the centralized part that manages and communicates with all the bots. It stores the stolen data received by the bots and provides an interface through which the botmaster can handle his bots. It allows to get statistics about the botnet, such as connected bots, information about each bot and it provides a mechanism that allows the botmaster to easily write scripts and send them to the target bots. They allow to perform any functionality a bot master may think of: rebooting or shutting down the remote OS, stealing files, updating the bot and its configuration file, enabling or disabling some of its features. From the server it is also possible to enable or disable BackConnect, a feature that allows to use an infected machine as a SOCKS proxy, circumventing firewalls and NAT restrictions.

2.3 Banking Trojan detection

The current number of different existing banking trojan samples is huge. New banking trojans families and new versions of already existing families are continuously released and, as we have seen above, each specific trojan can be customized and obfuscated generating millions of different executables. In addition also the custom configuration files are encrypted and embedded in the final executable. For this reasons analyzing manually all the samples is a mission impossible even for a great number of malware analysts and automatic mechanisms to extract configuration files from sample or, more simply, to detect the activity of an infected machine are needed.

2.3.1 Antivirus detection

Antivirus software are continuously fighting information stealing malware. The main problem is that they can offer an acceptable detection rate only after having some time to issue new signatures while they fail in detecting

fresh malware samples. This allows cybercriminals to evade signature detection updating regularly their sample executable. Furthermore, as explained in detail in Binsalleeh et al. [5], ZeuS executables, like most of the modern malware executables, are packed and obfuscated. Each sample executes different deobfuscation routines when it is executed. This makes really hard for antivirus software to achieve an acceptable detection rate on the basis of static signatures also because the packers and crypters used by the attackers are vary and constantly updated.

For instance, as of Aug 5, 2014, according to ZeuS Tracker² there are 7,852 distinct variants that are yet to be included to the Malware Hash Registry database³. This high number of variants results in a low detection rate overall (39.83% as of Aug 5, 2014).

Another limitation of Antivirus software is their inconsistent malware characterization as shown in [4]. The reason for this is that antivirus vendors are interested in reliable detection but only partly in naming the detected threat correctly. In most cases the vendor just wants to provide some name to the user. A more reliable classification can be achieved using dynamic signatures and behavioral information on the interaction between an application and the operating system.

2.3.2 Reverse Engineering

Reverse engineering is one of the oldest and most effective technique to analyze malware samples, even if they are obfuscated and embed encrypted configuration files. However reverse engineering is too time-consuming and requires a big effort. Sometimes malware present some vulnerabilities (e.g., SQL injection, weak cryptography) that can be leveraged to speedup the reverse engineering process or to extract the encrypted information hard-coded in the executable. This is the case of Ricciardi et al. [26] that found a vulnerability in the ZeuS internal cryptography scheme and exploited it performing a chosen-plaintext attack to recover the key used in the communication between ZeuS and its C&C. The attack is based on the fact that ZeuS malware does not update the RC4 initialization vector, exposing its communications to key reuse attacks. By executing a ZeuS sample in a controlled environment it is possible to perform a chosen-plaintext attack controlling, on the infected machine, the cookies and other information (e.g., computer hostname, user credentials) that will be sent after the infection phase.

²<https://zeustracker.abuse.ch/statistic.php>.

³<http://www.team-cymru.org/Services/MHR/>

Binsalleeh et al. [5] performed a complete reverse engineering of the ZeuS crimeware toolkit v.1.2.4.2 explaining the functionalities of each its component and focusing particular attention on the deobfuscation routines done by the malware when it is executed.

In [18] a reverse engineering of both SpyEye and ZeuS is presented. It gives a detailed overview of the hooking and the process injection mechanism of both the malware providing a comparison between the two families.

All these approaches based on a the reverse engineering of malware binaries are effective and useful to understand the activity of malware, extract the main features that can be used in the detection systems or to identify vulnerabilities. However the principal drawback of this modus operandi is the lack of generality. Often the results obtained by reverse engineering are valid only for the specific malware family and/or version and require a big effort to be adapted to different releases.

2.3.3 Other related work

Other works have been done regarding analysis and detection of banking trojans and WebInject. Bruescher et al. [7] proposed a different approach to identify WebInject based information stealers. The idea of the authors is similar to the usual rootkit detection, that means the detection of API hooks in common libraries. In particular the signatures are generated looking at hooks in browser and Internet related APIs. The objective of the proposed system is to search for code injection or modification inside Windows Internet related libraries. Furthermore, since the list of API function hooks is different for most of the trojan families this approach can be used also for classification. To prevent false positives detection of legitimate software they inspect the destination of each hook and check if the pointed module is trusted and correctly signed.

The limitation of this detection approach is the strong dependence on the version of the trojan, on the operating system and on the hooked browser. Different trojans or future releases could change the list of API functions to hook or target another browser that use different libraries.

2.4 Challenges and Goals

In Section 2.3 we discussed the limitations of the current techniques to detect and extract encrypted information from banking trojans. The objective of our work is to propose a different approach based on a common feature present in all the banking trojans, the WebInject.

Our work takes inspiration from the previous work of Criscione et al. [10]. We started from their results, which show that the adopted detection approach is sound, and we extended and engineered their work into a more complete prototype system, Prometheus.

The objectives of this work are the following. First, we want to develop a platform for analyzing banking trojans based on the idea proposed in Criscione et al., 2014, [10]. Differently from previous work in the field, we want to analyze banking trojans at high level of abstraction, completely independent on their implementation details. The key idea, proposed in Criscione et al., 2014, [10], is to analyze banking trojans exploiting the visible DOM modifications that they cause in the HTML pages. Comparing DOMs downloaded in clean machines with those downloaded in infected machines we want to generate signatures and extract the modifications reconstructing the WebInject configuration file.

Second, we want to guarantee low false positives. Comparing DOMs downloaded from multiple clean machines, we want to discard all the legitimate differences of a web-page due, for example, to server-side scripts and advertisement inclusions. This is not an easy task as nowadays most of the web pages are formed almost entirely by JavaScript and highly variable contents.

Third, we want to reduce the number of VMs needed in order to achieve the same level of precision reached by Zarathustra [10]. This implies a considerable improvement of the performance. However, since guaranteeing low false positives requires an high number of VMs, the challenge towards this goal is to design and implement a new set of heuristics to reduce the false positive rate.

Further, we want to combine the web-page differential analysis with memory forensic analysis in order to recover, at least partially, the valuable content of the encrypted configuration file and to exploit it to check and validate the results of the web-page differential analysis. We do not want to rely on any implementation details of a specific banking trojans' family. We want to provide an automated extraction mechanism that is as general as possible. Moreover, since the information we look for are not placed in fixed memory locations, we need to develop an automated mechanism able to scan the entire memory and extract only the correct information without generating false positives.

Another goal is to guarantee good performance, that means a low execution time required to analyze malware. The critical point that mostly affects the performance of the proposed system is the high number of VMs.

However, as we said, we need a lot of VMs in order to correctly discard legitimate differences. Therefore, a further challenge of this work consists in evaluating the trade-off between performance and false positive rate.

Finally, we want to implement our system to make it available to users and fellow researchers through a web application.

Chapter 3

Prometheus: the approach

In this chapter we introduce the approach on which Prometheus is based.

Section 3.1 provides a general view of the approach, Section 3.2 shows the two kinds of analysis that we conduct, then Sections 3.3.1, 3.3.2 and 3.3.3 provide a deeper look into the three main phases of the approach.

We reserve more architectural and implementation details for the next chapter.

3.1 Proposed Approach

For one of the main features of our work we take inspiration from Zarathustra [10], a tool developed at the NECSTLab of Politecnico di Milano. Zarathustra was a proof of concept to detect the behavior of any “WebInject-based information stealer” (WBIS) by looking at the evidence of Webinjects in the targeted websites. This approach do not take in consideration any implementation details of the information stealer analyzed. For this reason from now on we will talk of WBIS in their most general interpretation without taking into account any implementation detail. A WBIS is any kind of malware which employs a mechanism in order to change the content of a web page injecting some extra contents in the (decrypted) data that transits between the network layer and the rendering engine of a browser.

In Prometheus we completely reimplemented Zarathustra integrating its analysis results in a comprehensive web service. We completely rethought Zarathustra heuristics at an high level of abstraction. As we will show in Chapter 5 this approach leads us in drastically reducing the number of VMs needed in order to achieve the same level of precision reached by Zarathustra. This result is due to the nature of our heuristics which belong

more from a formal characterization of a webinjection than from a noise filtering process. In addition to [10] we also integrated a memory forensic inspection mechanism in order to retrieve from the memory dump of an infected VM some useful information. These information allows Prometheus to rank the most targeted URLs and improve the results of future analyses.

The base approach of the webinjections detection process consists in starting two virtual machines, then infecting one of them with the malware, downloading a page from a targeted site on both the machines and then comparing the two downloaded DOMs. On one hand, this approach is quite simple and has the main advantage to be easily scalable and automatable, on the other hand is very naive and could lead to high values of false positive rate. In fact there are a lot of cases in which the content of a web page may vary legitimately, for example this could be due to server-side scripts, advertisement inclusion that may change or include content that varies dynamically. For this reason we run the analyses on multiple machines in order to be able to discard the legitimate differences that occur between two or more clean machines. Moreover, to reduce false positive rate, we designed and implemented four heuristic-based filters.

As we said above, we combine this high level approach with a memory forensic inspection that allows us to extract from memory dumps, retrieved from an infected VM, the regular expressions, contained in the encrypted WebInject configuration file, that describe the WebInject targets. At the end of the analyses we check if any of the URLs stored in the database match any of the extracted regular expressions. The matches allow us to obtain information about the most targeted URLs. This information is used, in future analyses, to select and process the URLs that are most likely to be targeted. Moreover, relying on some implementation details of the most common WBIS we are able to extract from infected memory dumps also other useful data like the cryptographical keys used to encrypt the WebInject configuration file and the connection to the C&C.

These approaches are combined in a web service platform in which users can easily upload malware's sample or submit a certain URL and dynamically get the results of the analysis. The web interface also allows to navigate through old analysis looking at their results and inspecting the dumped DOMs.

3.2 Prometheus Analyses

Prometheus can conduct two kind of analyses: URL analysis and sample analysis.

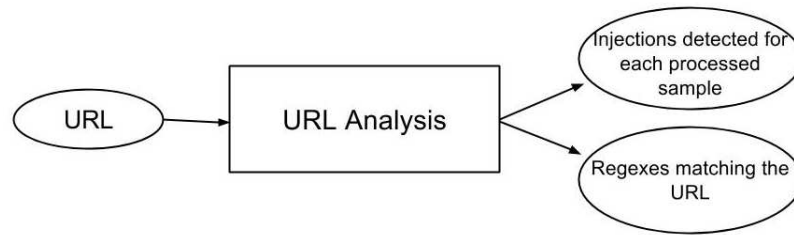


Figure 3.1: I/O Scheme of the URL analysis. It takes in input the URL and returns to the user the list of injections found for each processed sample and the regular expressions the URL matches.

3.2.1 URL analysis

The objective of the URL analysis is processing the submitted URL infecting the VMs with the last submitted samples, checking if the DOM of the webpage has been modified by some of them and returning to the user the list of injections found for each selected sample. The number of samples to be considered for a URL analysis is set in the configuration file but it can also be chosen by users during the submission. In particular Prometheus behaves as follow. When a new URL is submitted Prometheus checks if it matches any of the previously extracted regular expressions and selects for the analysis those samples from which the regular expressions have been extracted. At the end of the analysis Prometheus returns to the user the list of injections found for each selected sample and the regular expressions the URL matches (Figure 3.1).

This kind of analysis is thought to be useful for website administrators which would like to know if their site is targeted by a man in the browser attack. From the results of the analysis they can know if there is some active sample effectively targeting their page and what are the injections. Furthermore looking at the list of regular expressions they can have insights on which other pages can be targeted by the samples.

Moreover, every submitted URL is permanently added to our URLs database and it can be selected to be processed for sample analyses on the base of the ranking explained in the next Section.

3.2.2 Sample analysis

The sample analysis runs the submitted sample while visiting a list of URLs. Figure 3.2 shows the basic I/O scheme of the sample analysis. Once the user submits a sample Prometheus starts the analysis and interactively returns the results to user through the web interface. On the result page the user can

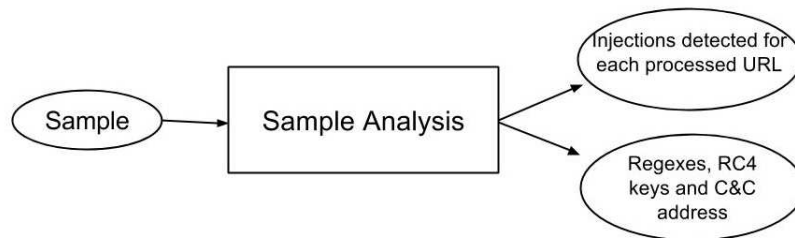


Figure 3.2: I/O Scheme of the sample analysis. It takes in input the sample and returns to the user the list of injections found for each processed URL and the information extracted through the memory forensic analysis (regular expressions, RC4 cryptographical keys, C&C address).

retrieve the injections that the analyzed sample produced on every analyzed page. The number of URLs to be processed in a sample analysis is set in the configuration file.

Furthermore, together with the webinjections detection process, Prometheus performs a memory forensic inspection in order to search and extract from infected memory dumps the WebInject targets and the cryptographical keys. The extracted WebInject targets are then used to check if any of the processed URLs match any of them. This allows to validate the results of the web-page differential analysis and to obtain information about the most targeted URLs. More precisely, Prometheus exploits the knowledge base created from past memory analyses to select, for the future sample analyses, the URLs that are most likely to be targeted. URL's likelihood is taken from the previously analyzed samples; in fact the more the system found, in the memory of infected VMs, regular expressions that match with a certain URL the more that URL is likely to be targeted also by other samples. In this way we generate a ranking of the URLs that is constantly updated after each analysis.

This kind of analysis is thought to be mainly used by malware analysts which find malware in the wild and are interested in retrieving as much information as possible. With our system analysts are able to detect WebInject based information stealers and to retrieve an almost complete summary of the WebInject configuration file of the sample they submitted, reducing to zero the whole effort in reversing and decrypting the malware.

3.3 System Overview

Both the sample and the URL analysis process can be described in three phases: Data Collection, Data Processing and Results. The main difference

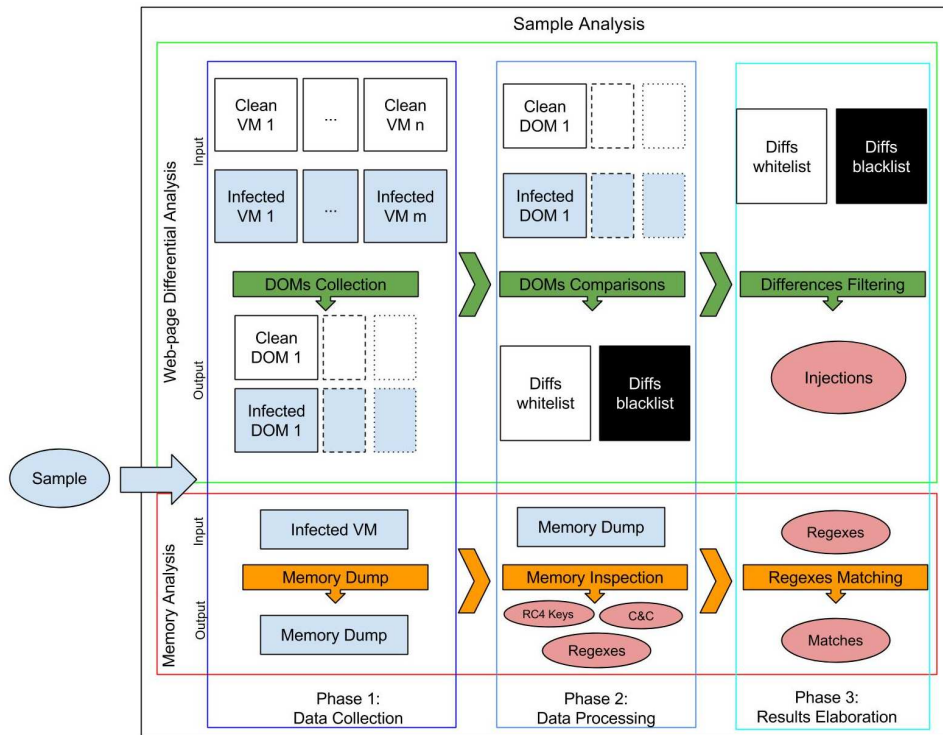


Figure 3.3: Overview of the sample analysis. The web-page differential analysis and the memory analysis are performed in parallel. Both the analyses follows the three phases described in Section 3.3. The overview of the URL analysis is the same with the exception that no memory analysis is performed.

is represented by the memory forensic analysis that is performed only in the case of a sample analysis. Figure 3.3 shows the overview of the sample analysis with all its phases. The following sections provide an high-level explanation of the three Prometheus phases.

3.3.1 Phase 1: Data Collection

During the data collection phase our system retrieves two kinds of data: the DOMs and the memory dump. For the DOMs retrieval it starts a certain number of VMs, half of which are infected with the sample to be analyzed, and visits the list of selected URLs on each VM. For the memory dumps, Prometheus starts another VM, opens the browser and dumps its memory.

DOMs Collection

In this phase, Prometheus processes a list of URLs to analyze and visits each of them on several clean and infected VMs, specified by the user during

the submission. For each visited URL Prometheus dumps its DOM and stores it. Since there are a lot of cases in which the content of a web page may vary legitimately, this could be due, for example, to server-side scripts or advertisement inclusions that may change or include content that vary dynamically, the DOMs collection phase should be performed on multiple machines some of which have to be infected with the submitted malware and the others left clean. This allows, during the results elaboration, the elimination of legitimate differences that occur between two or more clean VMs. When this phase terminates, the doms directory contains all the dumped DOMs and the database has an entry for each DOMs.

Memory Dump

In this phase Prometheus executes a further VM, infects it with the submitted sample and dumps its memory. We use a separate VM for the memory analysis because visiting many URLs, as the other machines do, can generate false positives as the URLs are stored in memory and they could be wrongly detected as WebInject targets. To get the WebInject targets we are looking for, Prometheus waits for the sleep time and opens the browser inside the VM in order to let the malware loads the WebInject targets in memory. This is fundamental because the regular expressions that specify the sample's targets are allocated in the process address space of the browser and if it is not executed we will not find any of them. At the end of this phase we get the memory dump stored as a file.

3.3.2 Phase 2: Data Processing

During the second phase of our approach Prometheus compares the DOMs, referring to the same URL, downloaded by different VMs and extracts the WebInject targets from the memory dump obtained in the previous phase.

DOMs Comparison

After the collection phase the dumped DOMs have to be compared. What is important to highlight is that Prometheus has been designed to parallelize the computations and to execute them as soon as the required data are available. This means that the DOMs comparison phase starts as soon as the first DOMs are downloaded and so it is partially overlapped with the DOMs collection phase.

In this phase we consider “clean DOMs” those downloaded by a clean VM and “infected DOMs” those downloaded by an infected VM. The comparis-

ons concerning each URL are done considering one clean DOM as reference and comparing all the others with that one. All the differences found are then appended into two lists (black and white) depending if the compared DOM is clean or infected. Every difference is composed by three elements:

- Type: The nature of the difference (deletion, insertion, modification etc.).
- XPath: The xml path to the node which is affected by the difference.
- Content: The value of the difference (e.g., in the case of a node insertion the content is the HTML node inserted with all his attributes).

At the end of this phase for each processed URL the blacklist and the whitelist contain all the differences output by the comparisons.

Memory inspection

When the infected memory dump is generated it is inspected in order to extract the WebInject targets and the RC4 cryptographical keys. The inspection is performed thanks to some forensic tools (Volatility and YARA) properly extended and customized. The approach is based on the definition of some regular expressions to scan the memory dump. Doing so we are able to recover a complete list of the WebInject targets, the RC4 cryptographical keys used for the encryption of the WebInject configuration file and of the connection to the C&C server and the address of the C&C server.

3.3.3 Phase 3: Results Elaboration

In the third phase of our approach Prometheus filters out the legitimate differences employing four heuristics. Moreover, it exploits the information extracted by memory analysis in order to update the URLs ranking.

Differences Filtering

The two lists of differences produced in the previous phase are filtered according to some heuristics in order to eliminate the legitimate differences, toward reducing false positives. After the filtering process the differences remaining in the blacklist are those considered as malicious webinjections. We designed and implemented the following four heuristic-based filters:

1. In certain pages there may be some nodes that change very often their contents (calendar, clock, advertisement and so on). This kind

of nodes generate a lot of differences that refer to the same node but with different content, and in particular they are present in both the blacklists and the whitelist. For this reason we remove from the blacklist all the differences that have the same Type and the same XPath of a difference belonging to the whitelist. For example in the case of a web-page containing an advertisement that dynamically change its message, thanks to this filter we are able to discard the differences that it causes.

2. In other pages may happen that some nodes with a fixed content (for example, and mostly, Javascript) sometimes are omitted or located in different places inside the webpage. We remove from the blacklist all the differences that have the same Type, the same last node in the XPath and the same content of a difference in the whitelist (in particular this heuristics is adopted as preparation for the application of the next one). For example in the case of a web-page containing an advertisement that presents a fixed content but that is loaded dynamically in different positions of the page, thanks to this filter we are able to discard the differences that it causes.
3. Since malware authors are interested in inserting new elements, in order to steal data from the victims, we are interested in looking only at those differences that are insertion or modification of something. For this reason all the differences that imply other mechanisms (like for example deletion of nodes) are filtered out. Moreover we filter out all the differences that regard harmless attribute (value, class, width, height, sizset, title, alt).
4. A typical injection has the following characteristics: it is present mostly in all the infected machines but it is never present on the DOMs downloaded by clean machines. An injection will always inject the same content and will inject it in the same node, even if the node changes his XPath in the DOMs downloaded by different VMs. So a typical webinjection will always refer to the same last node of an XPath that may vary somehow. Hence the idea is to filter out all the differences that are not present (with the same Type, the same last node of the XPath and the same content) in all the injected DOMs. However, sometimes may happen that on a certain VM the malware does not activate itself because malware authors take counter measures to prevent dynamic analysis, so, if the malware activation time is randomized, it may happen that a sample manifests its behavior

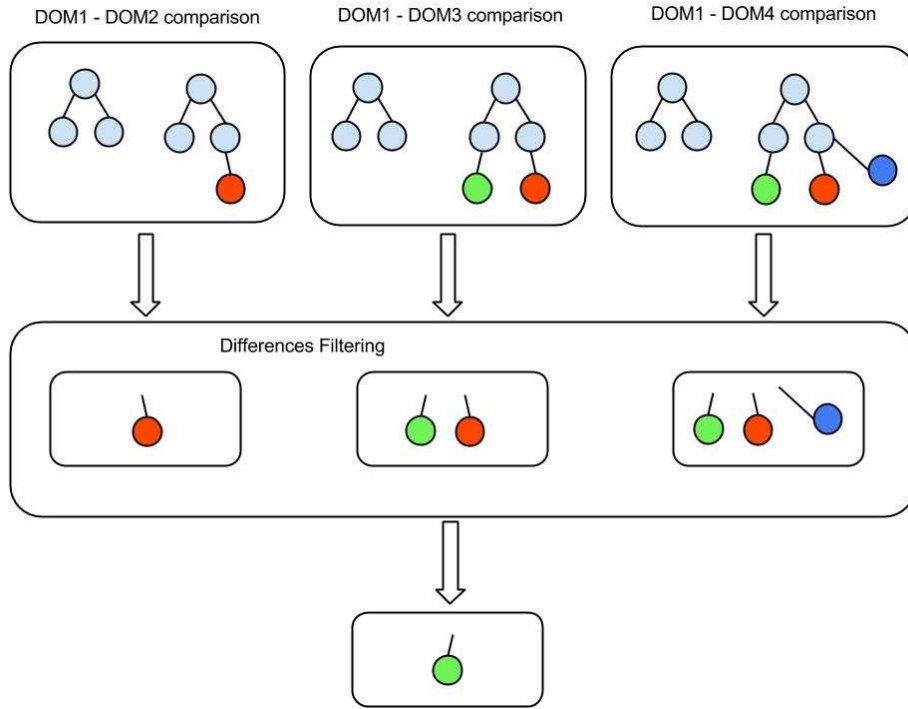


Figure 3.4: Graphical explanation of differences comparison and filtering: DOM1 is the Reference DOM for a given URL; DOM2 is a DOM dumped from a clean VM for the same URL; DOM3 and DOM4 are two other DOMs dumped from infected VMs again for the same URL. The differences produced by the comparisons shown are then filtered. Being present in another clean DOM the red difference is discarded. Also the blue difference is discarded because it is not present in all the infected DOMs

only in a subset of the VMs. For this reason we consider a threshold ε (configurable at submission time), and we filter out all the differences that are not present in at least $\varepsilon\%$ of the infected DOMs. While this approach could theoretically bring to false negatives, this never happened during the evaluation (Section 5.4.2).

Regular Expression Matching

Once Prometheus has extracted all the useful information from the memory dump it stores them in the database. Furthermore for every regular expression found Prometheus checks if it has some match with the URLs present in our database, that are the URLs used for the sample analyses. Finally, the full list of matches is stored in the database.

The screenshot shows the 'abuse.ch ZeuS Tracker' interface. At the top, there is a navigation bar with links: Home, FAQ, ZeuS Blocklist, ZeuS Tracker, Submit C&C, Removals, ZTDNS, Statistic, RSS Feeds, Contact, and Links. Below this, the page title is 'ZeuS Tracker :: Browse ZeuS BinaryURLs'. A warning box states: 'Warning: The files which you can download here may harm your computer!'. There are links to 'Set a filter for the list below: online | offline | all' and a 'Subscribe this list via RSS feed' button. The main content is a table with the following data:

Date added	ZeuS BinaryURL	Status	MD5 Hash	Filesize	Virustotal	Anubis	File download
2014-08-14	69.195.124.111/~suspene1/adminpane1	active	3caaab49b856d8f02b7959e08eb33086	42	0/42 (0.00%)	report	download
2014-08-13	carrosmezcladores.com/ZeuS/serverph	active	c9e0decc10f214b47bb8cbb628a20ea	141824	47/53 (88.68%)	report	download
2014-08-11	hisahdjalstudioaso.ru/tizf2qpsdsd/a	active	41f50928296f5b69eaf18557a41a9109	213875	0/54 (0.00%)	report	download

Binaries displayed: 3

Copyright © 2014 zeustracker.abuse.ch, version 1.2 / 2010-09-12

Figure 3.5: ZeuS tracker monitored page.

In this way every time a new analysis is performed its results are used to improve future analyses. In fact, as already explained in Section 3.2.2, during the analysis the VMs are programmed to visit a certain number of URLs. The URLs that Prometheus select to be processed in order to perform the analysis are those one that are most likely to be targeted by samples. URL's likelihood is taken from the malware's samples previously analyzed; in fact the more the system found, in the memory of infected VMs, regular expressions that match with a certain URL the more this URL is likely to be targeted.

3.4 The Crawler

Another important component of Prometheus is the crawler which automatically retrieves from ZeuS tracker [2] and SpyEye tracker [1] (Figures 3.5,3.6) new malware's samples uploading them on the web service, in order to keep updated our sample dataset. The two sites mentioned above are two of the most important malware trackers which offer to the analysts' community a great service collecting all the reports about new malware's samples active in the world. Once the malware has been reported on a tracker it has a short life, because either the malware owner remove it, or the owner of the server that unlawfully host the C&C server destroys it. For this reason it is very important to analyze active samples as soon as they are reported, obviously keeping this procedure automated will lead in having best results in analyzing the largest possible quantity of active samples and in keeping our database of regular expressions and URLs always fresh and updated in respect to the status of interest of the malware authors.



Figure 3.6: SpyEye tracker monitored page.

Chapter 4

Prometheus: implementation

In this chapter we introduce the architecture of Prometheus. We show how, referring to the approach expressed in Chapter 3, we developed Prometheus giving a better view of all the implementation details.

In Section 4.1 we provide a quick overview on the whole system, which can be considered divided in four parts: the web front-end, the back-end, the sandbox system Cuckoo and the crawler. These components will be analyzed into details in the next sections.

Section 4.4 gives a quick overview on Cuckoo, the sandbox system we decided to use to manage the VMs' execution.

In Section 4.5 we go deep into the details of the implementation of the system's back-end which is essentially the core of Prometheus, while in Section 4.7 we introduce the web front-end.

Section 4.8 explains the second kind of interaction we have with the back-end, the one given by the crawler which automatically submit new analysis to our system.

4.1 The Overall Architecture

As said before the whole system (Figure 4.1 and 4.2) is composed by four parts. The central part is the back-end. Its main objective is to receive as input the specification of the submitted analysis, then schedule it managing the available resources (VMs), interact with the sandbox system, receive the dumped DOMs from the virtual machines, receive the dumped memory from the virtual machine devoted to memory analysis, process the data in the way explained at the end of Chapter 3 and output and store the results. When partial results are ready the back-end notifies the web client (if it is

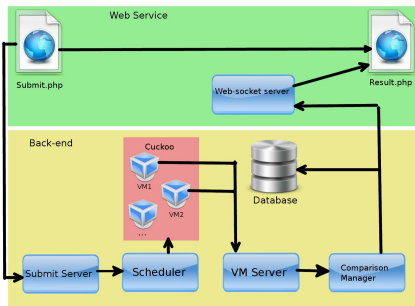


Figure 4.1: Schema representing the architecture of Prometheus. – web-service + back-end –

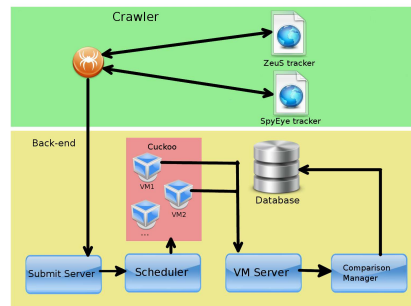


Figure 4.2: Schema representing the architecture of Prometheus. – crawler + back-end –

an online analysis) of the completion in order to let it retrieve and show the results to the user.

This project should have required a big effort for the implementation of a sandbox or a system able to automatically spawn VMs and control them automating the execution of the web browser. This work, also, would have been very risky since sandboxes are a very sensitive subject. For these reasons we decided to adopt a ready-to-use solution: the Cuckoo sandbox. Cuckoo is an open-source project widely supported by a great community of developers and researchers which easily allows to automate dynamic analysis of malware. Cuckoo offers wide set of Python APIs (it has also a set of REST APIs) that leave to the user the only burden to script the module that is executed on the VM to automatize the analysis.

Central and accessed by almost every component the MySQL database contains all the information about the regular expressions found during the memory analysis, the full list of interesting URLs, the list of sample analyzed plus other information found during the memory analysis and a table indexing by sample and URL all the files containing the results of the analysis.

The third component is the web interface that allows to every analyst to directly interact with our system and retrieve results in a human readable format. The interface allows to the users to navigate through old analysis, retrieve information and statistics about the most targeted URLs looking at the actual status of interest of the malware authors. It also gives the possibility to retrieve the code of the injections.

Furthermore, during the analysis processing, the front-end communicates continuously with the back-end via web-socket. In this way every time partial results relative to certain URLs are ready they are instantly showed

to the users drastically reducing the waiting time.

Another interaction with the central back-end is done by the crawler that automatically submits samples found online on two of the most famous trackers for the malware of our interest.

After this quick overview is interesting to notice that every of the above mentioned components is independent from the implementation of the others and can be easily replaced or enhanced only keeping constant its APIs. Furthermore since all the components are developed as servers they can be moved on different machines without any effort making Prometheus very easy to scale and also adaptable to a cloud environment.

4.2 Libraries and Tools

Prometheus is implemented mainly in Python2.7. Python's elasticity and usability allowed us to get quickly to our results and, since it has a big support from the developers community, we were able to find all the libraries we needed already implemented and well documented.

Our system interacts with the Cuckoo sandbox which is also written in Python2.7. As we will better explain in Section 4.4 Cuckoo is an open-source sandbox which interacts with some of the most common virtual machine manager. It is able to start a VM and automate the execution of an analysis task and it offers the possibility to process its result with some powerful forensic tools. We used as Virtual Machine Monitor VirtualBox by Oracle which is the one suggested by Cuckoo's developers. In order to perform memory forensic analyses we used Volatility. The Volatility Framework is a completely open collection of tools, implemented in Python, for the extraction of digital artifacts from volatile memory (RAM) samples. Volatility offers a wide set of common functionalities but it also allows to increase its power through easy to develop plugins.

We developed a Volatility Plugin base on YARA to automate the WebInject targets extraction. YARA is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples. YARA allows to define rules based on textual or binary patterns. Each rule consists of a set of strings and a boolean expression which determine its logic.

Another important Python tool we used is WebDriver by Selenium. This library is installed inside our VMs and is used to control and automate the execution of the browser, in our case Internet Explorer. Moreover WebDriver APIs allow to deal with popups, alerts and, above all, to retrieve the DOMs of the visited web pages

We also used a Java tool: XMLUnit. XMLUnit is used by our back-end to compare different DOMs downloaded during the analysis. We decide to use this Java tools as we have seen that other existing Python tools and libraries are not so precise and efficient in resources consumption as XMLUnit will.

In developing the crawler we took advantage of some Python libraries: Mechanize and BeautifulSoup. Mechanize allows to easy simulate the behavior of a web browser via code, automating the retrieval of web pages. Exploiting BeautifulSoup then the crawler inspects the HTML structure parsing the tables of the web sites we crawl looking for active samples.

For the web server we employed and configured an installation of Apache2 with PHP5. Employing PHP we were able to easy develop a web service which interact with our back-end and the database. We use a MySQL database that is central and interacts with all the components of our system working as glue and coordinating the whole process. We developed the front-end in HTML5, we used JQuery for the AJAX implementation and we employed Bootstrap for the style. JQuery gives us a stable set of JavaScript APIs in order to implement AJAX call and DOM managing. Instead, Bootstrap allows to quickly develop a very userfriendly GUI that enhances the usability of our system. Furthermore we implemented a JavaScript's web-socket in order to dynamical send results to our clients so to drastically reduce users waiting time. The Web socket communicates with a Python server coordinated by our back-end.

Furthermore we used the “Wkhtmltopdf”¹ command line toolkit to render collected DOMs and present them to the user. We believe that showing injections also in a graphical way could be useful for analysts in presenting the threat to their clients giving them a taste of what are the kind of injections that happen on their websites, furthermore a graphical representation could help in finding countermeasures.

4.3 The Configuration file

Listing 4.1 shows the Prometheus configuration file. The first four parameters are the amount of time, in seconds, to wait for the malware activation, the amount of time to wait after having loaded a web page and before start loading the next one, the number of DOMs to collect before sending all them in single JSON, the allowed amount of time to load a web page and dump its DOM. The next parameters are the network address of the machine where

¹<http://wkhtmltopdf.org/>

the Prometheus back-end is hosted and the TCP ports on which the VMs Server and the Submit Server listen. Then it is specified the number of Comparer Threads to be started, the number of clean and infected VMs to be used during the analysis, the number of last samples considered during an URL analysis, the threshold ε explained in Section 3.3.3, the number of URLs processed in a single analysis and the critical timeout of the analysis. This last parameter set a timeout that, when expired, causes the interruption of the running analysis. Finally we have the parameter required to connect to the database.

Prometheus reads and parses the configuration file every time a new analysis is scheduled. This makes it possible to modify the configuration parameters and to make the modification effective without restarting Prometheus. Some of these parameters are also configurable during the analysis submission via web service.

```

<prometheus.config>
  <vm>
    <sleep_time>40</sleep_time>
    <loading_time>0</loading_time>
    <doms_per_json>2</doms_per_json>
    <page_timeout>15</page_timeout>
  </vm>
  <server>
    <host>192.168.56.1</host>
    <port>54500</port>
    <submit_port>54510</submit_port>
    <n_task_threads>10</n_task_threads>
    <n_clean_vm>6</n_clean_vm>
    <n_infected_vm>6</n_infected_vm>
    <last_samples>1</last_samples>
    <black_diff_threshold>0.8</black_diff_threshold>
    <n_urls>62</n_urls>
    <timeout>600</timeout>
  </server>
  <db>
    <db_host>localhost</db_host>
    <db_name>db_name</db_name>
    <db_username>username</db_username>
    <db_password>password</db_password>
    <db_unix_socket>/path/to/mysql.sock</db_unix_socket>
  </db>
</prometheus.config>

```

Listing 4.1: Prometheus configuration file

4.4 Cuckoo

As stated before, malware sandboxing is a very sensitive process and the implementation of a sandbox could have lead us to spend a lot of time to come

out with an efficient solution. For this reason we decided to use an already existing sandbox: Cuckoo². Cuckoo exposes to us the required elasticity also to implement a task that is quite on the border line of Cuckoo scope, in fact in our analysis, on average, half of the machine are not effectively sandboxing any malware but they are simply acting as reference without any infection. Furthermore Cuckoo monitors the VMs and allows us to retrieve some additional information, like the memory dump that we use for our memory analysis, but also it could take screenshots or perform taint analysis that are interesting features that could be included in next development of Prometheus. In the next section we will quickly introduce how Cuckoo works (Section 4.4.1) then we will explain what are the modules that we implemented to automate our analysis in Section 4.5.1.

4.4.1 Quick Overview

Cuckoo is a malware sandbox written in Python which transparently interacts with all the most common virtual machine hypervisors. Oracle VirtualBox³ is the one suggested by Cuckoo's developers. Cuckoo exposes a wide range of APIs and the one we used in this project is the command line APIs which easily allows to submit new analysis. At configuration time Cuckoo needs to have registered in the configuration files a set of VMs already installed and ready to be used. Each VM has to run the Cuckoo agent that is a Python script implementing an RPC server through which Cuckoo communicates with the VMs. Moreover for each VM there has to be a snapshot from which the VM is restored. For our project we configured 20 VMs with Windows XP and Internet Explorer 8. The network card was set in "host only" mode but the host had iptables configured in order to forward Internet traffic to the VMs.

When an analysis starts, Cuckoo unfreezes the first free VM restoring the snapshot, then it contacts the agent running on it uploading the malware and specifying the analysis to execute. After the execution of the analysis Cuckoo restores the snapshot and starts processing the information it has retrieved. In our case this last part is completely excluded since we want to process results with our components and since most of the data that we retrieve are sent back to our back-end by the VM during the analysis. Once Cuckoo was configured the only thing the we needed to do was writing the modules to be executed on the VM. For the analysis described before we needed to implements two modules: one for the DOMs retrieval and one

²<http://www.cuckoosandbox.org>

³<https://www.virtualbox.org>

to retrieve the memory dump. These two modules are introduced in the next section. From now on we will refer to the term analysis indicating the whole process performed by our system between the submission and the representations of the results. Instead we will refer to Cuckoo's task indicating one single execution of a VM from which our systems retrieve the data for an analysis. In practice an analysis is composed by multiple Cuckoo's tasks. When a new analysis is submitted our system submits to Cuckoo a number of tasks equal to the selected number of VMs plus a further task from which retrieving the memory dump. Every Cuckoo's task runs a VM executing the specified package as we will explain in Section 4.5.1. All the task are executed in parallel.

4.5 Back-end

The Python back-end is the central core of Prometheus and also the most complex component. It is composed by the following main modules:

- The Submit Server: in charge of receiving analysis submissions and forwarding them to the Scheduler.
- The Scheduler: it receives analysis requests from the SubmitServer, enqueues them, and schedules them when the system is free. It interacts with Cuckoo as it submits the required tasks.
- The VMs Server: in charge of communicating with the VMs. It provides to the VMs the URLs list to process and receives from each VM the dumped DOMs. Every time a DOM is dumped and sent, the VMs Server forward it to the Comparisons Manager.
- The Comparisons Manager: it is the main module of the architecture. It gets notified whenever a new DOM has been correctly sent to the VMs Server and handles the entire DOMs comparisons process.
- The Comparer.jar: it is the component that performs the DOMs comparisons. Given two DOMs it outputs the differences between them.
- The Memory Analysis module: it handles the memory forensic analyses, extracting useful information from infected memory dumps.

Moreover two other components interact with the back-end system:

- Cuckoo analysis packages: Cuckoo modules invoked during the Cuckoo tasks. Their role is to initialize and guide the execution of the VMs.

- The Executor: is the component that runs inside each VMs. It handles the VMs' parameters, controls the browser and communicates with the VMs Server.

4.5.1 Cuckoo analysis packages

The analysis packages are a core component of Cuckoo Sandbox. They consist in structured Python classes which, executed in the guest machines, describe how Cuckoo's analyzer component should conduct the analysis. A package class contains three methods *start*, *check* and *finish* that are executed on the VM respectively when the VM start, during the VM running and when the VM is stopped. The check method is called repeatedly every two seconds and specifies the condition to stop the VM (e.g., stop the VM where a process complete its task).

We developed two different packages: the first one, the webinject package, deals with the VMs that have to process URLs and download their DOMs, the second one, the meminject package, deals with the VM from which we get the memory dump.

- Webinject package: it is the main used package. As shown in Listing 4.2, it receives some parameters through which a boolean that indicates if the specific machine has to be infected (as we said we use both clean and infected machines). According to this parameter the package executes the sample and starts the Executor module. The VM is stopped when the Executor process terminates.
- Meminject package: it executes the malware, waits for the sleep time and opens the browser in order to let the malware load in memory the WebInject targets. Then it dumps the memory and stops the VM.

```
class WEBINJECT(Package):
    """WebInject malware analysis package."""

    def start(self, path):
        ip = self.options.get("ip", None)
        sample_name = self.options.get("sample", None)
        clean = self.options.get("clean", False)
        doms_per_json = self.options.get("doms_per_json", 2)
        url = self.options.get("url", None)
        url_id = self.options.get("url_id", None)
        sleep_time = self.options.get("sleep_time", 1)
        loading_time = self.options.get("loading_time", 0.5)
        page_timeout = self.options.get("page_timeout", 15)

        if ip is None:
```

```

        raise CuckooPackageError("WebInject Package: Unable to find ip option
            , analysis aborted")
    if sample_name is None:
        raise CuckooPackageError("WebInject Package: Unable to find sample
            option, analysis aborted")
    if not clean:
        #run malware sample
        malware = Process()
        if not malware.execute(path=path, suspended=False):
            raise CuckooPackageError("WebInject Package: Unable to execute
                malware process, analysis aborted")

    #build sample analysis args string
    if url is None or url_id is None:
        args = r"C:\\seltests\\executor.py {0} {1} {2} {3} {4} {5} {6}".
            format(ip,sample_name,sleep_time,loading_time,clean,doms_per_json
                ,page_timeout)
    #build URL analysis args string
    else:
        args = r"C:\\seltests\\executor.py {0} {1} {2} {3} {4} {5} {6} {7}
            {8}".format(ip,sample_name,sleep_time,loading_time,clean,
                doms_per_json,page_timeout,url_id,url)

    #run executor
    executor = Process()
    if not executor.execute(path=r"C:\\seltests\\selenv\\Scripts\\python.exe"
        , args="%s" % args, suspended=False):
        raise CuckooPackageError("WebInject Package: Unable to execute
            executor process, analysis aborted")

    return executor.pid

def check(self):
    #check stop condition
    return Process(pid=self.pids[0]).is_alive()

def finish(self):
    return True

```

Listing 4.2: Webinject Cuckoo package

4.5.2 Executor

The Executor is the component that, running inside the VMs, manages the processing of the URLs and the download of their DOMs. It is invoked by the webinject Cuckoo analysis package and requires the following parameters:

- the address and the port of the VMs Server to communicate with it;
- the sample ID;
- the sleep time to be waited in the infected VMs;

- a boolean indicating if the VM is infected or not;
- the loading page timeout;
- the number of DOMs to collect before sending them to the VMs in a single shot;
- the URL to visit. In case of an URL analysis.

As we said in Section 4.3, most of these parameters can be set in the Prometheus configuration file.

In the infected VMs the first thing the Executor does is sleeping, since we need to allow a minimum amount of time for the trojan to hook the APIs used by the browser, otherwise no WebInject will be detected before this interval (see Appendix B).

Now we have to distinguish if the running analysis is a sample analysis or an URL analysis. In the case of an URL analysis the Executor opens the browser interacting with WebDriver, visits the desired web page, downloads its DOM and sends it to the VMs Server.

In the case of a sample analysis the Executor first of all downloads from the VMs Server the URLs list to process and then executes two different threads in parallel. The first thread visits each web page downloading their DOMs. The second thread sends the already downloaded DOMs to the VMs Server. The two threads communicates through a shared queue according to the usual producer-consumer scheme. As soon as the first thread, the producer, downloads a DOM and puts it in the shared queue, the consumer get the DOM from the queue and sends it to the VMs Server. The sender can also collect a certain number of DOMs, as specified in the configuration file, before sending them all together. The downloaded DOMs are sent to the VMs Server in a JSON format. The producer-consumer scheme allows Prometheus to send the already dumped DOMs and so to start processing them without waiting for the entire DOMs collection phase to finish. This improve performance and allows Prometheus to show partial results during the analyses.

4.5.3 Submit Server

The Submit Server receives via socket a description of the analysis that is required (by a web user or by the crawler, from now on both referred as user). The description indicates whether the user has required to analyze an URL or a sample. In the first case the description is composed by the URL to be analyzed and optionally some parameters: the sleep time to

wait in order to be sure that the sample is active on an infected machine, number of samples to be considered in the analysis, number of clean and infected VMs to use for the analysis and the ε threshold for the fourth filter heuristic. If these parameters are not given default values are taken from a XML configuration file. In the second case it is composed by the name of the sample, we adopted the format `< md5sum(sample) > .exe`, and some optional parameters like before, regarding the number of VMs to use, the sleep time and the ε threshold. Then the Submit Server forwards to the Scheduler the analysis request and its parameters. When the request is forwarded to the Scheduler the latter returns to the state of the Scheduler: free or busy depending whether or not the system is processing another analysis. This is used to inform the user that his request has been enqueued or it has been executing.

4.5.4 Scheduler

The Scheduler is the component that handles the analysis requests. In this first release Prometheus is able to run one analysis at time. The Scheduler receives the analysis request from the Submit Server and puts them in a queue. The state maintained by the Scheduler can be free or busy. The Listing 4.3 shows the Scheduler run method. If the system is not busy the Scheduler takes the next analysis to process from the analysis queue or waits if no analyses were submitted. We used the Python synchronized *Queue* that embeds the synchronization handling and the wait-notify mechanism. The Scheduler checks also that the critical timeout did not expire. In this case it stops the running analysis and set the system's state to free. If the system is processing another analysis the Scheduler simply enqueues the analysis request in the analyses queue otherwise it proceeds as follow. It takes the analysis request on top of the queue and distinguishes if it is an URL analysis or a sample analysis. It then interacts with Cuckoo submitting the required tasks, that means starting the VMs.

As shown in Listing 4.4 the Cuckoo tasks are submitted through the *submit.py* Cuckoo utility called through a Python *Popen*. This utility allows to select the package to be used and to specify all the required parameters.

Furthermore, if the scheduled analysis regards a sample and not an URL, the Scheduler starts the MemoryAnalysis component which submits to Cuckoo a further execution of a VM using the specific Cuckoo package in order to retrieve the memory dump.

```

def run(self):
    print "Scheduler: active!"
    while(True):
        if self.free:
            analysis = self.analysis_queue.get()
            self.analysis_queue.task_done()
            self.free = False
            self.analysis_start_time = time.time()
            self.submit_analysis(analysis)

        elif not self.free and (time.time() - self.analysis_start_time > self.config.
            timeout):
            self.stop_current_analysis()
            self.free = True

        else:
            time.sleep(10)

```

Listing 4.3: Scheduler run method

```

def submit_sample_analysis(self, sample_id, sample_name):
    print "Scheduler: start sample analysis, {0}".format(sample_name)
    #start mem analysis
    mem_analysis = MemoryAnalysis(self.db_man, sample_name, sample_id, self.
        config.sleep_time)
    mem_analysis.start()

    for i in range(self.config.n_infected_vm):
        p = Popen("python ../cuckoo/submit.py --package webinject --options
            \ip={0}:{1},sample={2},sleep_time={3},loading_time={4},doms_per_json
            ={5},clean=False,page_timeout={6}\" ../samples/{7}".format(self.config.
                host, self.config.port, sample_id, self.config.sleep_time, self.
                config.loading_time, self.config.doms_per_json, self.config.
                page_timeout, sample_name), stdout=PIPE, stderr=PIPE, shell=True)
        (output, err) = p.communicate()

    for i in range(self.config.n_clean_vm):
        p = Popen("python ../cuckoo/submit.py --package webinject --options
            \ip={0}:{1},sample={2},sleep_time={3},loading_time={4},doms_per_json
            ={5},clean=True,page_timeout={6}\" ../samples/{7}".format(self.config.
                host, self.config.port, sample_id, self.config.sleep_time, self.config.
                loading_time, self.config.doms_per_json, self.config.page_timeout,
                sample_name), stdout=PIPE, stderr=PIPE, shell=True)
        (output, err) = p.communicate()

```

Listing 4.4: Scheduler submit_sample_analysis method

4.5.5 VMs Server

The VMs Server is the component that communicates with the VMs. It consists in an HTTP server, implemented with the *ThreadedHTTPServer*

Python class, used to exchange JSON. It contains two methods to handle GET and POST requests. The GET request returns a JSON containing the list of the URLs to be processed by the VMs. In particular, as explained in Section 3.2.2, the VMs Server selects the most matching URLs, that means the URLs that have the highest probability to be injected by trojans, on the base of the knowledge gained in the past analyses. The POST requests are used by the VMs to send the dumped DOMs. Even in this case we use the JSON format to send data.

When a VM sends a new DOM the VMs Server preprocesses it and stores it. The preprocess function convert the HTML DOM into an XML format in order to allow easier comparisons. The VMs Server inserts a new entry in the DOM table of the Database and saves the dumped DOM in a file named `< id_dom > .dom`. The VMs Server shares a queue with the Comparisons Manager used to notify the arrival of new DOMs.

4.5.6 Comparisons Manager

The Comparisons Manager is the component that handles the entire DOMs Comparisons process. It has been designed to be fully parallelizable and asynchronous. This means that all the computations are performed in parallel and as soon as the data required are available. The functioning of the Comparisons Manager is based on the data structure that it holds (Figure 4.3). The main data structure that the Comparisons Manager handles consists in a two levels dictionary. When we discuss about the Comparisons Manager and how it works we refer to DOM, sample and URL but we actually mean their ID (DOM ID, sample ID and URL ID) that are unique identifiers and, since they are long integers, they are obviously easier to be managed. The IDs refer to the primary key in the database tables.

The first dictionary of the data structure is indexed by sample and points to a second level of dictionaries. For each sample the first dictionary points to a second dictionary indexed by URL. Finally for each URL the second dictionaries point to a structure containing four lists: the list of clean DOMs, the list of infected DOMs, the differences blacklist and the differences whitelist. Since the blacklists and the whitelists are shared and accessed by different Comparer Threads, the Comparisons Manager holds also a dictionary of Python *Locks* used to handle the race conditions.

Whenever the VMs Server notifies the arrival of a new DOM this is appended in the correct list, clean or infected, according to the sample-URL pair. The first clean DOM is considered as reference and from now on we will call it the reference DOM. When comparing two DOMs, we call control

DOM the one we use as the reference and we call test DOM the second term of the comparison. It is important to stick to this arbitrary assignment of control and test node/DOM to define the direction of the difference: for instance, if a field on the test node is missing in the control node that may be a WebInject. If it's the other way round that is a removed node. The objective of the Comparisons Manager is to compare all the infected DOMs and all the clean DOMs with the reference DOM. The differences output from the comparisons of a clean DOM and the reference DOM are added into the whitelist while the differences output from the comparisons of an infected DOM and the reference DOM are added into the blacklist.

When the Comparisons Manager is notified of the arrival of a new DOM, it checks if there are new comparisons to do. If it is the case it puts in the tasks queue the DOMs to be compared and some other information. In the meanwhile, a number of threads in charge of computing the comparison tasks, called Comparer Threads, waits for a queue to be filled with tasks: as soon as one is added, a free thread pops it and begins to process it. Each comparison task consists in: the ID of the DOMs to compared, the sample and the URL the DOMs refer to and a further boolean indicating if the produced differences have to be appended in the whitelist or in the blacklist. The DOMs comparisons are performed by a further component, the Comparer.jar. The Comparer Threads execute the Comparer.jar giving the IDs of the DOMs to be compared. The Comparer.jar reads from files the previously stored DOMs, compares them and outputs the differences found. When the Comparer.jar finished comparing the DOMs, the ComparerThread that called it parses appropriately its output and insert the differences in the whitelist or the blacklist according to the DOMs compared.

Finally, when all the comparisons relating a URL are done, the ComparerThread that executed the last comparison performs the filtering phase.

4.5.7 Comparer.jar

The Comparer.jar, based on the XMLUnit library, is the component that executes the comparisons between two DOMs and outputs the differences between them. As we said all the comparisons are done respect to the reference DOM. When it is called, the Comparer.jar, receives as parameters the identifiers of the DOMs to compare. The Comparer.jar reads from files the corresponding DOMs, compares them and outputs the differences found. The XMLUnit classes that are most important to us and that are used by the Comparer.jar are the *Diff* class and one of its subclasses,

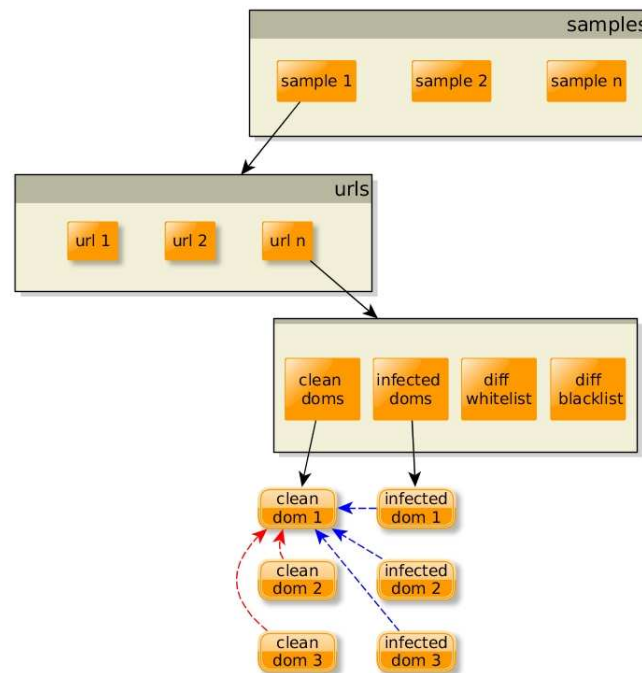


Figure 4.3: Comparisons Manager Data Structure. The red arrows indicate the comparisons whose differences are appended to the whitelist, the blue arrows those whose differences are appended to the blacklist

DetailedDiff. The first one walks through the DOMs and returns a boolean value that states whether the two DOMs are equal. The second class, on the other hand, goes on with the comparison even after the first difference has been found, collecting all the differences in the remaining part of the document. In particular the instances of the *DetailedDiff* class have a method, *getAllDifferences()*, which returns a List of the differences. Each difference is then printed on the standard output. The Comparer Thread that called the Comparer.jar through a Python *Pipe* receives the output of the Comparer.jar and parses it properly getting all the differences.

4.5.8 Memory Analysis

The memory analysis is performed in parallel to the webinjections detection process. It is handled by the Memory Analysis module, running on a different thread, which is launched by the Scheduler when it starts a sample analyses. The memory analysis thread first of all submits the task to Cuckoo invoking the meminject package previously described and waits for the memory dump. When the memory dump is generated it is inspected.

The most important role of the memory analysis thread is extracting the URLs and Regular expression from the memory dump. To do this we developed a volatility plugin based on YARA. The plugin extracts the strings that satisfy the designed YARA rule and that are allocated in the address space of the browser (in our case Internet Explorer). Once the Memory Analysis module extracted the URL regular expression, it stores the data and checks if some of the URLs in the database match any of the extracted regular expressions.

Exploiting another volatility plugin the Memory Analysis module extracts also the address of the C&C server and the RC4 cryptographical keys used to encrypt the configuration file and the connection to the C&C server.

4.6 Back-end functioning

As we said in Section 3.2, Prometheus provides two kinds of analyses, the sample analysis and the URL analysis. Both the analyses can be logically separated in three phases, Data Collection, Data Processing, Results elaboration, with the exception that the URL analysis does not include the memory forensic inspection. The three phases can be thought separately but indeed they are partially overlapped. Moreover the URL analysis can be seen as a part of the sample analysis since the sample analysis executes the same three phases considering more URLs. In parallel to the webinjections detection process also the memory forensic analysis follows the above three phases.

4.6.1 Phase 1: Data Collection

DOMs Collection

When a new analysis is submitted to the Submit Server and scheduled by the Scheduler the first phase, the data collection starts. The Scheduler submits the Cuckoo tasks that start all the require VMs in parallel. The Cuckoo package infects part of the VMs and starts the Executor. The Executor, after having waited for malware activation in the infected machines, downloads from the VMs Server the JSON containing the list of URLs to process and, leveraging the WebDriver APIs, opens Internet Explorer. It then starts visiting all the URLs dumping their DOMs and sending them to the VMs Server through a POST request in JSON format. During the URL visiting process we set a timeout for the page download that means that we allow limited amount of time to dump its DOM. The page timeout can be set in the configuration file, during the evaluation with set 15 seconds. If the

timeout expires the page is skipped and an Error notification is sent to the VMs Server. When a VM sends a new DOM the VMs Server preprocesses it, inserts a new entry in the Database and stores the DOM in a file named with the ID of the database entry. The preprocess function convert the HTML DOM into an XML format in order to allow easier comparisons. Finally the VMs Server put the DOM ID in a queue shared with the Comparisons Manager.

Memory Dump

In case of a sample analysis Prometheus performs also a memory forensic inspection. When the sample analysis is scheduled the Scheduler starts the MemoryAnalysis module that manages the memory inspection.

In the first phase it submits the Cuckoo task starting another VM. The meminject Cuckoo package running inside the VM waits for malware activation and then open Internet Explorer. We do this because we noticed that the regular expression indicating the target website configured in the WebInject configuration file are loaded in memory only when the hooked browsers is launched. In fact the regexes are loaded exactly in the browser address space. After some seconds the browser has been launched the VM is stopped.

Just before stopping the VM Cuckoo, through the VirtualBox API, dumps the VM memory and save the dump into a file in the report directory of the task.

4.6.2 Phase 2: Data Processing

DOMs Comparison

The DOMs Comparison process is entirely managed by the Comparisons Manager and by the Comparer Threads it starts.

As described in Section 4.5.6 whenever a the VMs Server puts a new DOM in the shared queue the Comparisons Manager pops it and adds it into its own data structure. When a new DOM is registered in the data structure the Comparisons Manager checks, on the base of the already dumped DOMs, if new comparisons can be done.

- If the new DOM is a clean one and it is not the first DOM dumped, that means there is already the reference DOM, a new task that indicates the comparison between the new DOM and the reference one is added in the task queue.

- If the new DOM is a clean one and it is the first DOM dumped, the new DOM is selected as reference DOM and the Comparisons Manager add a comparing task for each injected DOM already dumped.
- Otherwise if the new DOM is an infected one the Comparisons Manager add a comparing task if the reference DOM has been already dumped.

This mechanism (shown in the Code Snippet 4.5) allows to perform all the comparisons as soon as the data required are available.

```
clean_doms = self.data[sample][url][CLEAN]
infected_doms = self.data[sample][url][INFECTED]

if clean:
    if clean_doms: #reference dom yet received
        self.task_queue.put([sample, url, clean_doms[0], dom, False, analysis])
    else:
        for infected_dom in infected_doms:
            self.task_queue.put([sample, url, dom, infected_dom, True, analysis])

        clean_doms.append(dom)
else:
    if clean_doms:
        self.task_queue.put([sample, url, clean_doms[0], dom, True, analysis])

    infected_doms.append(dom)
```

Listing 4.5: Comparisons Manager checking new comparisons to do after the arrival of a new DOM

The tasks enqueue in the task queue are then removed from the queue and executed by a pool of Comparer Threads.

A comparison task consists in: the ID of the DOMs to compared, the ID of the sample and of the URL the DOMs refer to and a further boolean indicating if the produced differences have to be appended in the whitelist or in the blacklist.

Each Comparer Thread calls the Comparer.jar passing as parameters the two DOMs to be compared. The output of the Comparer.jar is then parsed and the differences are appended in the whitelist or in the blacklist according to the DOMs compared.

During the parsing a first difference filter is applied as we do not consider the differences we are not interested in. We consider four type of difference:

- Node insertion: This is really important to detect one of the most common web injection. Most of the information stealer add new fields

in forms injection the `<input/>` node.

- Attribute insertion: This type of difference detect injection that are mostly related to JavaScript code injection. In the common case the trojans add attributes such as *onclick* to bind JavaScript code and perform malicious actions whenever certain user-interface events occur.
- Node modification: This type of difference happens when trojans modify the content of an existing node. In the most cases the target node is a `<script>` one.
- Attribute modification: This type of difference happens when trojans change the value of an existing attribute to change the server that receives the data submitted in a form or to modifies the JavaScript code bound to a specific event.

Every difference is an object composed by three fields:

- Type: The nature of the difference. One of the four types previously discussed (node insertion, attribute insertion, node modification, attribute modification)
- XPath: The XML path to the node which is affected by the difference.
- Content: The value of the difference (e.g., in the case of a node insertion the content is the HTML node inserted with all its attributes).

Moreover, we consider for each difference an integer that counts how many times the difference has been detected. This is needed to apply one the filters (Section 3.3.3).

At the end of the DOM comparison phase we have the full differences whitelist and blacklist for each URL processed.

Memory inspection

As soon as the memory dump is generated the Memory Analysis module start inspecting it. To extract the target URLs and regular expressions we developed a volatility plugin based on YARA. YARA is a tool aimed that allows to define rules based on textual or binary patterns. Each rule consists of a set of strings and a boolean expression which determine its logic.

The plugin we developed scans the memory dump looking for all the strings that match a YARA rule. In particular, since we observed that the URLs and the regular expressions are loaded in the browser's memory, the plugin inspects only the Internet Explorer address space.

```
rule WebInjectTargets
{
  strings:
    $URL = /((http[s\*]?:\//)|\*)[\*\./0-9a-zA-Z-#]*\.(com|it|org|nl|uk|es|ru|
      net|ae|fr|de|bg)[\*\./0-9a-zA-Z-#]*/ fullword

  condition:
    for any i in (1..#URL) : (@URL[i] < @URL[i-1] + 20)
}
```

Listing 4.6: YARA rule defined to extract URLs and regular expression. In this example we omitted all the top-level domain showing only the most common ones.

Listing 4.6 shows the YARA rule that we defined. The \$URL string defines a regular expression that matches the URLs and the regular expressions defined in the WebInject configuration file. The modifier "fullword" guarantees that the string will match only if it appears in the file delimited by non-alphanumeric characters. The condition forces that the matching strings have to be placed at most 20 bytes of distant from the previous. Since we noticed that the regular expression are located sequentially this condition filters out all the matching strings that are not WebInject targets. In all the samples we manually analyzed the maximum distance between two URLs/regexes was 16 bytes and during the evaluation we did not incur in false negatives extracting correctly all the WebInject targets.

We integrated also another volatility plugin, ZeuSscan⁴, developed by the volatility's community. This plugin allows us to extract from infected memory dumps the RC4 cryptographical keys used to encrypt the configuration file and the connection to the C&C server and the address of the C&C server. However this solution depends on the implementation of the encryption scheme and it is guaranteed to work correctly only on Zeus and Citadel samples. We aim to improve the keys extraction mechanism to make it as general as possible in the next release.

4.6.3 Phase 3: Results elaboration

Differences Filtering

The differences filtering phase starts when all the comparisons related to an URL have been done and the differences whitelist and blacklist are completed. The objective of this last stage is to remove from the blacklist all the

⁴<https://code.google.com/p/volatility/source/browse/trunk/contrib/plugins/malware/zeusscan.py?r=2835>

differences that are not caused by the malware. At the end of this phase the blacklist will contain the final list of differences detected as webinjections. The filtering phase is performed by the ComparerThread that executed the last comparison.

We implemented the filters on the base of what said in Section 3.3.3. A first filter, the one discarding useless differences (e.g., deletion differences and differences regarding useless attributes), is implemented directly after the differences parsing process, so in this phase we consider the remaining three filters.

The first filtering action is removing from the blacklist all the differences present in the whitelist. Removing differences implies that we have some criteria to compare them and to determine whether two differences may be considered equal or not. In our context, two differences are equal if the same element has been added or modified in two different DOMs, with respect to our clean reference DOM. We observed which features actually allow to uniquely identify modified (or added) objects in two different versions of the same page and we decided to base our comparison on the type of difference and on the XPath of the elements. If two differences of the same type have equal XPath we consider them equal. This criteria is used in the first filter. However, dealing with highly dynamic page, we decided to add another concept of equality to remove from the blacklist the whitelisted differences. Since sometimes it happens that some node with a fixed content (mostly JavaScript) are omitted or situated in different places, that means different XPath, we consider the content of the difference. We remove from the black list all the differences that have the same Type, the same last node in the XPath and the same content of a difference in the whitelist. This represents the second filter. The last filter as said in Section 3.3.3 consists in removing those differences that are not present in at least $\varepsilon\%$ of the infected DOMs (with the same Type, the same last node of the XPath and the same content).

When the filtering is finished a new entry is inserted into the diff table in the database and the differences remained in the blacklist are stored into a JSON file (Listing 4.7). Then the Scheduler is notified that the URL processing has been completed so that it can notify the Web socket in order to update the results page (Section 4.7.1).

```
{
  "sample": 353,
  "url": 3,
  "differences": [
    {
      "xpath": "/html[1]/body[1]/form[1]/center[1]/table[2]/tbody[1]/tr[2]/
```

```

        td[1]/div[1]/table[1]/tbody[1]/tr[2]/td[2]/table[1]/tbody[1]/tr
        [3]",
    "id": 3,
    "value": "<tr><td class=\"BLUE\" width=\"220\">Password Dispositiva</
        td><td><input maxlength=\"8\" name=\"PASS\" onkeypress=\"
        TastoEnter(event.which, event.keyCode)\" size=\"8\" type=\"
        password\" value=\"\"/></td></tr>"
    }
  ]
}

```

Listing 4.7: Example of differences JSON file. It refers to the analysis of the sample 40ed6f385f4665537a9f401621deb2c2 and the URL <https://www.gruppocarige.it/grps/vbank/jsp/login.jsp>

Regular Expression Matching

When the Memory analysis module finished to extract and store in the Database the target regular expressions it, as explained in Section 3.3.3, checks if some of the URLs stored in our database match any of the regular expression. In order to do so, the extracted regular expressions are converted into the standard Python regex format of the *re* library. Every match, represented as a couple of URL ID and regex ID, is then stored in the database.

4.7 Web Service

The whole front-end is presented as a set of HTML pages rendered with PHP (Figure 4.4 show the link relation among web pages). We used personalization of the kit Bootstrap [17] for the CSS page layout and some JQuery [14] scripts for AJAX implementations and dynamic contents management. The web application offer to the users a navigation-bar with the following options:

- Navigate through DOMs (Figures 4.5, 4.6): This function allows users to look at a PDF version of the DOMs retrieved during every analysis in such a way to give developers the possibility to see the injections on their page with a visual representation and try to remove or randomize the injection hooking points. The GUI offers the possibility to search DOMs indexed by URL or by the MD5 hash of the sample whom analysis has produced the DOM. HTML DOMs are rendered by PHP via the `wkhtmltopdf` command line tool. Since this process can be quite time consuming and we believe that a very few portion of users

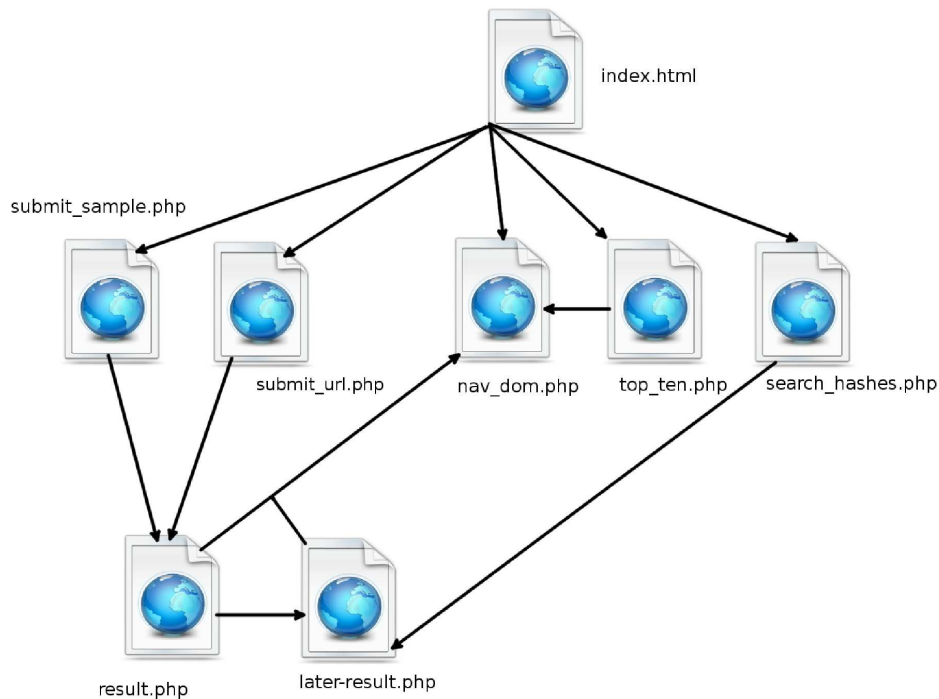


Figure 4.4: Link relations between web pages.

will use it, we preferred to render the PDF the first a user wants to see it instead of rendering it during DOM dumping to not interfere with the analysis performances.

- Search Through Hashes (Figure 4.7): This function allows users to retrieve analysis of old inactive samples already analyzed by our systems, either submitted by other users either by the crawler. The format of the representation of the results is identical to the one offered when the analysis is followed online by the submitter.
- Top Ten Targeted URLs (Figure 4.8): In this page we expose a very useful statistic, which is a ranking of the Top Ten URLs that results to have more matches with regular expressions found in the analyzed samples' memory. Administrating a page belonging to this top ten may be a big alert for web developers.
- Submit Sample (Figure 4.9): For sample analysis submission.
- Submit URL (Figure 4.10): For URL analysis submission.

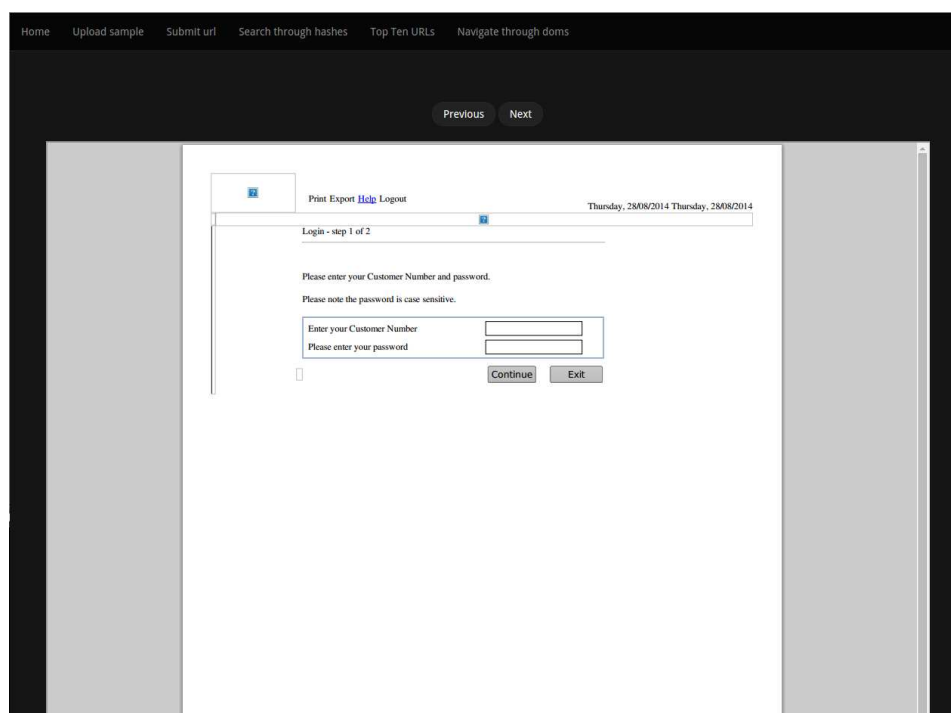


Figure 4.5: Snapshot of the DOM visualization page (no injections).

4.7.1 Analysis Submission and Results Retrieval

Both the two submission pages behave quite in the same way: in the submission pages users upload a malware's sample or an URL and optionally specify some analysis parameters and then attend for the results. When a user uploads a sample (or submit an URL) and pushes the button in order to start the analysis, the parameter inserted in the form are passed as analysis description to the Submit Server which forwards the analysis request to the Scheduler, as we have seen before. The analysis submission happens through a Python script `submit_analysis.py`, called by a PHP `shell_exec`, which sends the analysis request to the SubmitServer through a TCP socket. When the `submit_analysis.py` is called it returns the identifier of the sample or URL submitted and a boolean that indicates if the submitted analysis has been enqueued, because the system is processing another analysis, or if it has been already launched. After the submission the user is redirected to a dynamic page (`result.php`). `Result.php` is connected via web-socket with the web-socket server. Every time the Comparisons Manager stores on the filesystem and on the database the result of the analysis of a certain URL (or a sample) it notifies the web-socket server which notifies the web-client

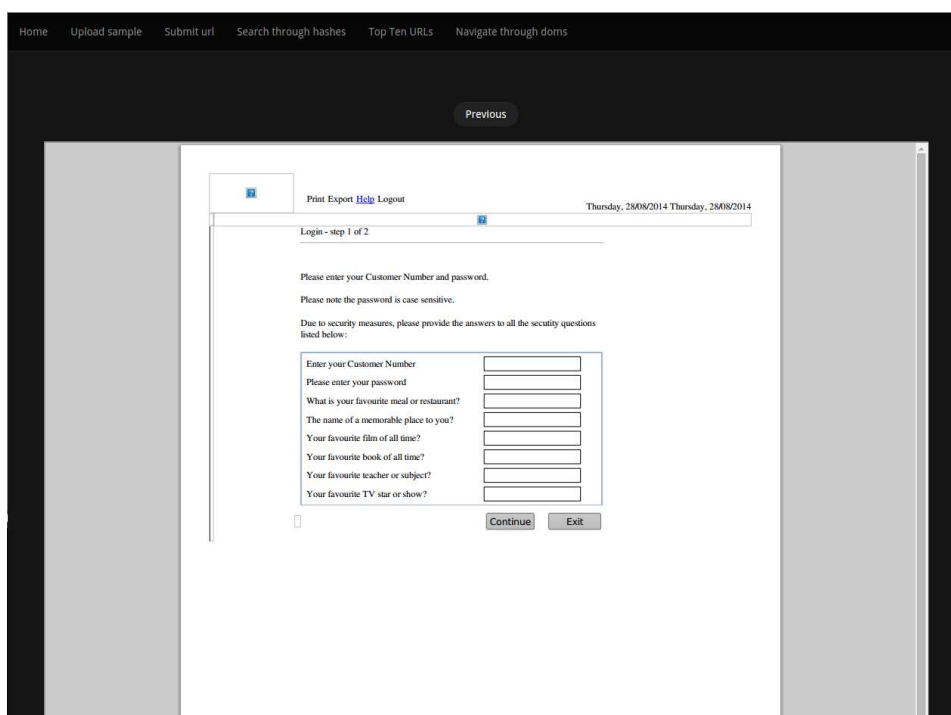


Figure 4.6: Snapshot of the DOM visualization page (with injections).

which retrieves the results from the filesystem in JSON format. If the analysis has been launched the page dynamically renders partial results as soon as they are ready, otherwise it simply show a link where the user will later find the results of the analysis. The same link is also proposed in the dynamic page for those users who do not want to assist dynamically at the analysis process. The results page is initially blank with a loading bar (Figure 4.12) and every time partial results came to the client the loading bar proceeds illustrating the user the progress of the analysis. When the analysis end the user is alerted with a sound message. Every time the client retrieves a result from the web-server it renders in that way: the ID of the analyzed element (URL's or sample's name) followed by the number of injections found as header of a hidden dropdown list in which are shown all the injections (type, XPath and content) (Figure 4.11). A click on the ID will redirect the user to Navigate through DOM section in which he can look at the injections found. When the analysis of a sample ends, the client loads also the result from the memory analysis (Figures 4.13, 4.14, 4.15) and, furthermore, it marks all the URLs that have a match with any of the found regular expressions adding a tip reporting the matching regular expression.

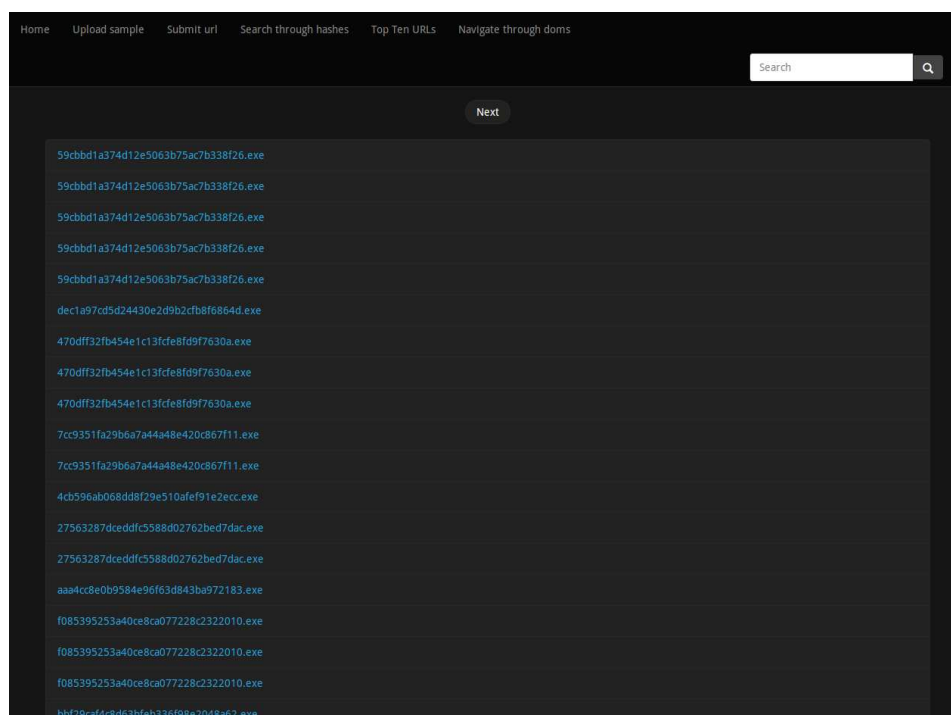


Figure 4.7: Snapshot of the search through hashes page.

4.8 The Crawler

The crawler is written in Python and uses the libraries Mechanize and BeautifulSoup to automatically check the presence of new samples on the ZeuSTracker and SpyEyeTracker pages. Mechanize offers a set of utilities used to automatically navigate in the web and monitor the pages of the two trackers.

Every time a new malware is online the crawler compares its hash with the ones present in the Prometheus database and, if the sample has not yet been analyzed, downloads the sample and submits its analysis to the Submit Server.

```
[22-08-2014 12:50:32] Start crawling..
[22-08-2014 12:50:32] Crawling https://zeustracker.abuse.ch
[22-08-2014 12:50:35] Crawling https://spyeyetracker.abuse.ch
[22-08-2014 12:50:36] Sleeping..
[22-08-2014 13:10:37] Crawling https://zeustracker.abuse.ch
[22-08-2014 13:10:38] Found new active Sample, d03c15ec47b096136d61bec17507b4c1
[22-08-2014 13:10:39] Sample Downloaded: d03c15ec47b096136d61bec17507b4c1.exe
[22-08-2014 13:10:39] Submitting Sample: d03c15ec47b096136d61bec17507b4c1.exe
[22-08-2014 13:10:41] Crawling https://spyeyetracker.abuse.ch
```

Listing 4.8: Crawler log example

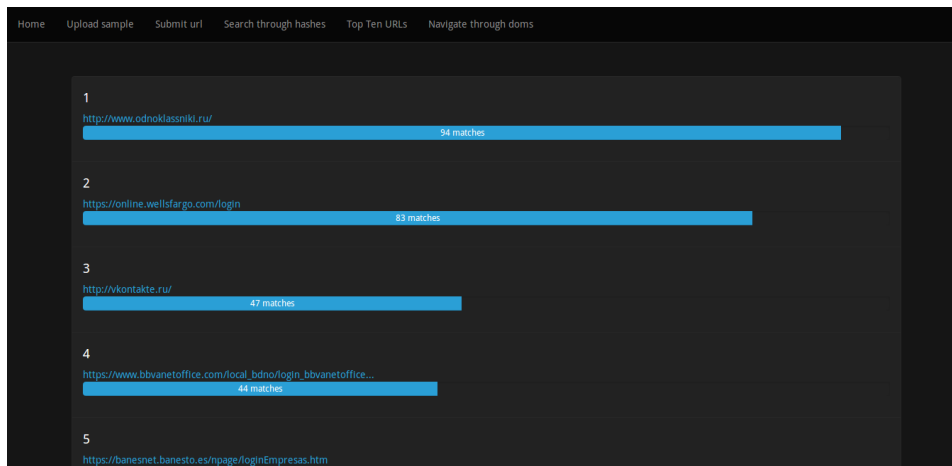


Figure 4.8: Snapshot of the Top Ten targeted URLs page.

The screenshot shows a web interface with a dark background and a light blue header. The header contains navigation links: Home, Upload sample, Submit url, Search through hashes, Top Ten URLs, and Navigate through doms. The main content area is titled "upload a sample:" and features a large grey document icon on the left. To the right of the icon is a form with the following fields and buttons:

- A file selection button labeled "Scegli file" with the text "Nessun file selezionato" below it.
- An "Advanced" button.
- A "sleeptime [seconds]:" label followed by an input field.
- A "number of clean vm:" label followed by an input field.
- A "number of infected vm:" label followed by an input field.
- A "black diff threshold [0-1]:" label followed by an input field.
- An "Upload and start analysis" button.

Figure 4.9: Snapshot of the sample submission page.

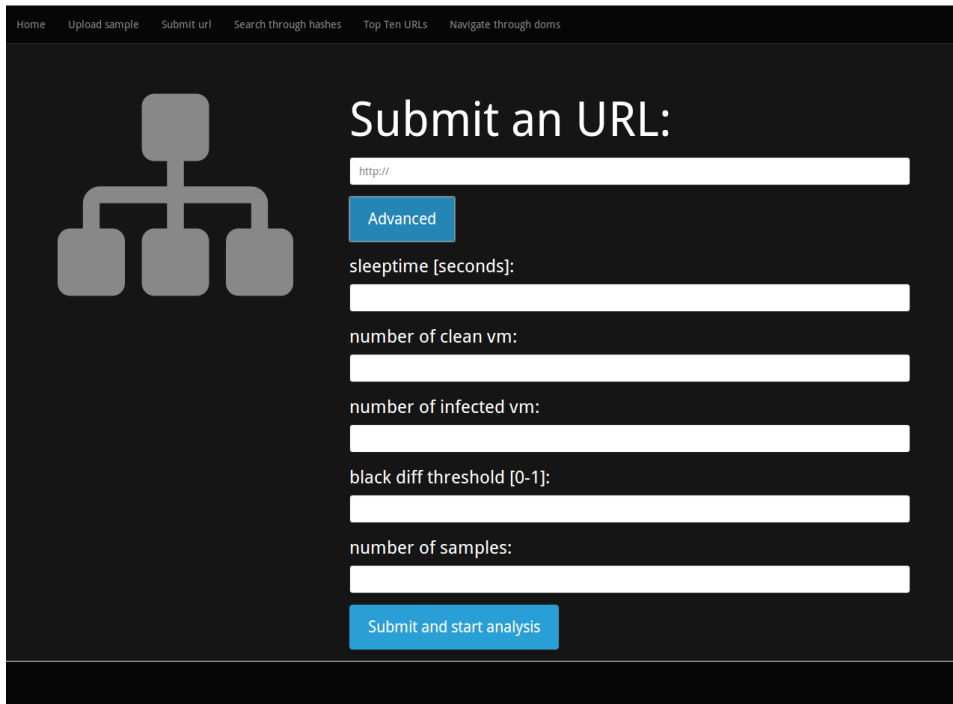


Figure 4.10: Snapshot of the URL submission page.

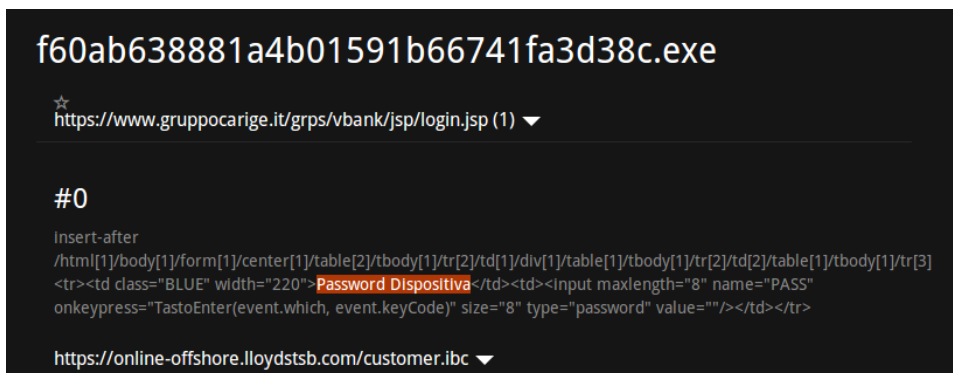


Figure 4.11: Particular of the results page.

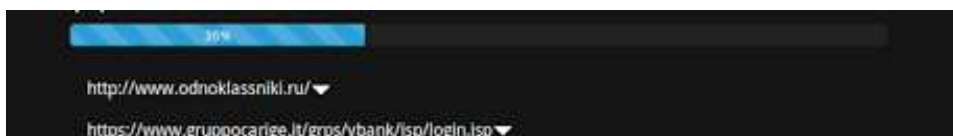


Figure 4.12: Loading bar in results page.

```

aaa4cc8e0b9584e96f63d843ba972183.exe
c&c and key:
  • c&c: http://extant.in/clips/server.php/config.bin
  • Credential key:
    6f821461b8df55f875ea02c8dc1896a231412743eecd7d892b8ef1233b77b4560c920d012f91cc39989a7c5f69352
    9e89d595a212d3367f4454084d4669470ddfa85d2726a4abc54a797d32cc33ab1888f0acb1f7ff6713e1ca811a446
    c7c6e51683ab9e6e4c4dc09315fcffb90ff3fef774586099038c347badc40807c9b7d7e426366878c5054e6552538
    7c21eae695286b2209517abf7ea59b9c3276da8b477925e320ec1df0ed2edbefebaf124924e21b383d6cce103c5
    c9faa3719d58da9cfe7812ab230645df9f2d94b57bb6d3f17bd0e4290bea37348d0e962b3005bba06c1aca14ff5a0
    b680860bde63fd5ee050131acab004a6fb44d1d8b58ae6e10000
  • Config key:
    70a74cf65c44bd171f919749fb5f16726ef81b032ccd7f04965a98470cce46db22cce6c548d5328b57e14f991218b
    bd83883fec471ddb364a405600fa04b0842af265d8a149228ca4e51870dcf4af150e5b266f059aba3c8c9071a9f23c
    22d9c751ed368d454676a2bc121343aeae7b5e2c081f7f961ad78944527b7a890ed56bf20769bac3601e42fb96fc
    382a510b1dafc293163c68d58d98fd26c3faed706b0eb73d0a6bb8500f5cb957b7ca1bc89d6ff93557902ec7d8e62
    6ddc4d0a1369df5e1dc76bfde0eef319390be30efae83b529a40a9ba33f2be863d1c84aa9d652a2ee935439e74d1
    77ef80b830f4373c8809118c7a255b7eb4de5315a23e41240000

```

Figure 4.13: Keys extraction in results page.

```

e4d3e14a3b4c275b52b16621954970ca5dbd51f81418a370000
☆☆
https://online.wellsfargo.com/login ▼
☆☆
http://www.odnoklassniki.ru/ ▼
☆
https://www.bbvanetoffice.com/local_bdno/login_bbvanetoffice... (5) ▼
☆☆

```

Figure 4.14: Regular expressions matching in results page.

```

Regexp From Memory
  • '*.microsoft.com/*'
  • 'https://www.nwolb.com/help.aspG'
  • 'http://*myspace.com*'
  • '*myspace.com*'
  • 'https://www.gruposantander.es/*'
  • 'http://*odnoklassniki.ru/*'
  • '*odnoklassniki.ru/*'
  • 'http://vkontakte.ru/*'
  • '*/login.osmp.ru/*'
  • '*/atl.osmp.ru/*'
  • '*/my.ebay.com/*CurrentPage=MyeBayPersonalInfo*'
  • '*.ebay.com/*eBayISAPI.dll?*'
  • 'https://www.us.hsbc.com/*'
  • 'https://online.wellsfargo.com/das/cgi-bin/session.cgi*'
  • 'https://www.wellsfargo.com/*'

```

Figure 4.15: Regular expressions extraction in results page.

Chapter 5

Experimental Evaluation

We evaluated Prometheus on a dataset of 53 distinct samples of Zeus (Table 5.1) analyzing 62 real, live URLs of banking websites. However Prometheus is still running and the crawler is continuously feeding our system with new samples.

Our choice to use Zeus as testing family is due to the fact that it is the most diffused banking trojan family and so it is easier to find valid samples. SpyEye is less monitored than Zeus and so it is more difficult to obtain an ample set of recent samples.

The purpose of the evaluation is to measure the correctness of the injections detection process, especially studying how the quality of the results depends on the two most influencing parameters: the number of virtual machines used and the threshold ϵ used to filter out the differences not present in most of the infected VMs.

As for the injection detection process we measure also the correctness of the results extracted by memory analysis.

Finally in Section 5.4.4 we focus our attention also on the evaluation of the performance.

5.1 Deployment

We deployed Prometheus on a 2.0GHz, 8-cores Intel machine with 24GB of RAM running Ubuntu 12.04. We used VirtualBox as virtual machine monitor and each VM was equipped with Windows XP SP3 and Internet Explorer 8. Each VM was configured with 1GB of RAM. The network configuration was set to Host-Only but we add iptable rules to the host virtual network interface to grant Internet access to the VMs.

5.2 Challenges

The main challenge to face was finding enough samples to obtain a good dataset. The principal reason of this is due to the fact that all the banking trojans contact the C&C server and download the encrypted configuration file as soon as they start and, in the case they don't manage to do it, they do not manifest their behavior. This means that for our purpose they became useless because they do not perform any injection. Most of the samples posted on online services as ZeuSTracker¹ remain active for just few hours because when they are detected the C&C server is inserted in a blacklist and blocked. In this scenario we developed a crawler (Section 4.8) to automatically download and submit to Prometheus new samples as soon they are published. Thanks to the crawler we managed to create a sufficiently ample dataset.

5.3 Datasets construction

5.3.1 Samples dataset

The sample dataset used for the evaluation has been created downloading samples from ZeuSTracker and VirusTotal². The dataset is composed entirely by ZeuS samples. This is due to the fact that ZeuS is the most diffused banking trojan family and so it is easier to find samples with the C&C online. SpyEye is less monitored than ZeuS and so it is more difficult to obtain an ample set of recent samples.

We downloaded and submitted to Prometheus 65 samples. However we noticed that 12 samples failed to install because they were able to detect and evade the controlled environment (some recent ZeuS variants implement this feature) or because the executables were corrupted (sometimes corrupted samples are posted on ZeuSTracker). Hence our final dataset is composed by 53 samples and it is shown in Table 5.1.

5.3.2 URLs list creation

We evaluated Prometheus on a list of 62 URLs. The URL list was created starting from a webinjects.txt leaked as part of the ZeuS 2.0.8.9 source code³ and adding some new URLs extracted from initial memory analyses.

¹<https://zeustracker.abuse.ch>

²<https://www.virustotal.com>

³<https://bitbucket.org/davaeron/zeus/>

FAMILY	MD5	DETECTED INJECTIONS	% FPR
Zeus	c07d6b14db7b896942b0e3dcc871488c	40	0%
Zeus	ba8acf19dcfeae5ad362169dcf752fd5	0	0%
Zeus	8240b17d7fb0fe5251b268befddca180	21	0%
Zeus	ba57db487bb18b15217cbb08c923da50	31	0%
Zeus	db028fad3cf758169eac36d361c92a8	32	1.61%
Zeus	df677b3461944899bec4f9d45b0dfe59	21	0%
Zeus	f60ab638881a4b01591b66741fa3d38c	21	0%
Zeus	ca2e8e008cfb74ee8952e9d5a4d03799	0	0%
Zeus	f9d67916f8d348158ba0f6c7f4e22940	59	1.61%
Zeus	e988cf945afcce08ec467bad977db530	21	0%
Zeus	3b997a5a3918b2ae5d7d15ed3b288792	32	0%
Zeus	e5ff769dd2a98c0dd240e176aaef0d2b	21	0%
Zeus	e6e944d021431c826aa9b4d67bed686e	0	0%
Zeus	4bcead4107ed219f4af8a6d206ad3285	32	1.61%
Zeus	9b1da26b16f40ded6a9a45cf7c07e02e	32	0%
Zeus	ff05f92688c9ce5f4d9d9b53cd21484f	37	3.22%
Zeus	73287fe7e25abff78054ba7b0f4621f0	0	0%
Zeus	7081eaf92b6cc54bda51377276d74966	32	0%
Zeus	dd944a21d4fa63b7edd96efc43cef774	31	1.61%
Zeus	e8404581991f8fc218c26525b1b77f3f	32	1.61%
Zeus	49e000294f4d287ee6b758ec72c89657	32	0%
Zeus	ab45ebf631bd68892ee5a02c399c5525	85	0%
Zeus	0f1fc1347d85b5d0b74deb73e4567296	38	0%
Zeus	24921a1f6f7ec496a26e7246808666a9	38	0%
Zeus	2c72564cb24fa1eeb38addf4177d7b4	0	0%
Zeus	fe4cfd09c85b1e529c4af48bd9c04a42	38	1.61%
Zeus	13b2c0c5c5d946a1b42f0e97454e6569	31	0%
Zeus	6d3221e22750a3f017f043eeda3a89a0	31	1.61%
Zeus	5cf9b0d66bc7fdba0053ca3b55a6de6c	38	0%
Zeus	c4a848a1b6aeda6b36bd21fc3e4c2c08	31	1.61%
Zeus	2226e7cfae79ba763d35424435f7c2c6	38	0%
Zeus	a55ac78eca45212d4947209914a930fb	38	0%
Zeus	de491974dee9a33d2278b50f01f03b04	32	1.61%
Zeus	06db9d5e4030da3a3d84d9d644a60734	31	0%
Zeus	90f4d535c3b79d8ed7c67ba38fb06ac6	31	0%
Zeus	f55f44678a68ea615b997585a8d72cb2	31	0%
Zeus	aaa4cc8e0b9584e96f63d843ba972183	38	0%
Zeus	27563287dceddfc5588d02762bed7dac	32	1.61%
Zeus	4cb596ab068dd8f29e510afef91e2ecc	38	0%
Zeus	7cc9351fa29b6a7a44a48e420c867f11	38	1.61%
Zeus	59cbbd1a374d12e5063b75ac7b338f26	38	0%
Zeus	c78a6aeecaf14f7e4ce54ee8264a9fb4	32	0%
Zeus	40ed6f385f4665537a9f401621deb2c2	38	0%
Zeus	470dff32fb454e1c13fcfe8fd9f7630a	32	1.61%
Zeus	5d873312d764a3ae77c6cb408bfe51e5	32	0%
Zeus	41098823375ae77d85ed29658a8d3be5	32	0%
Zeus	717c8ed361d724754df8ccdbe0f09b53	0	0%
Zeus	fbef17f3212de2cabb91eee41c575999	32	1.61%
Zeus	45a1e07d0a66b27a337f0f2e0e223bd3	32	0%
Zeus	6451caab830185967cceece215c76c13	32	0%
Zeus	144afb85337981169cdbcb23300b18c4	31	1.61%
Zeus	dd838e7033debab5bdc51004a78ca8f3	32	0%
Zeus	0d7b5bb7d32f4db92f822b3a3605d128	32	1.61%

Table 5.1: Samples dataset. For each evaluated sample the Table shows the number of injections detected (false positives excluded) and the False Positive Rate of the analysis. The configuration the results refers to was: 12 VMs (6 clean VMs and 6 infected VMs), 40 seconds as sleep time and threshold $\varepsilon = 0.8$

However most of the URLs contained in the initial list have been removed because they presented one or more of the following problems:

- the URL was unavailable. Since the leaked `webinjects.txt` was quite old some of the URLs were not still active;
- Internet Explorer 8 always crashed or froze when the website was visited;
- the web-page was too heavy to be loaded in Internet Explorer 8 and it took too much time to download the page;
- the URL was a post login page that required user authentication.

The final URL list that has been evaluated was composed by 62 distinct URLs.

5.3.3 Ground truth

Finding a way to measure the quality of the evaluation results was not an easy task since to determine the real injections performed by a sample we should have decrypted and checked his own WebInject configuration file.

We initially created a controlled botnet and we build a Zeus sample with a customized `webinject.txt`. However we wanted to test Prometheus on real samples found in the wild. Therefore we decided to manually analyze the results provided by Prometheus to ensure if false positives and false negatives were found.

As regards false negatives, as explained in Section 5.4.2, the only cause of false negatives for our system is the non activation of a sample in some of the infected VMs. However this did not happen during the dataset evaluation.

As regards false positives, two further methodologies were applied in order to measure the false positive rates. First of all we submitted to Prometheus some known benign executables. In this way we were sure that ideally no differences should have been detected and each difference was a false positive. Secondly we exploited the knowledge base created by memory forensic analysis to validate the results as follows: if a difference is detected on a URL that does not match any of the regexs extracted from infected memory dumps it is a false positive. This reduced the volume of data to analyze manually.

In our evaluation we considered a granularity at the URL level and not at the difference level. We are interested in signaling an URL as injected or not independently from the number of injections found. Using this approach

DOMAIN	AVERAGE NUMBER OF INJECTIONS
ybonline.co.uk	10.244
cbonline.co.uk	9.723
lloydstsb.com	7.482
bbvanetoffice.com	4.275
banesto.es	1.620
gruppocarige.it	1.121
scrigno.popso.it	0.916
isideonline.it	0.861
wellsfargo.com	0.861
uno-e.com	0.747

Table 5.2: Most injected websites

we have only two possible case: zero detected differences, that means a clean URL, and a higher than zero detected differences, that means an injected URL.

5.4 Experiments

Prometheus correctly (true positives) detected web injections in the 17.74% of the URLs analyzed. Table 5.2 shows the most injected URLs and their average number of injections. What we noticed is that most of the samples have many common injections and the targeted websites are often the same. We performed the tests changing the number of VMs used and the threshold ε . Our results indicate that, as expected, the heuristic based on the ε threshold (Section 3.3.3) did not generate any false negative even for high value of ε while it helped to filter out false positive differences. We determined that 12 VMs are enough to have good results, which means almost zero false positives. However in order to lower false positives the number could be increased, even if this will slightly worsen the performance.

5.4.1 False positives discussion

A false positive occurs when Prometheus detects a benign difference in a web-page and classifies it as a malicious injection. As showed in Table 5.1 Prometheus produced, with some exceptions, no false positives analyzing most of the samples in our dataset. The overall average of the False Positive Rate is a low 0,52%. We manually observed that the vast majority of the remaining false positives were caused by JavaScript-based modifications. Most of the modern websites contain JavaScript code that modifies the DOM of

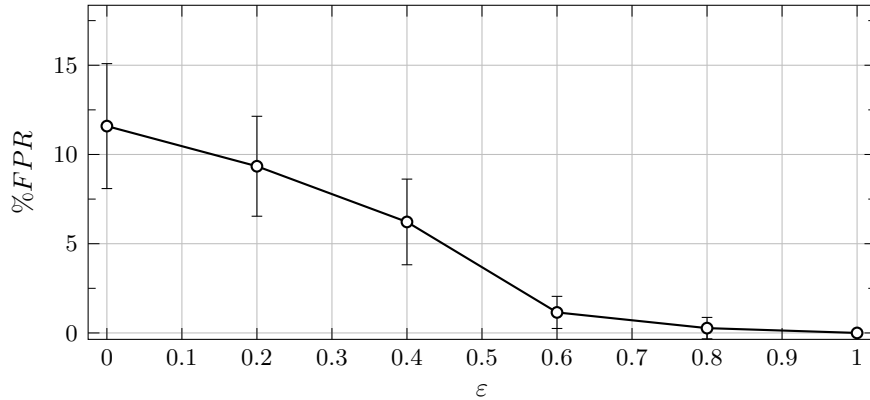


Figure 5.1: False Positive Rate depending on the ϵ threshold evaluating 62 distinct URLs and using 12 VMs (6 clean ones and 6 infected ones) for the analysis of ZeusS 59cbbd1a374d12e5063b75ac7b338f26.

the web-page at runtime loading dynamically changing content. Furthermore sometimes server-side scripts generate different JavaScript code every time the page is requested (e.g., advertisement). Another cause of false positives is the failing in dumping DOMs. Sometimes happened, during the evaluation, that some of the VMs did not manage to load a web-page and dump its DOM within the timeout, due to temporary network slowdown or overloading. This decreases the number of DOMs to be considered in the filtering phase and can generate false positives, as we want an high number of different DOMs in order to be able to discard all the legitimate differences. In any case most of remaining false positives are easily distinguishable looking at the results extracted by memory analysis, since a difference detected on a URL that do not match any of the extracted regular expressions is certainly a false positive. We preferred to leave this check to the users, instead of automatically discarding the differences relating to URLs that do not have references in memory, because this procedure could generate false negatives if the memory analysis fails in extracting some regular expressions. The results presented in Table 5.1 refer to a configuration with 12 VMs (6 clean ones and 6 infected ones) and threshold $\epsilon = 0.8$. We analyzed the effects of the number of VMs and of the threshold ϵ on the false positives (two examples are shown in Figure 5.1 and Figure 5.2). We observed that the number of VMs is the most effective parameter in the reduction of false positives. However since the number of VMs required to guarantee good results is high, at least 24, and this can worsen the performance because of the overloading, using an high value of ϵ helps in guaranteeing such results with a lower number of VMs (10-12).

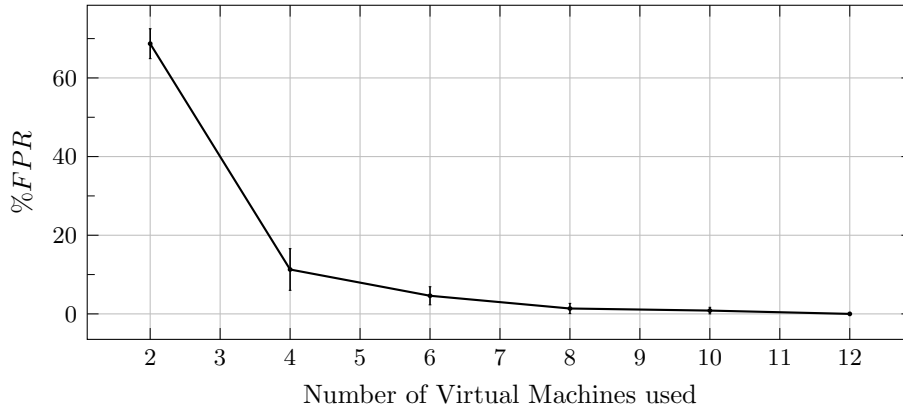


Figure 5.2: False Positive Rate depending on the number of VMs used for the analysis of `ZeuS 59cbbd1a374d12e5063b75ac7b338f26` on 62 distinct URLs with threshold $\varepsilon = 0.8$.

5.4.2 False negatives discussion

A false negative occurs when Prometheus does not detect a malicious injection in a web-page. Since the DOMs comparison process is deterministic there are only two cases in which this can happen. The first one occurs when an high threshold is used and a sample fails to execute in some of the infected machines. As explained in Section 3.3.3 the differences that are not present in most (depending on ε) the infected machines are filtered out, so if a sample manifests his behavior just in few of the infected machines his injections might not be detected. However this scenario never happened during the evaluation and all the evaluated samples succeeded to install and execute in all the machines or in none of them. The second case is related to the ε threshold too. Since the heuristic used is based on the assumption that each sample injects a static content, if a sample injected different dynamic content on the same web-page every time it is executed it would evade our system. Even this case never happened during evaluation. Our assumption was right because the content of each injection is statically specified in the WebInject configuration file.

5.4.3 Memory analysis discussion

Prometheus successfully extracted the RC4 keys and the entire WebInject targets list from all the samples in our dataset. Despite the average number of targets for sample is about one hundred, the total number of distinct regular expressions we extracted by memory analysis is only 203. That means that most of samples have almost the same WebInject configuration file.

REGULAR EXPRESSION
http://vkontakte.ru*
odnoklassniki.ru/
https://www.paypal.com/*/webscr?cmd=_account
https://ibank.barclays.co.uk/olb/x/LoginMember.do
https://www.bbvanetoffice.com/local_bdno/login_bbvanetoffice.html
https://banesnet.banesto.es*/loginEmpresas.htm
https://online.wellsfargo.com/signon*
https://ibank.barclays.co.uk/olb/x/LoginMember.do
https://home.ybonline.co.uk/login.html*
https://scrigno.popso.it*
https://www.gruppocarige.it/grps/vbank/jsp/login.jsp
https://home.cbonline.co.uk/login.html*
*.ebay.com/*eBayISAPI.dll?*
https://bancaonline.openbank.es/servlet/PPProxy?*
https://www.us.hsbc.com/*
https://www.gruposantander.es/bog/sbi*?ptns=acceso*
https://online*.lloydstsb.co.uk/logon.ibc
https://www.uno-e.com/local_bdnt_unoe/Login_unoe2.html
https://www.isideonline.it/relaxbanking/sso.Login*
*//money.yandex.ru/index.xml

Table 5.3: Most found regular expressions

What it is important for us is that the true injections detected were present in URLs that matched some regex extracted. This is a further proof that can help distinguish real injections from false positives.

Another important finding that emerged from the results is that not all the regular expressions imply injections. There are some cases in which some URLs were not injected even if they were present in the memory-extracted targets list. The cause of this fact could be that the injections failed because the hooking points configured in the WebInject configuration file were wrong or old (because the web-page changed and do not contain that HTML code anymore) or simply the sample just monitors the URLs stealing the data submitted by the victim without injecting new contents.

In conclusion, our results show that combining both the web-page differential analysis and the memory forensic inspection is important in order to guarantee low false positives and low false negatives.

5.4.4 Performance

We measured the execution time of Prometheus. Prometheus has been designed and implemented trying to parallelize all the computations. Furthermore the approach used to perform DOMs comparisons is asynchronous and all the computations are executed in parallel as soon as the required data is available. For this reason the Prometheus execution time is dominated by

the time required by the VMs to sequentially visit each URL and dump its DOM, while the time required to compare DOMs and filter the differences is irrelevant.

Prometheus performs a sample analysis on 62 URLs using 10 VMs in about 6 minutes. The sample analysis includes also the memory forensic inspection that is performed in parallel and requires less time than the webpage differential analysis. However, while we have considered all the URLs during the evaluation phase, our idea is to reduce the number of URLs to be evaluated on each sample analysis. Since the Prometheus execution time scales linearly with the number of URLs processed this will bring to faster analysis. At the same time this will not limit the capacity of Prometheus because, as explained in Section 3.2.2, it will process the URLs that have the highest probability to be injected.

Prometheus is designed to exploit all the available resources. As shown in Figure 5.3 Prometheus is able to process each URL in little more than 4 seconds when 2 VM are used. The picture shows that Prometheus scales well increasing the number of VMs with just a little overhead. However when the number of VMs is higher than 10 the overhead slightly increases. This is due to the overload on the single physical machine. Moreover increasing the number of parallel VMs the time required to dump a DOM increases and some of the DOMs fail to be dumped within the timeout. This is due to the fact that all the VMs network traffic flows through the single virtual interface between VirtualBox and the host OS. This problem could be solved deploying Prometheus on the cloud to exploit the maximum scalability allowing it to scale directly with the amount of resources and avoiding bottlenecks.

Figure 5.4 shows the trade-off between performance and false positive rate depending on the number of VMs.

We measured also the amount of memory required by Prometheus. Prometheus required at most 15 GB for a sample analysis using 13 VMs, each of them set with 1 GB of memory. However, since VirtualBox allocates the entire amount of memory assigned to the VMs, we inspected the VMs from their internal and we measured that each VM requires about 300 MB.

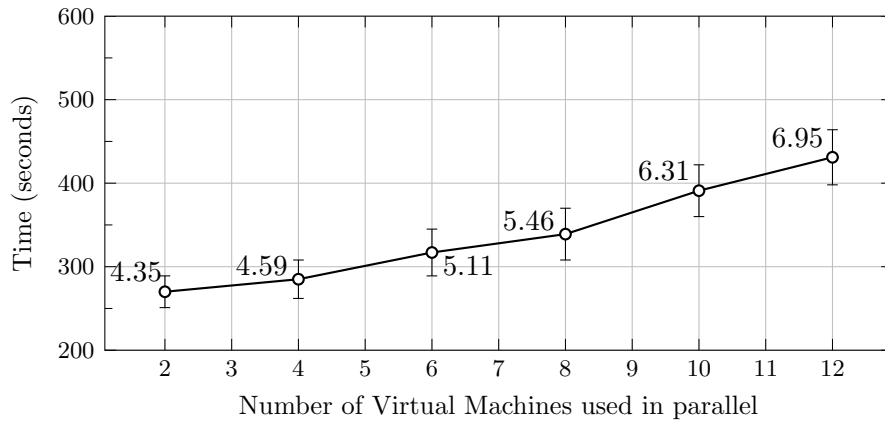


Figure 5.3: Speed and Scalability of Prometheus: Mean time required to process 62 URLs for each sample. The labeled points indicate the mean time required to process a single URL.

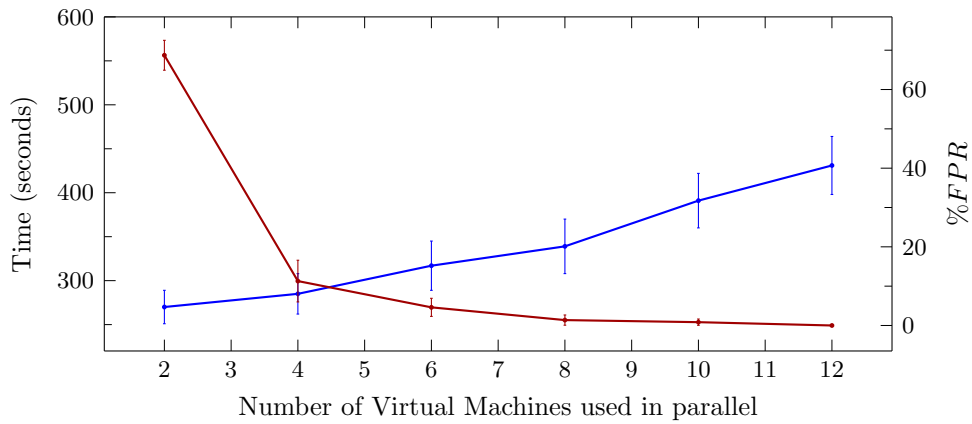


Figure 5.4: Trade-off between Performance and False Positive Rate. The blue line refers to the execution time required to process 62 URLs. The red line refers to the False Positive Rate for the analysis of Zeus 59cbbd1a374d12e5063b75ac7b338f26 on 62 distinct URLs with threshold $\varepsilon = 0.8$.

Chapter 6

Conclusions

We have presented Prometheus, a web based platform, which combines different dynamic malware analysis techniques to offer an automated analysis of those malware which steal information via man in the browser attacks. Prometheus compares the different DOMs retrieved by VMs when they are infected with a malware and when they are not. These differences are then filtered via some heuristics in order to filter out legitimate ones. This technique do not rely on any implementation detail of the analyzed malware and so can successfully detect any kind of WebInject based malware.

Differently from previous work in the field, our system combines this webinjection detection analysis with a memory forensic inspection. In this way our system offers to the users more useful information on the malware they submitted and retrieves some hints on the targeted URLs trying to make every times new analyses more effective. These features are offered through a web interface that allows users to submit and analyze samples or URLs and to get dynamically the results of the analyses avoiding any kind of reverse engineering effort.

Furthermore, we created a crawler which automatically submit new samples to Prometheus every time they are tracked and published on ZeuSTracker or SpyEyeTracker. In this way our sample base is always up to date and most of the times analysts do not need to upload their sample but only to look at the results of previous analyses.

We evaluated Prometheus on a dataset of 53 distinct samples of ZeuS analyzing 62 real, live URLs of banking websites. The results show that Prometheus correctly detected the injections performed by the analyzed trojan with a low fraction of false positives (0.52%) mostly due to the dynamic content generated by JavaScript code. However, while we have considered

all the URLs during the evaluation phase, our idea is to reduce the number of URLs to be evaluated on each sample analysis. Since the Prometheus execution time scales linearly with the number of URLs processed this will bring to faster analyses. At the same time this will not limit the capacity of Prometheus because, as explained in Section 3.2.2, it exploits the knowledge base gained by memory forensic analysis to select and process the URLs that have the highest probability to be injected.

Prometheus is able to process a single URL in about 6 seconds and the analysis of a sample processing 62 URLs requires about 6 minutes. Furthermore Prometheus has been designed to scale directly with the amount of available resources and the designed architecture hypothetically allows to increase the number of analyses conducted simultaneously.

In the next sections we will introduce the limitations 6.1 that we put to our work and the future works 6.2 that may be conducted starting from this limitations. Other future works that we propose are raised during the development phase in which we noticed some lack of technology or some very useful improvements that could be done on the tools that we used.

6.1 Limitations

One limitation of Prometheus is represented by the assumption on which some heuristic-based filters have been implemented. As we said, some of the filters assume that the content of the injections performed by WebInject based information stealers is static, that means that the injections performed by the same malware present always the same content. The assumption holds for all the samples we analyzed but if in the future new samples will perform dynamically changing content injections, these filters will have to be disabled and new heuristic will have to replace them to keep low false positive rates.

Another limitation is represented by the RC4 keys extraction mechanism. Since the mechanism we use to extract RC4 cryptographical keys is depended on the malware implementation it could fail to correctly extract cryptographical keys from some banking trojan version. Furthermore new banking trojans began using AES to encrypt their configuration files and their connections. For this reason it will be useful to improve our key extraction mechanism.

Another obstacle that Prometheus has to face are evasion mechanisms employed by the malware to fool dynamic analysis. Furthermore the actual version of Prometheus is not able to distinguish between those samples which evade the sandbox and those ones which are simply inactive. This

problem could be solved working on bare metal and adopting the method proposed by [20] to obtain a virtual-machine-equivalent snapshots on physical hardware.

6.2 Future Works

As mentioned in the previous chapter Prometheus' performance could be improved. Due to its design Prometheus is very easy to scale, in fact it will be sufficient to deploy the web server, the database, Cuckoo and the back-end on different machines in order to guarantee better performances. Moreover all this system could be deployed on a cloud with some adjustments to make it automatically scale upon the needs of the current workload. In this scenario Prometheus could be improved to handle more than a single analysis simultaneously.

A second improvement could be done in the cryptographical keys extraction phase in order to make it less relying on the actual Zeus implementation and to extract also AES keys.

Actually, for the population of our URLs database we rely only on users submission. It will be very useful to have a tool which starting from our regular expressions database searches in the web for valid URLs matching these expressions and automatically submits them to our system. In addition, we noticed that a significative number of URLs extracted from infected memory dumps are post login pages. Hence designing a solution that allows Prometheus to detect injections even in this scenario will contribute to improving Prometheus' effectiveness.

A further improvement will be to detect and reveal to the users which samples are evading our sandbox and suggest a manual analysis. In order to do that it will be useful to extend the analysis conducted by Lindorfer et al. [24] to Cuckoo sandbox in order detect major points of failure of Cuckoo and improve them.

In addition our system could be used to conduct an interesting analysis observing how webinjects vary in respect with the malware families and so could be used to generate signatures describing different malware. In particular these signatures could be used by antivirus to detect if a certain machine has been infected by a given sample simply looking at how the content of certain web pages is presented and comparing it with our results.

Furthermore it will be very useful to make our system interact with some of the most diffused malware analysis platform (VirusTotal [28], Cybercrime [29] etc.) in order to automatically publish on their pages our results, interacting with their APIs in order to make results more reachable by analysts' community.

Bibliography

- [1] Abuse.ch. Spyeye tracker. <https://spyeyetracker.abuse.ch/>.
- [2] Abuse.ch. Zeus tracker. <https://zeustracker.abuse.ch/>.
- [3] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michael van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. Measuring the cost of cybercrime. In *WEIS*, 2012.
- [4] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2007.
- [5] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.
- [6] Zheng Bu, Pedro Bueno, Rahul Kashyap, and Adam Wosotowsky. The new era of botnets. *White paper from McAfee*, 2010.
- [7] Armin Buescher, Felix Leder, and Thomas Siebert. Banksafe information stealer detection inside the web browser. In *Recent Advances in Intrusion Detection*, pages 262–280. Springer, 2011.
- [8] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.
- [9] Graham Cluley. Corkow, the lesser-known bitcoin-curious cousin of the russian banking trojan family. <http://www.welivesecurity.com/2014/02/11/corkow-bitcoin-russian-banking-trojan/>, February 2014.

-
- [10] Claudio Criscione, Fabio Bosatelli, Stefano Zanero, and Federico Maggi. Zarathustra: Extracting WebInject signatures from banking trojans. In *Twelfth Annual International Conference on Privacy, Security and Trust (PST)*, volume (to appear), Toronto, Canada, July 2014. IEEE Computer Society.
- [11] Stephen Doherty, Piotr Krysiuk, and Candid Wueest. The state of financial trojans 2013. *Luettavissa*: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_state_of_financial_trojans_2013.pdf. *Luettu*, 4:2014, 2013.
- [12] Nicolas Falliere and Eric Chien. Zeus: King of the bots. *Symantec Security Response* (<http://bit.ly/3VyFV1>), 2009.
- [13] FBI. The fraud scheme. <http://www.fbi.gov/news/stories/2010/october/cyber-banking-fraud>.
- [14] JQuery Foundation. JQuery. <http://jquery.com/>.
- [15] Max Goncharov. Russian underground 101. *Trend Micro Incorporated Research Paper*, 2012.
- [16] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 821–832. ACM, 2012.
- [17] Twitter Inc. Bootstrap. <http://getbootstrap.com/>.
- [18] IOActive. Technical white paper. reversal and analysis of zeus and spyeye banking trojans. Technical report, 2012.
- [19] Loucif Kharouni. Automating online banking fraud. Technical report, Technical report, Trend Micro Incorporated, 2012.
- [20] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM, 2011.
- [21] Brian Krebs. Operation trident breach. <http://krebsonsecurity.com/tag/operation-trident-breach/>, 2011.

-
- [22] Brian Krebs. Police arrest alleged zeus botmaster "bx". <http://krebsonsecurity.com/2013/01/police-arrest-alleged-zeus-botmaster-bx1/>, January 2013.
- [23] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 349–358. ACM, 2012.
- [24] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin Heidelberg, 2011.
- [25] Michael Mimoso. 64-bit version of zeus banking trojan in the wild. <http://threatpost.com/64-bit-version-of-zeus-banking-trojan-in-the-wild/103159>, December 2013.
- [26] Marco Riccardi, Roberto Di Pietro, and Jorge Aguila Vila. Taming zeus by leveraging its own crypto internals. In *eCrime Researchers Summit (eCrime), 2011*, pages 1–9. IEEE, 2011.
- [27] Jerome Segura. Hiding in plain sight: a story about a sneaky banking trojan. <http://blog.malwarebytes.org/security-threat/2014/02/hiding-in-plain-sight-a-story-about-a-sneaky-banking-trojan/>, February 2014.
- [28] VirusTotal Team. Virustotal. <https://www.virustotal.com/>.
- [29] Xylitol. Cybercrime. <http://cybercrime-tracker.net/>.

Appendix A

The Database

All the data produced by our system are stored in a single central MySQL database, whose structure is proposed in Figure A.1. Every component has direct access to the database. The database is composed by 6 tables: 5 representing entity and one representing a relationship:

- 1) *sample*: This table indexes all samples retrieved during the analysis, storing the unique id (that identifies also the sample analysis), the name (hash MD5 of the sample), timestamp of submission and the information retrieved from the memory forensic analysis.
- 2) *url*: This table stores all the URLs used by our system with a couple id-URL; the id is unique and identifies also the submission analysis of the URL.
- 3) *regex*: This table is a collection of the regular expressions retrieved during the memory analyses. It contains the regular expressions with their unique id and the sample with which the machine was infected when the system retrieved the regular expressions.
- 4) *domDump*: This table indexes all the DOMs retrieved by our system during the analysis keeping a reference to the sample and URL that has generate them. All the DOMs are stored in files named $\langle dom - id \rangle$ *.dom* in a directory called *doms*, inside the web directory.
- 5) *diff*: This table indexes the differences generated by our system during the analysis keeping a reference to the sample and URL that has generate them. All the diffs are stored in files named $\langle diff - id \rangle$ *.diff* in a directory called *diffs*, inside the web directory.
- 6) *match*: This table represents the match relation between regular expressions and URLs. Every time a regular expression is added to the

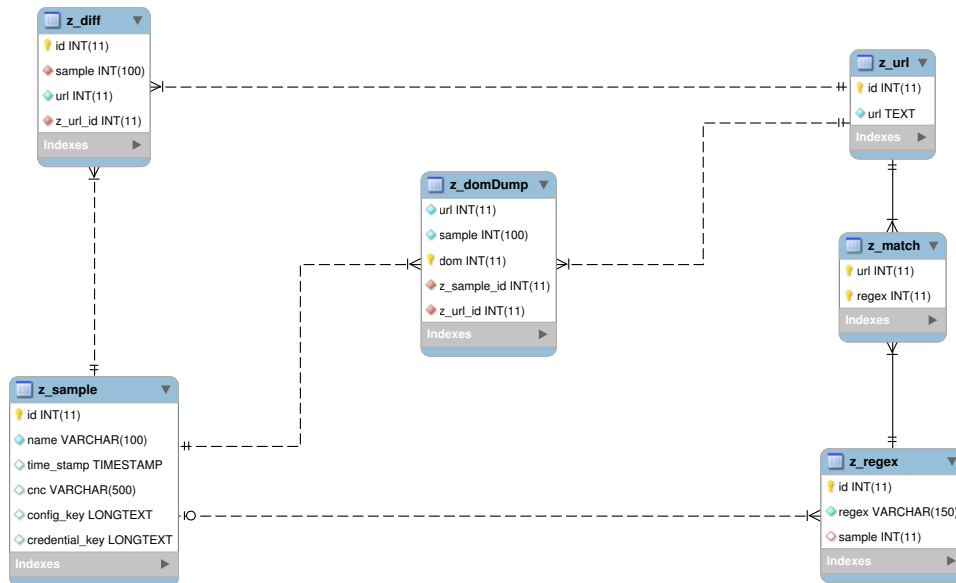


Figure A.1: EER schema of the database.

database every URL is compared with this regular expression and if they match the couple is added to the match table, the same is done when a new URL is added to the database. This table is the key point in the implementation of the URLs ranking (Listing A.1). A similar query is done when VMs require the list of URLs at the begin of a new sample analysis.

```
SELECT COUNT(m.regex) AS m, u.url AS name, u.id AS id
FROM prometheus.z_url AS u JOIN prometheus.z_match AS m ON m.url = u.id
GROUP BY u.id
ORDER BY COUNT(m.regex) DESC
LIMIT 10;
```

Listing A.1: Example Query

Appendix B

Determine Malware Activation Time

Dynamic malware analysis has to take into account many issues, that are due to the fact that malware do not let it be analyzed so easily. The most common technique used for this purpose is the delayed activation of the sample, which means that a sample infecting a machine may activate itself some time after the infection, and the activation time is often randomized. In order to conduct our analysis it is necessary that the analyzed sample is active. For this reason we conducted some experiments in order to estimate an average of the activation time to be used as default sleep time in the configuration file. The VMs will then wait this sleep time before starting downloading DOMs. In order to mitigate the effect of random activation we also employed the threshold for the last heuristics as expressed in Section 3.3.3.

In order to estimate an average time of activation of banking trojans, we build a specific analysis package for Cuckoo. The module works as follow: when the analysis is submitted through the Cuckoo command line interface we specify together with the sample to analyze a sleep time (in seconds); Cuckoo then starts the VM, infects it, waits for the given amount of time and then starts the browser, dumps the memory and stops the analysis. Cuckoo offers some module for results processing in particular the new version offers support for volatility analysis. We exploited a module that shows all the API hooks installed by the malware. In particular we are interested in looking for a hook on the DLL *WININET* at the function *openURL()* (Figures B.1, B.2). In fact this hook is installed by malware to perform man in the browser attacks. Hence when we find this hook it means that the sample has activated himself. Then to estimate the activation time we

Appendix B. Determine Malware Activation Time

PID	Process name	Victim module	Victim function	Hook address	Hooking module	Hook mode	Hook type
412	csrss.exe	KERNEL32.dll	KERNEL32.dll!SwitchThread at 0x7c9329aa	0x7c91dfa0	ntdll.dll	Usermode	Inline/Trampoline
412	csrss.exe	sxs.dll	kernel32.dll!HeapFree	0x7c91ffb0	ntdll.dll	Usermode	Import Address Table (IAT)
412	csrss.exe	sxs.dll	kernel32.dll!HeapAlloc	0x7c9200a4	ntdll.dll	Usermode	Import Address Table (IAT)
412	csrss.exe	sxs.dll	kernel32.dll!GetLastError	0x7c91fe01	ntdll.dll	Usermode	Import Address Table (IAT)
412	csrss.exe	sxs.dll	kernel32.dll!<unknown>	0x7c92135a	ntdll.dll	Usermode	Import Address Table (IAT)

Figure B.1: Table of API hooks in cuckoo results.

1172	explorer.exe	WININET.dll	WININET.dll!InternetCp enA at 0x771ca6dd	0x656e62d8	<unknown>	Usermode	Inline/Trampoline
1172	explorer.exe	WININET.dll	WININET.dll!InternetCp enURLA at 0x771cc8bd	0x656e6a28	<unknown>	Usermode	Inline/Trampoline
1172	explorer.exe	WININET.dll	WININET.dll!InternetCp enURLW at 0x77215a51	0x656e6bfc	<unknown>	Usermode	Inline/Trampoline

Figure B.2: WebInject hook.

repeated our analysis applying a search algorithm (bisection) in order to find with a reasonable precision the instant of activation of the sample. We have repeated these experiments on 6 different active samples obtaining the results that after 40 seconds every sample is active. This result has also been confirmed by the precision of our system achieved during the evaluation.