POLITECNICO DI MILANO
Scuola di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

Zarathustra: Detecting Banking Trojans via Automatic, Platform-independent WebInjects Extraction

Relatore:
Prof. Stefano ZANERO
Correlatori:
Dr. Federico MAGGI
Ing. Claudio CRISCIONE

Studente:
Fabio BOSATELLI
Matr. 746982

Anno Accademico 2011–2012

Table of Contents

List of Figures

List of Tables

List of code and log excerpts

Abbreviations

API Application Programming Interface

BYOD Bring Your Own Device

C&C Command and Control

DNS Domain Name System

DOM Document Object Model

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

SSL Secure Sockets Layer

URL Uniform Resource Locator

USD United States Dollar

VM Virtual Machine

Abstract

Banking trojans are currently the most widespread class of malicious software. They are particularly dangerous because they directly impact the victim's financial resources. Modern banking trojans are distributed as "kits" that anyone can customize. The existence of various customizations, often sold or traded for money, logically lead to a high volume of trojan variants, which traditional approaches based on manual analysis and signature crafting cannot possibly handle.

Modern banking trojans such as ZeuS, SpyEye, or Citadel all have a common, distinctive feature called WebInject, which eases the creation of custom procedures to inject arbitrary content in a (banking) website page. The attacker's goal is to modify the page, typically with additional, legitimate-looking input fields, which capture sensitive information entered by the victim. The result is that a web page rendered on an infected client differs from the very same page rendered on a clean machine. We leveraged this observation to implement a system to generate cross-platform signatures of any arbitrary WebInject-based trojan with no reverse-engineering effort required. These fingerprints can be used to determine whether a client is infected or not.

Our evaluation on 56 distinct ZeuS samples and 213 banking websites shows that our system reaches a good accuracy level and it is able to extract fingerprints from infected clients with a fully-centralized and server-controlled infrastructure.

Sommario

I cavalli di troia bancari, conosciuti come *banking trojans*, sono attualmente la classe più diffusa di software malevolo (*malware*, in inglese). Sono particolarmente pericolosi in quanto compromettono direttamente le risorse finanziarie della vittima. I moderni cavalli di troia bancari sono distribuiti come "pacchetti" che chiunque può personalizzare. L'esistenza di molteplici versioni personalizzate, spesso vendute o commercializzate per soldi, porta chiaramente ad un alto numero di varianti, che gli approcci tradizionali basati sull'analisi manuale e la creazione di *signature* non possono affatto gestire. I moderni trojan bancari come ZeuS, SpyEye o Citadel sono dotati di numerose funzionalità finalizzate al furto dei dati dai computer degli utenti: attraverso le credenziali private, infatti, i criminali informatici possono avere accesso al conto bancario delle vittime. Oltre al furto di dati sensibili, i cavalli di troia consentono anche di controllare le macchine sui quali sono installati, così da variare le tecniche di sottrazione del denaro, rendendo l'attività criminosa più efficace e, al tempo stesso, più difficile da rilevare. Una funzionalità comune e distintiva dei recenti trojan bancari è la WebInject, una tecnica che facilita la creazione di procedure personalizzate per iniettare contenuto arbitrario nella pagina di un sito (bancario). Lo scopo di questa funzionalità è quello di modificare la pagina, solitamente con ulteriori campi di input, che sono apparentemente legittimi, ma che catturano le informazioni sensibili aggiuntive che la vittima inserisce. Il risultato è che una pagina web caricata su una macchina infetta è diversa dalla stessa pagina caricata su una macchina pulita. Siamo partiti da questa osservazione per implementare un sistema che estrae le differenze introdotte in una pagina web da un qualsiasi cavallo di troia che effettua WebInject, senza adottare alcuna pratica di reverse engineering. Queste differenze, alle quali ci riferiamo come *signature* o *fingerprint*, possono essere utilizzate per determinare se un client è infetto o no.

Il nostro sistema automatizza l'estrazione delle differenze tra i DOM di due pagine HTML, delle quali una è ottenuta visitando il sito con una macchina virtuale pulita, mentre l'altra è ricavata da una macchina virtuale infettata con

ZeuS. Il flusso dell'esecuzione si articola in tre fasi: la prima fase consiste nella raccolta dei dati, cioè dei DOM, che vengono salvati su file. I dati provengono da diverse macchine non infette, che indicano come una pagina web dovrebbe comparire in condizioni normali, e da macchine infettate con diversi campioni di banking trojan, sulle quali le pagine potrebbero presentare dei campi iniettati da codice malevolo; durante la seconda fase avviene la comparazione dei DOM ottenuti dalle macchine infette con i corrispondenti DOM ottenuti dalle macchine pulite; la terza e ultima fase elabora le differenze ottenute nella fase precedente al fine di eliminare quelle che, essendo dovute a differenze lecite introdotte lato client o lato server, non sono ascrivibili a ZeuS. Questo viene fatto ricavando le differenze che intercorrono tra più macchine virtuali pulite ed eliminando tali differenze dalla lista di quelle ottenute confrontando una pagina infetta con una pulita. Nella prima e terza fase ci avvaliamo di alcune euristiche con lo scopo di ridurre le differenze legittime che, in quanto tali, costituiscono dei falsi positivi.

La nostra valutazione su 56 diversi campioni di ZeuS e 213 indirizzi bancari mostra che Zarathustra raggiunge un buon livello di accuratezza ed è in grado di estrarre differenze da client infetti attraverso una infrastruttura completamente centralizzata e controllata da un server.

Chapter 1

Introduction

The Internet has become the infrastructure of choice for storing, transmitting and using sensitive personal and business information. A large and diverse population of users accesses online banking services, or performs different kinds of electronic financial transactions. Unsurprisingly, endpoint devices such as computers, mobile phones and tablets have become easy targets for cyber criminals, whose current activities include the infection of such devices with malware, targeted to steal sensitive data, or perform fraudulent monetary transactions without the owner's consent.

These "banking trojans" are a widespread, sophisticated threat. The most successful families (such as ZeuS and SpyEye) make use of obfuscation and encryption, as well as multiple advanced techniques to hide in users' systems to grab credentials and perform transactions. A flourishing, complex underground ecosystem supports their development and spreading with configuration kits, web-based administration panels, builders, automated distribution networks, and easy-to-use customization procedures.

Goncharov [9] recently studied the Russian underground market of cyber criminals: he estimated a USD 2.3-billion market, responsible for 18% of the estimated total USD 12.5 billion worldwide cybercrime figure in 2011, according to [4]. In this market, malicious goods are a "service" with a price tag, from distributed denial-of-service attacks to spamming. A spam campaign is particularly cheap, costing down to USD 10 per million of emails. Anybody can easily buy a customized trojan or rootkit, or a malware-building toolkit to create a customized sample. Interestingly, malware authors and their "affiliate" employees offer paid support and customizations, or sell advanced functionality packages that the customers can include in their builds, for instance to add new functionalities, or to target the users of a specific website. The customer can pay on a per-installation basis, with prices depending on the targeted coun-

try: 1,000 infected Russian users, for instance, cost approximately USD 100. This malware-as-a-service (*MaaS*) phenomenon is alarming, as it turns botnets into a commodity, and allows traditional crime gangs to enter the cyberfraud landscape. Unsurprisingly, online banking fraud is one of the fastest growing segments of cybercrime, amounting to just below USD 1 billion.

The goal of these banking trojans, also commonly referred to as "information stealers", is to intercept credentials such as username, password, and second factors of authentication such as PINs or token-generated codes. To do so, as we detail in Section 2.1.1, these malware families rely on web injection components that manipulate and inject arbitrary content into the data stream transmitted between an HTTP(S) server and the user browser. Such modules are placed between the rendering engine of the browser and the network-level libraries. Thus, they are able to circumvent any form of transmission encryption such as SSL, as we describe in Section 2.4. Such manipulations and injections typically result in changes to the document object model (DOM). This is the key intuition behind our work.

We hereby propose Zarathustra, an automated system that detects the activity of banking trojans that perform WebInjects on the client side. Zarathustra extracts the DOM differences by first rendering a banking website's page multiple times in an instrumented browser running on distinct, clean virtual machines. This builds a model of legitimate differences (e.g., due to ads, A/B testing, cookies, load balancing, anti-caching mechanisms). Zarathustra repeats the same procedure on an infected machine and extracts and generalizes the differences, which we call "fingerprints". The fingerprints are generated on dedicated machines, which operate offline, without any interaction with real clients. Our system has the advantage of requiring no reverse-engineering effort: the only requirement is a binary sample of the malware to infect the controlled machine, which is used to identify differences in web pages generated by the malware's web injection techniques.

We evaluated Zarathustra against 213 real, live URLs of banking websites and 56 distinct samples of ZeuS. In all the cases, our system detected the injections correctly. We analyzed the low fraction of false positives (about 1%) and found that most of them were caused by legitimate differences found in the original web pages. These are mitigated by Zarathustra with specific heuristics, which can be safely enabled under certain, realistic conditions, as detailed in Section 4.

We measured the system's performances in terms of time and number of processed URLs and observed that it scales well: it can process 1 URL in less than 3 seconds (the time required to match the fingerprints are negligible) on

our limited infrastructure. Furthermore, as fingerprint generation and matching can be performed independently on samples and URLs, the process is fully parallelizable and capacity scales directly according to available resources. In summary, Zarathustra gives a contribution in the detection of trojans that inject fields in HTML pages as a data-stealing technique. It is not meant to be an alternative to existing antiviruses, as it focuses on a specific feature that not all banking trojans may implement. Its goal is to propose a new approach that leads to their detection, relying on a very lightweight and little invasive methodology.

Chapter 2

Background and state of the art

This chapter provides information about different aspects of banking trojans. First, we talk about infostealers in general, explaining what they do (Section 2.1) and focusing on WebInjects (Section 2.1.1).

Next we concentrate on the ZeuS crimeware toolkit (Section 2.2): after showing how cyber criminals use it for their malicious purposes and how it evolved during its history, we talk about the components in the toolkit and how the malware infects the system.

After this, we detail the hooking mechanism of trojans in Section 2.3.

In Section 2.4 we describe the state of the art, discussing the limitations of current techniques and mentioning what has been done for the detection of WebInjects (Section 2.4.3).

Last, we set the goal of our work and introduce the challenges we faced when we started to work on it (Section 2.5).

2.1 Information-stealing Trojans

State-of-the-art malware is very sophisticated. From a technical point of view ZeuS and SpyEye are a masterpiece of complexity (e.g. encryption, advanced credential grabbers. Figure 2.1 shows the structural complexity of ZeuS); this aspect reveals a mature malware development industry. In this regard, Lindorfer et al. [17] recently measured that these trojans are actively developed and maintained by the authors. Indeed, both malware families live in a complex environment with development kits, web-based administration panels, builders, automated distribution networks, and easy-to-use customization procedures. The most alarming consequence is that anyone can buy a malware builder from underground marketplaces and create a customized sample. Interestingly, cyber criminals also offer paid support and customizations, or sell advanced

configuration files that the end users can include in their custom builds, for instance to extract information and credentials of specific (banking) websites.

Malware families that follow the same approach of ZeuS and SpyEye usually include data-stealing functionalities. For instance, since version 1.0.0, SpyEye features a so-called "FormGrabber" module, which can be arbitrarily configured to intercept the data that the victim types into (legitimate) websites' forms. This type of trojans are often referred to as "infostealers", in jargon. Unsurprisingly, the main goal of money-motivated criminals that rent or operate information-stealing campaigns is to retrieve valid, full credentials from infected systems. Online-banking websites credentials are among the most targeted ones. Typically, these credentials comprise both the usual username and password, and a second factor of authentication such as a PIN or a token. This (one-time) authentication element is normally used only when performing money transfers or other sensitive operations. As a security measure, many banking websites use separate forms, and do not ask for login credentials along with the second factor of authentication.

2.1.1 The WebInject functionality

As of version 1.1.0, SpyEye incorporates the WebInject module, which can be used to manipulate and inject arbitrary content into the data transmitted between an HTTP(S) server and the browser. We said in Chapter 1 that the WebInject module is placed between the browser's rendering engine and the HTTP(S) API functions and, for this reason, the trojan has access to the decrypted data, if any encryption is used (e.g., SSL).

In the case of information stealers, the WebInject module is leveraged to selectively inject the HTML code that is necessary to steal the target information. For example, as shown in Figure 2.2, the WebInject module inserts an additional input field in the main login form of an online banking website. The goal is to lure the victim such that he or she believes that the web page is legitimately asking for the second factor of authentication up front. In fact, the victim will notice no suspicious signs (e.g. invalid SSL certificate) because the page is modified "on the fly" (see Section 2.3) right before display, directly on the local workstation. WebInjects effectively allow attackers to modify only the portion of page they need by means of site-specific content-injection rules. Additionally, at runtime, the malware polls the botnet command-and-control (C&C) server for further configuration options—including new injection rules.

Differently from phishing, which requires the attacker to host and maintain a spoofed web page, WebInjects do not require any external resource. There-

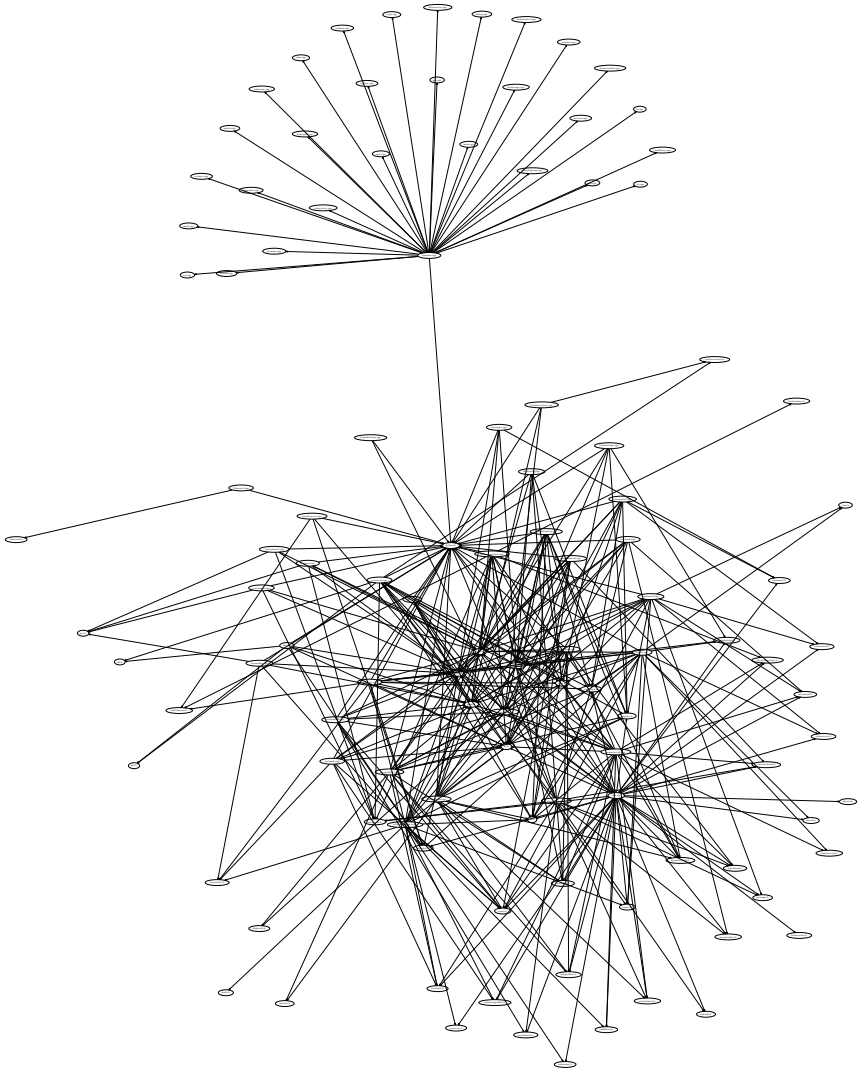

Figure 2.1: The dependency graph of ZeuS gives an idea of its structural complexity.

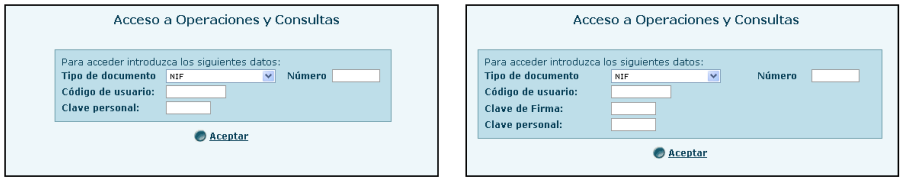

Figure 2.2: Example of a real WebInject found on a page of `extranet.banesto.es`, performed by a ZeuS variant (MD5 15a4947383bf5cd6d6481d2bad82d3b6).

fore, they reduce the upkeep effort for the attacker and also remove a point of failure (i.e., the external web page). Unfortunately, unlike phishing, which is indeed affected by take-down actions [20], the targeted organizations can do little to protect infected clients, because the injection itself is only visible on the client side.

Because of their effectiveness and flexibility, WebInjects have gained a lot of popularity in the underground economy, and contributed to the MaaS phenomenon. The focus thus shifted from malware toolkit itself to the configuration files, which embody the actual value of an information stealer. The content of these files consists in a list of URLs with custom HTML code to inject. The syntax to define WebInjects follows simple rules, as shown in Section 2.2.3. In the case of ZeuS, the WebInjects configuration file is named `webinjects.txt`. Configuration files, and in particular `webinjects.txt` files, are traded[1] or sold[2] on underground marketplaces.

2.2 The ZeuS Crimeware Toolkit

Among the different banking trojans circulating nowadays, the prevailing one is ZeuS [2], also known as Zbot. Its first detection can be dated back to 2007, but it spread massively in 2009, when the first commercial version came out. During its history, ZeuS captured media attention especially when police operations led to the arrest of botmasters that had stolen tens of millions of dollars from bank accounts. One of the biggest operations was accomplished by the FBI along with other international law enforcement in 2010: they discovered a theft ring running ZeuS botnets that allowed felons to steal USD 70 million from companies, towns and churches [1], with a potential loss of USD 220 million. In January 2013, the 24-year old Algerian Hamza Bendelladj was arrested with

[1] `http://trackingcybercrime.blogspot.it/2012/08/high-quality-webinject-for-banking-bot.html`

[2] `https://www.net-security.org/malware_news.php?id=2163`

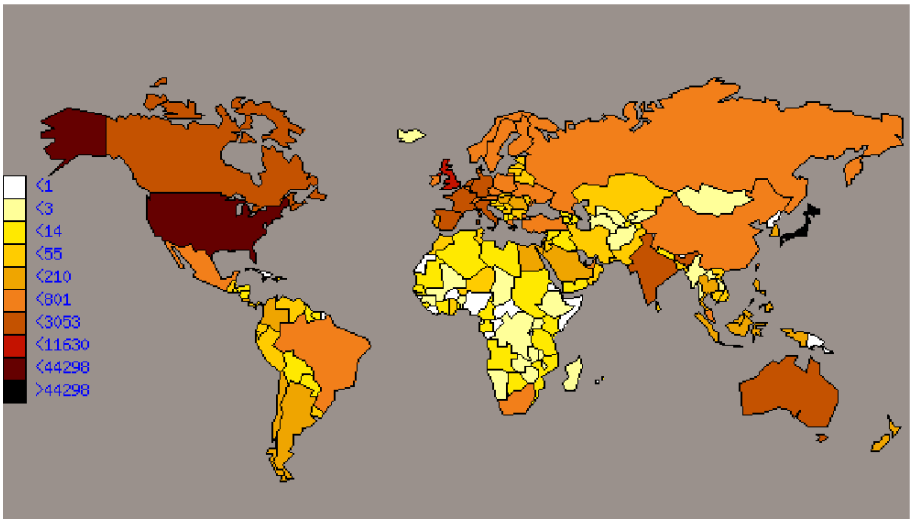

Figure 2.3: The world map showing ZeuS diffusion in 2009 (source: [8])

the charge of having stolen USD 10 to USD 20 million from bank accounts, running a ZeuS botnet [16].

In 2011, ZeuS v2 source code was leaked and this resulted in the creation of several variants, among which Gameover, an evolution of ZeuS that uses P2P communication to send data to botmasters, needs to be mentioned. In September 2012, researchers at F-Secure found that nearly the 10% of Gameover-infected PCs were located in Italy[3].

Today ZeuS can infect many versions of Microsoft's operating systems, including Windows Seven, as reported on the malware's user manual. Supported browsers are Internet Explorer and Mozilla Firefox, but recent variants of ZeuS and trojans that perform web injections also target Google Chrome and Opera[15]. An alarming phenomenon is represented by the increasing diffusion of malware versions for mobile phones: as explained in Section 2.2.2.3, they allow attackers to effectively steal session tokens that banks send users as an additional security measure.

2.2.1 The fraud scheme

The fraud miscreants put in practice comprise different phases and different actors, as shown in Figure 2.4. It all starts from the malware writers that create the code to implement the malware or the toolkit. Once the binary is configured either for third-party customers or for the creators themselves, it is delivered to infect victims. The ways this is done are mainly three:

[3] http://www.f-secure.com/weblog/archives/00002424.html

- The first one is by the so-called "drive-by download": a user is brought to visit a website that hosts malicious content. Without the user's awareness, or consent, a malware sample is delivered and installed on the victim's machine. The ways a user may land on a malicious website are many and vary from a short link the user clicks on, to redirection chains that, starting from a website, bring the user to the final infected website. The exploitation of the browser's vulnerabilities, or its plugins and extensions, can be sold as a service as well. The most popular toolkit to do this is called Blackhole [10].

- The second way to deceive a user to download ZeuS is using spamming campaigns. They consist in a massive dispatch of e-mails to users' mailboxes pretending to be from trusted institutions or websites. The message of these mails usually requires users to login at suitably disguised websites with a graphics that recalls the one of the original website. Users may also be required to run executables that supposedly increase protection measures.Therefore the user deliberately downloads and installs malicious software.

- The third common way ZeuS is installed on a victim's PC or device is using fake tools. A fake tool is an executable that is presented as a benign application, while it actually conceals a malware that executes malicious code. Thus, when a user runs a fake tool, he/she is not aware that a malicious application is infecting the system. In addition to its malicious code, a fake tool may implement the actual functionalities the user downloaded it for.

While the infection keeps on spreading, botmasters exploit their botnet to carry out criminal actions. The stolen money is kept on bank accounts that are not in the criminals' name, but they're property of another actor, called "money mule". Money mules are in charge for withdrawing the cash from the accounts used as destination for stolen money and for moving it to the criminals' real accounts, keeping part of the sum for themselves as a reward. Relying on money mules means adding another layer between the victims and the criminals, making it very hard to identify the mastermind behind a botnet that may involve thousands of infected machines located in different continents.

2.2.2 Evolution

One of ZeuS' strong points has always been its thrust toward new attacking techniques. This continuous evolution was aimed at bypassing the countermea-

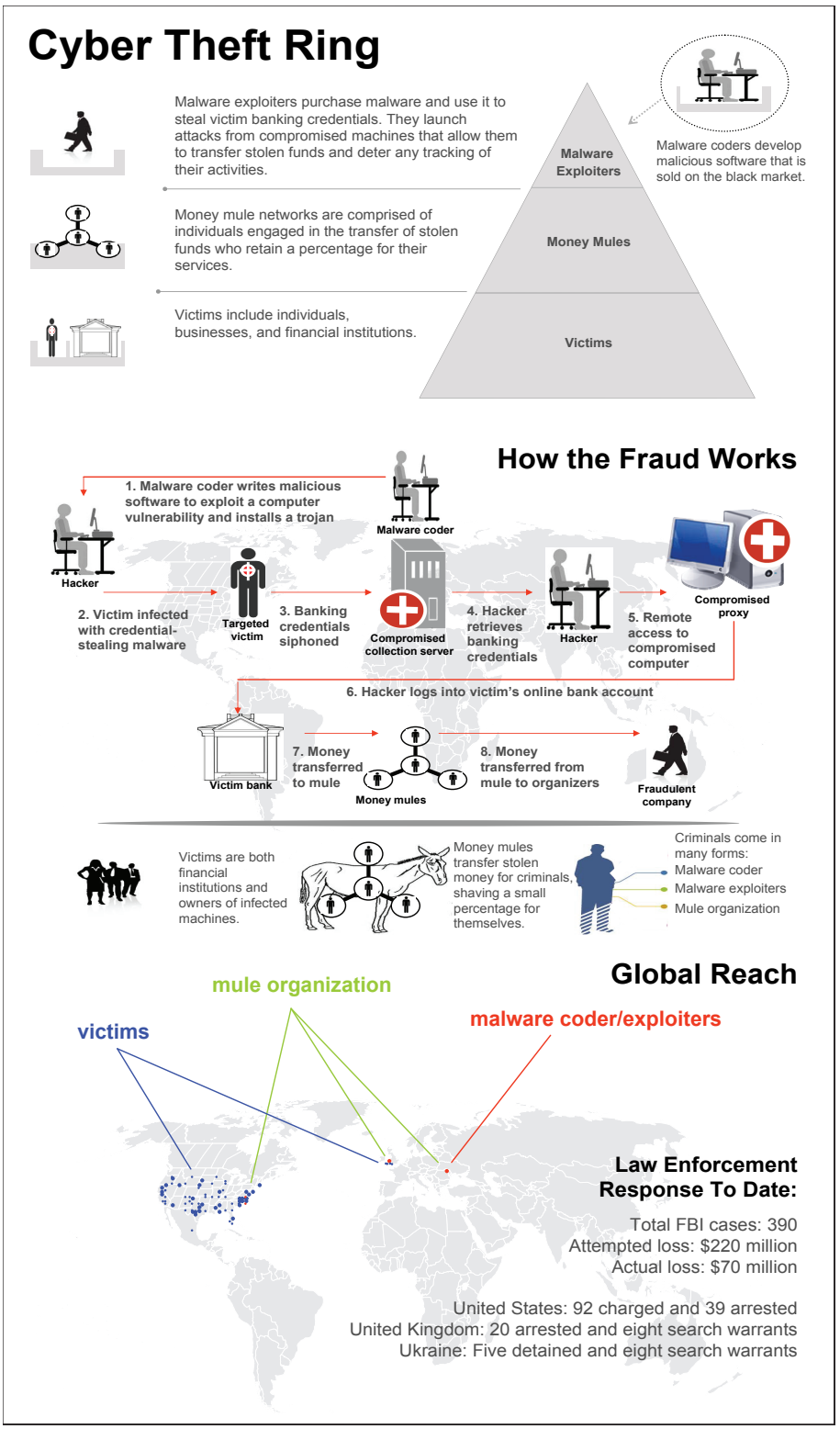

Figure 2.4: The fraud scheme (source: `http://www.fbi.gov/news/stories/2010/october/cyber-banking-fraud`)

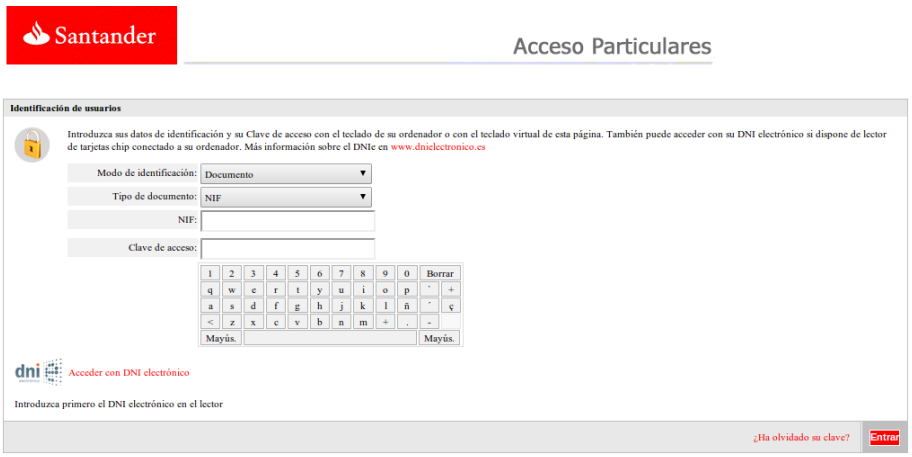

Figure 2.5: A detail of the login page on the website of Banco Santander: the virtual keyboard can be used to insert data.

sures adopted by banks in an attempt to thwart the malware's functionalities.

2.2.2.1 Keylogging and formgrabbing

When it first appeared, ZeuS implemented several information-stealing techniques. The most effective one was keylogging that consists in storing to file each keystroke the user inputs. The file with the recorded keystrokes is then sent to the botmaster. The drawback of this technique, from the point of view of an attacker, is that there are situations in which input data can not be intercepted (e.g., when a user copies and pastes data from a file or when an options menu does not require users to type data on keyboard). Another feature was form-grabbing, already mentioned in Section 2.1.

2.2.2.2 Screenshotting, clickgrabbing and WebInjects

The way banks tried to mitigate the keylogging problem was using on-screen keyboards written in JavaScript: the user was no more required to insert his or her username by typing it on the physical keyboard, but he or she needed to type login credentials by clicking on the letters of a virtual keyboard that appeared on the bank's website. In 2009, ZeuS's authors answered back by enhancing their malware with more advanced attacking modules: screenshotting, clickgrabbing and WebInjects. The first one allows the attacker to take screenshots of the victim's PC when performing login operations. The second one records the clicks the user does, allowing the attacker to know the position of mouse clicks and their sequence. Last, WebInjects, as previously mentioned,

are custom code added to the webpage, and they can be scripts, input fields or whatever element the attacker wishes to add to the page.

2.2.2.3 ZeuS in the mobile

To hinder this new malicious practice, banks introduced two-factor authentication. This type of authentication comprises, along with the normal online registration routine, the use of a one time password (OTP) sent to the customer's mobile phone by SMS. Being only usable once and often within a limited time span, the OTP should protect the user even from web injections: stealing a token that has already been used or that has expired is totally useless. It is under these circumstances that, in 2011, the mobile version of ZeuS made its appearance. Zitmo, an acronym that stands for "ZeuS in the mobile", works in tandem with a PC version of ZeuS and it has been designed to defeat the new authentication method: it can access all the information in the user's phone, including SMS with sensitive information, and send it to its C&C server to complete malicious actions. The diffusion of mobile banking malware is even more dangerous when concerning companies that adopt the bring your own device (BYOD) policy, thus allowing employees to use their devices to do job-related tasks. In August 2012, an attack using the Citadel trojan (a ZeuS variant) targeted the employees of a major international airport to steal credentials for the internal VPN, gaining access to airport applications [14]. Coupling a PC-based version of the malware with one that runs on the mobile is a much more invasive way to steal valid OTPs with respect to the real-time notification feature ZeuS was already equipped with. This last module, an add-on costing $500 [25] in the underground market, sent an instant message through Jabber, so that the botmaster was notified about ongoing operations by the user on a monitored website.

2.2.2.4 The Automatic Transfer System

In 2012 a new threat made the scene: it is the automatic transfer system (ATS). The ATS consists in *de facto* WebInjects, but they're far more sophisticated than the previous ones. It relies on JavaScript code written to perform two operations: one is the automatic transfer of money from the user's account to the attacker's one, the other is the automatic data modification, to deceive the user and make him/her believe that the amount of money has been transferred to the specified recipient. The truth is that the ATS changed both the amount and the recipient on-the-fly without the user even realized it. In some cases,

the ATS takes advantage of a lenient discretionary control of the banks when authorizing and confirming operations.

2.2.3 Components

The ZeuS toolkit is delivered with a user manual that explains how to setup the different components, giving a description of the available features and of the options that can be enabled.

The toolkit includes the following components:

The builder is written in C++ and it is the program that botmasters use to configure the bot and create the executable. The encryption key and other static configuration parameters listed below are hardcoded into the bot executable, so each customer that uses the builder obtains an executable that is different from that of other customers [26]. The builder also encrypts the configuration file that is uploaded to the C&C server. Compiling a custom bot is as simple as clicking on a button.

The configuration file is divided in two parts:

- the StaticConfig part of the file contains the information needed to communicate with the server and directives about what to do during the installation. This information will be embedded into the executable at compilation time. Static data include: a 4-character name to identify the botnet, the time intervals to communicate with the server, the URL from where to download the configuration file, the key to encrypt and decrypt the configuration file and the traffic to and from the server, and instructions about removing the certificates from the victim's storage and disabling the TCP server to prevent warnings from the Windows Firewall. All these data are embedded in the bot when it is compiled;
- the DynamicConfig part of the file stores information that is saved in a separate file, which will be encrypted with the chosen key and provided to the bot when it contacts the C&C. Dynamic data include: the URL where an up-to-date version of the binary is, the URL to send collected data to, the location on disk of the `webinjects.txt` file, used to retrieve WebInjects information to embed in the file when it is created and encrypted, alternate URLs for updated configuration files, the list of websites to monitor with, optionally, regular expressions to identify session parameters to steal data from, fake web pages to redirect requests to when the botmaster enables this

option, DNS servers URLs to hijack requests, a list of URLs from which to steal transaction authentication numbers (TAN), which are a form of OTP used by banks to authorize financial transactions.

The `webinjects.txt` file stores all the WebInjects that have been defined by the attacker, as explained in Section 2.1.1. The definition of a single WebInject rule is very simple:

- `set_url` specifies the URL the WebInject refers to. Special characters (called "masks") and additional parameters can be used to specify under which conditions the injection should be enabled;
- `data_before` specifies the hooking point inside the page, i.e. the HTML code after which the injection is located. The same special characters used to set the URL can be used here as well, so to create regular expressions that increase chances for a successful injection;
- `data_inject` is the actual code to inject;
- `data_after` is another way to specify a hooking point;
- `data_end` is used to close each code part.

Listing 2.1 shows how a WebInject is defined in the `webinjects.txt` file.

```
set_url https://online-offshore.lloydstsb.com/* GP
data_before
To log on, enter your User ID
data_end
data_inject
, Password and Memorable Word.
data_end
data_after
</p>
data_end
data_before
name="Password"*</tr>
data_end
data_inject
<tr align="left">
  <td bgcolor="#ceefe7" align="left" valign="middle">
    <div class="entries3">
      <b>Memorable Word</b>
    </div>
  </td>
  <td bgcolor="#ceefe7"><img src="img/space.gif" width="3"
      height="30" border="0"></td>
  <td bgcolor="#ceefe7">
    <div class="entries">
```

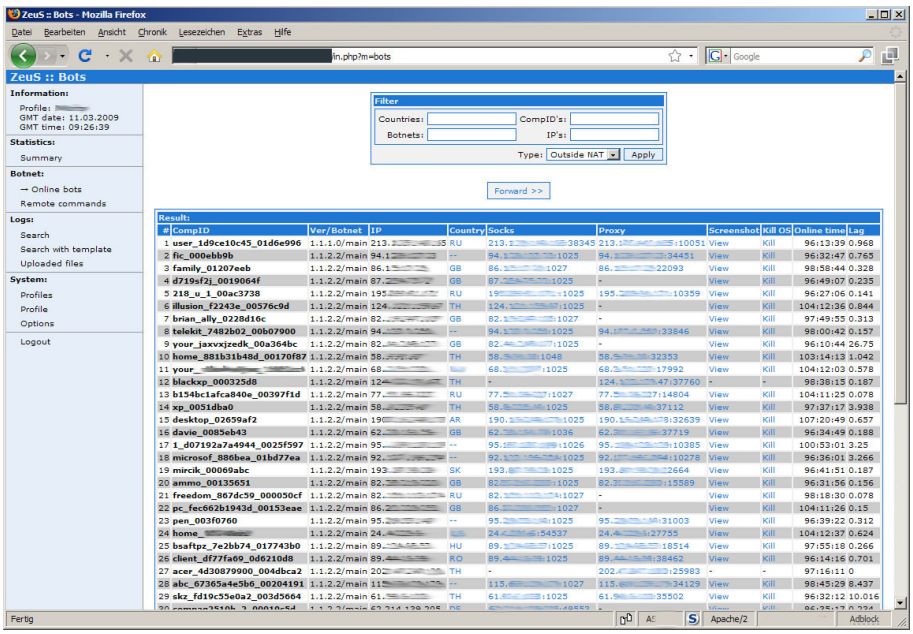

Figure 2.6: A screenshot of the control panel of ZeuS (source: `abuse.ch`)

```
      <input type="password" name="MemorableWord" size="15"
          maxlength="15">
    </div>
  </td>
  <td valign="top" bgcolor = "#ceefe7"><img src="img/space.
      gif" width="1" height="40" border="0"></td>
</tr>
data_end
data_after
data_end
```

Listing 2.1: Example of the definition of a WebInject

The C&C control panel tool is a management backend (written in PHP) that is installed on the C&C server. Through this interface, bot herders communicate with the installed bots, send commands and receive stolen data, which is stored into a MySQL database. The C&C control panel provides:

- Statistics about the botnet, such as connected bots, their OS version and service packs installed.
- A way to interact with the bots, by searching for infected machines that meet some given parameters (e.g., country, online status, host

name) and by creating scripts to launch on the specified machines. Scripts can be easily written using the commands explained in the toolkit user manual and also in the control panel. 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.

- Reports about the activity that bots themselves sent to the server at defined time intervals, including login credentials and screenshots.
- Information about the server system status.

2.2.4 Infection

The installation routine performed during the injection is rather complex:

1. the malware executable creates a directory and copies itself in there. Since ZeuS version 2, both the directory and file names are randomly generated;

2. by inserting registry keys, ZeuS ensures that `winlogon.exe` spawns it at the next startup and lowers the security settings of Internet Explorer;

3. it injects malicious code into other processes (e.g., `winlogon.exe`), creating a thread to execute its custom code, so that the main process can terminate but the malicious code still runs, hooking APIs in the target processes. Then it creates new folders to store the dynamic configuration file downloaded from the server and the stolen information, which will be sent to the server. Since ZeuS version 2, the configuration file is stored in the Windows registry [26];

4. a new code injection in `svchost.exe` is performed: this part of code is responsible for network communications and Internet-related APIs hooking (see Section 2.3). The communication between injected processes is made possible through mutexes and pipes;

5. it harvests credentials (e.g., certificates, cookies, FTP, Windows Mail, Outlook Express;

6. it downloads the configuration file from the server and processes it;

2.3 Hooking mechanism

WebInjects are currently implemented by hooking into the Windows API functions, a technique also adopted by so-called "userland rootkits". To perform WebInjects, malware executables inject control code into the web browser process when it starts (see Section 2.2.4); ZeuS and SpyEye inject malicious code into all the other user processes as well. They hook the API functions in system libraries, in order to intercept APIs calls to filesystem, registry and process control functions [6]. In the case of web browsers processes, ZeuS hooks into the `ntdll.dll` and `Wininet.dll` libraries, which define functions (e.g. `HttpSendRequest`, `InternetReadFile`, in the second case) that are used by browsers to handle web traffic, as summarized in Figure 2.7: this allows to process all the network data, including HTML pages, to and from the browser. We already mentioned in Chapter 1 that hooking network APIs gives access to the information exchanged by the browser before it is encrypted outbound, or after it has already been deciphered inbound.

Trojans adopt different types of hooks:

Inline hooks overwrite the first five bytes of an API function code, redirecting the execution flow to custom code.

Import address table hooks replace the address associated to API functions in the import address table, which is used by processes to retrieve the location of dynamically loaded functions. This table is filled by the Windows loader when an executable is loaded into memory.

Export address table hooks replace the address associated to API functions in the export address table. Each module has its own export address table, which is used to locate the functions that the module dynamically loaded.

Other hooking techniques are used to perform the aforementioned hooks in child processes spawned by infected parents.

A list of typical API hooks is given in Table 2.1.

2.4 Banking trojan detection: state of the art

From the malware analysis point of view, the threat landscape described in Section 2.1 translates into an increased volume of distinct samples. In fact, not only the malware binaries can be packed and obfuscated with a wide array of options (e.g., packing method, encryption key), also the custom configuration

Library	API
ntdll.dll	NtCreateThread (pre Vista)
ntdll.dll	NtCreateUserProcess (Vista and later)
ntdll.dll	LdrLoadDll
kernel32.dll	GetFileAttributesExW
wininet.dll	HttpSendRequest
wininet.dll	HttpSendRequestEx
wininet.dll	InternetCloseHandle
wininet.dll	InternetReadFile
wininet.dll	InternetReadFileEx
wininet.dll	InternetQueryDataAvailable
wininet.dll	HttpQueryInfo
ws2_32.dll	closesocket
ws2_32.dll	send
ws2_32.dll	WSASend
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	RegisterClassEx
user32.dll	BeginPaint
user32.dll	EndPaint
user32.dll	GetDCEx
user32.dll	GetDC
user32.dll	GetWindowDC
user32.dll	ReleaseDC
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	TranslateMessage
user32.dll	GetClipboardData
crypt32.dll	PFXImportCertStore
nspr4.dll	PR_OpenTCPSocket
nspr4.dll	PR_Close
nspr4.dll	PR_Read
nspr4.dll	PR_Write

Table 2.1: Typical hooked APIs divided by library name

files are encrypted, and embedded in the final executable. This characteristic, combined with the evolving nature of modern trojans such as ZeuS, makes it very difficult to create automatic mechanisms that, for instance, extract the configuration files from a sample or, more simply, detect the activity of an infected machine.

2.4.1 Reverse engineering

Both the static and dynamic configuration files are encrypted, and thus cannot be extracted trivially or automatically—beside, of course, through time-consuming reverse engineering efforts, or in the lucky case that the malware itself exposes some vulnerabilities (e.g., SQL injection, weak cryptography). An example of such approaches was presented in Riccardi et al. [22], where the authors leverage a vulnerability of the encryption routines to create a chosen-plaintext attack against the ciphered stream that flows between ZeuS (1.x and 2.x) and its C&C. The chosen plaintext is a combination of the information from the analysis of the malware toolkit and the data collected while running a sample in a controlled environment (e.g. cookies, user credentials, or computer hostname). Other approaches on how to decrypt ZeuS' configuration files have been proposed, like that by Shevchenko [23], where the encryption mechanism introduced in ZeuS version 2 is explained and a tool to perform the decryption routine is made available.

With particular attention to ZeuS and SpyEye, many technical analysis can be found. Among these, Sood et al. [24] give a detailed overview of the components of SpyEye, including its development kit, and describe how SpyEye integrates in the whole criminal ecosystem. Binsalleeh et al. [5] performed a similar study on the ZeuS crimeware toolkit.

On the one hand, such approaches that revolve around an initial, in-depth reverse engineering of a malware binary are useful to identify vulnerabilities or patterns of activities that can be exploited as detection criteria. On the other hand, the generality of these approaches is likely to decrease quickly from one release to another and to drop significantly between different families of malware.

2.4.2 Classic approaches of anti-malware tools

Classic approaches adopted by anti-malware tools, such as the generation of static binary signatures are easily evaded by packing and obfuscation, as shown by Buescher et al. [6, Section 5.2]. Additionally, they are also not very effective on unknown variants or configuration files and this is particularly true

in the case of malware compiled with dedicated toolkits: Wyke [26] explains that ZeuS customers use to pack the executable with their own packing tools in addition to that ZeuS already provides, populating the scene with malware packed through different packing techniques. Dynamic, behavioral signatures seem a more promising research direction, although they present some drawbacks when the malware adopts anti-analysis techniques. For example, a sample may refuse to expose its true malicious behavior when it detects the action of well-known debugging tools or analysis environments. These techniques aim at slowing down, or in the worst case making it impossible, to automate the extraction of signatures to recognize the typical behavior of a malware. There are two research lines that partially mitigate this problem: one consists in finding efficient ways for running the malware on bare metal (see the work by Kirat et al. [13]), thus reducing the chances that the malware can realize that it is being analyzed; another line leverages hybrid static-dynamic analysis techniques to automatically derive polymorphic-resilient static signatures of dynamic behaviors (see the work by Comparetti et al. [7]).

Another recent work on malware analysis, by Lindorfer et al. [17], is particularly related to ours, because, as part of their evaluation, the authors analyze the ZeuS and GenericTrojan families. Their system, which monitors the dynamic behavior and the respective static code for changes, detected an interesting evolution in these two and other families. This finding supports the importance of forward-looking detection methods such as Zarathustra, which are less dependent on the current or past characteristics of the targeted malware.

2.4.3 WebInjects detection

The detection of the WebInject functionality has been tackled by Buescher et al. [6] to identify information stealers. Their key intuition is that WebInjects are currently implemented by hooking into the Windows API functions. The authors analyzed all the possible hooking mechanisms that could be implemented in the Windows OS and, from them, they derived behavioral fingerprints to detect information stealers that work in userspace. Basically, they look for extra code sections in the basic Windows libraries, by comparing the version stored on disk to the version loaded in the process memory. Extra code sections are a sign of code injection due to hooking.

The weakness of this and other previous work based on an initial reverse-engineering of the malware is that the effectiveness of detection is dependent on the version of the trojan, as future versions might change which functions

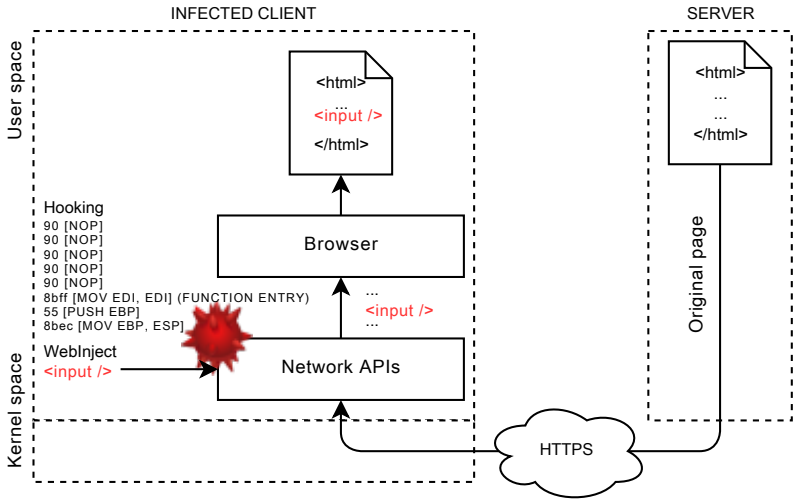

Figure 2.7: The HTML source code produced by the banking website transits encrypted over the Internet. When it reaches the OS and thus the `Wininet.dll` library, the source code is decrypted and intercepted. ZeuS modifies it on the fly and sends it through the same pipeline, up to the browser rendering engine.

are hooked. Additionally, the whole approach is OS and browser dependent. Browsers other than Internet Explorer (e.g. Firefox, which, as described in [3], is already targeted by current versions of this class of malware) use different libraries, and OSs other than Windows have different userland APIs. Indeed, as highlighted in the latest ENISA Threat Landscape [19], the popularity of cross-platform malware increased in 2011–2012 (the most notable example is the Flashback botnet, which was reported that contained more than 600,000 Apple Macs).

The approach proposed by Heiderich et al. [11] protects the browser from malicious websites that perform dynamic changes of the DOM. Although not designed specifically to target information stealers, it could be applied to recognize WebInjects. The system instruments the ECMA script layer by proxying its functions so to profile their execution and recognize malicious patterns. However, the authors mention that their method can detect dynamic changes of the DOM, whereas WebInjects work at the source-code level.

2.5 Challenges and goals

We showed in Section 2.4 the limitations that current detection techniques are subject to: when common approaches loose effectiveness, new solutions need to be found. The goal of our work is to propose a novel approach that relies

on a peculiar feature of some malware families, called WebInject. Working at a high level, i.e., at the top of the browser, allows to abstract our solution from deep system-related aspects, but, at the same time, we don't take advantage of lower level information that possibly reveals incontrovertible evidence of the presence of malware. Moreover, comparing DOMs to extract differences is a conceptually intuitive approach, but, as we explain in Section 3.1, it deals with highly variable documents, making its implementation much harder than it theoretically looks.

It is important to remark that WebInjects detection at the browser-level is not a replacement for antiviruses, but it can help in the detection of trojans when antiviruses fail or when the system is not properly protected.

Chapter 3

Zarathustra

This chapter is devoted to the description of the system. We will follow a top-down approach divided into three sections, adding technical details as we proceed in the dissertation.

Section 3.1 introduces the properties Zarathustra, mentioning some of the challenges in the comparison of website pages.

Section 3.2 explains how the system works in general terms. It introduces the three phases that characterize the workflow and that will guide the description also in the following sections.

Section 3.3 introduces the components that intervene in the workflow, explaining their function at each step of the computation.

Section 3.4 gives technical details about how the above-mentioned components have been implemented.

Last, Section 3.5 suggests possible scenarios to detect WebInjects using the signatures extracted with Zarathustra.

3.1 Proposed approach

Zarathustra recognizes the behavior of any WebInject-based information stealer by looking for evidence of WebInjects in the targeted websites (e.g. an online banking website). Zarathustra does not leverage any system-specific component or vulnerability to observe this malware behavior; consequently, it is flexible and very general. From hereinafter we use the term "WebInject" in its most general interpretation to refer to any mechanism used by malware to inject arbitrary content in the (decrypted) data that transits between the network layer and the rendering engine of a browser (see Figure 2.7). Therefore, our approach is by no means limited to SpyeEye and ZeuS (although we use various ZeuS samples as a use case). It applies to (possibly unknown) software

that may implement an in-the-browser injection mechanism.

To remove the dependence on a specific (version of the) OS, browser, and malware, we lift the detection of WebInjects at a higher level in userland. In particular, we position the observation point of Zarathustra on top of the process space of the browser. Specifically, by generalizing the sample WebInject in Figure 2.2, we observe that the source code of a website rendered on an infected client differs from the source code of the very same page rendered on a clean machine. Performing the comparison between these two different versions allows to extract their differences, hence obtaining the WebInjects in the page on the infected client.

Although this method is in principle quite simple when applied at a small scale (e.g. by manual analysis of a handful of target websites and samples, as shown in an example by Ormerod [21]), streamlining the generation of these fingerprints for detecting information stealers in the large scale presents two main issues:

- websites, by their own nature, vary legitimately. This can be due to several factors, including server-side caching, web-replication mechanisms adopted by Internet service providers or content-delivery networks and, mainly, upgrades of the (banking) web application. As noted by Maggi et al. [18], these changes are very frequent in real-world web applications. The unfortunate side effect is that protection tools based on anomaly models yield false detections, because they confuse legitimate changes with tampered HTTP interactions.

- The rendering of a legitimate web page can vary depending on the inclusion of external resources performed on the client side (e.g. mashups), which is a common practice in websites nowadays (e.g. advertising, asynchronous content).

Both these issues yield false positives (i.e., differences); we address them in four ways as detailed in Sections 3.3.1.1 and 3.3.3.1.

3.2 System overview

The way Zarathustra works is this: it renders a (banking) website in an instrumented browser running on a clean machine; it repeats the same procedure on an infected machine; finally, it extracts and generalizes the differences, which we call “fingerprints”.

Zarathustra performs these operations in three phases:

The DOM collection phase is when Zarathustra visits the webpage on both the clean and infected machines and store their DOMs;

The DOM comparison phase consists in comparing the DOMs obtained from clean machines with those from infected machines, extracting the differences;

The fingerprint generation phase is the last step. All the differences previously extracted are definitively filtered and processed here.

We will now walk through these three different phases, providing a high-level explanation of what Zarathustra does in each of them.

3.2.1 Phase 1: DOM collection

In this phase, Zarathustra processes a list of URLs to analyze and visits each of them on several infected VMs, specified by the user when the system starts. The same list of URLs is analyzed also on a clean VM, which will be used as the reference for the comparison, and on a number of other clean VMs: as explained in Section 3.4.3.2, the reason why we also run different clean VMs is to allow the elimination of legitimate differences that occur between two or more clean VMs.

When this phase terminates, we have one folder for each VM that executed and, inside it, one file for each URL that was visited and successfully dumped by the VM. One file stores the DOM of a single page.

3.2.2 Phase 2: DOM comparison

For each URL correctly processed in phase 1, the comparison stage outputs a list of differences detected when comparing the DOM from the clean VM with the DOM from the infected VM. From this list we perform the first filtering operation, discarding those differences we are not interested into. In this phase, DOMs from clean VMs are compared as well and the resulting legitimate differences are provided as a separate output. The comparison of the DOMs dumped from an infected machine begins when the machine itself terminates the DOM collection phase: this means that while some of the VMs are busy collecting DOMs, some others may be already at the comparison phase, hence parallelizing the workflow for separate machines.

3.2.3 Phase 3: Fingerprint generation

This last stage of the workflow is when the differences extracted from the previous phase are processed and filtered to keep only those that are ascribable to ZeuS.

This phase is extremely important to identify and discard legitimate differences that would otherwise be labeled as actual fingerprints and that would therefore increase the number of false positives.

3.3 System details

This section gives a more detailed explanation of the system: we introduce the components that take part to the different phases; their implementation will be discussed in Section 3.4. Figure 3.1 shows which operations are performed in the different phases. Figure 3.2 gives a graphical representation of what is explained here, showing which components contribute to the different stages of the execution.

3.3.1 Phase 1: DOM collection

When the system starts, a specific folder is checked for ZeuS executables and all the files found are added to a queue. This is done by the `FolderScanner` component. Executables can be put in the folder by hand or downloaded by a crawler that can be optionally started. In the meanwhile, a number of threads in charge for handling the VM, called `VMThread`s, waits for a queue to be filled with the samples found: as soon as one is added, a free thread pops it and begins to process it. In this way each sample is dispatched to one thread. A `VMThread` starts the component that runs on the VM and that interacts with the browser to obtain and store the DOM dumps. This component is the `dumper.jar`. When a thread is done with processing, we obtain a folder with the serialized DOM representations of the URLs that have been analyzed. Before the `VMThread` takes a new sample from the queue, it pushes the just-processed sample into a new queue of completed tasks, hence signaling that the sample can undergo the next analysis step.

3.3.1.1 Heuristics applications in the DOM collection phase

The greatest part of legitimate differences between two versions of the same page, rendered either in different time instants or in different VMs, are due to client and server-side code that dynamically modifies the static structure of the page. To deal with this, we introduced two heuristics:

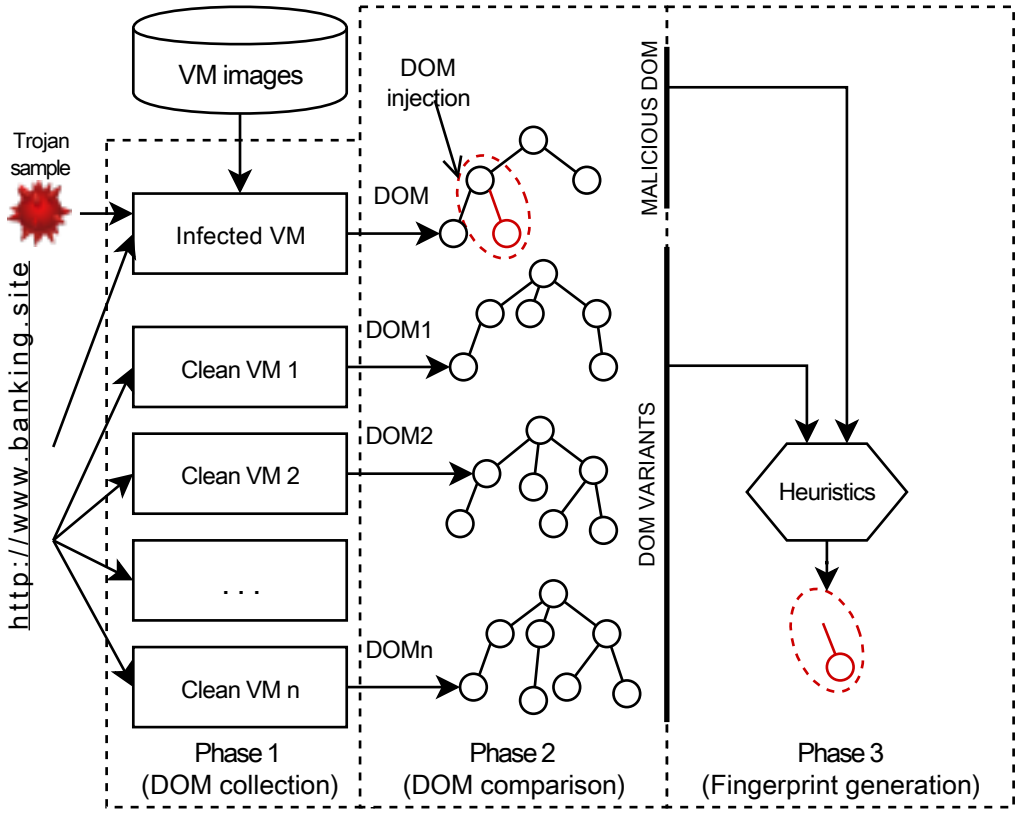

Figure 3.1: Server side architecture of Zarathustra, which is in charge of analyzing a given URL against a given trojan. The detection phase is not visualized: We detail it in Section 3.5.

Heuristic 1: Disabling Client-side Code. Our system can work in "no-script mode" by disabling the JavaScript interpreter. This heuristic helps at reducing the false positive differences due to client-side code manipulations (e.g. advertisement networks). At a first glance, this heuristic may lead to excluding malicious DOM modifications caused by the malware. However, our attacker model considers the WebInjects, which always result in at least one static code injection. As we discuss in Section 5.1, even in the corner case of a malware that injects code inside an existing `<script />` with client-side code that performs the actual DOM manipulation, Zarathustra still detects the text injection in the first place.

Heuristic 2: Caching Server Responses. From our experiments, we observe that the **DOM Comparison** phase needs at least $n = 30$ clean VMs in order to correctly distinguish between legitimate and malicious differences.

However, visiting or rendering the same site n times may not be feasible (e.g. banning, hardware restrictions). By caching the server responses—using the URL as the caching key—we reduce the false positive differences due to dynamic code on the server side, which may insert, for instance, a unique identifier in each response (e.g. to avoid cross-site request forgery or caching).

Most of the work to improve the performances of the system was done disabling these heuristics, as we wanted to alter real-case conditions as little as possible.

Results in Section 4, however, show that the best reliability is achieved when these heuristics are enabled.

3.3.2 Phase 2: DOM comparison

The **DOM Comparison** phase is where the "malicious DOM", and the "clean" $DOM_i \in [1, n]$ are compared and the resulting differences processed at a first stage. It is at this point that the comparer launcher steps in: a variable number of `ComparerThread`s waits for new samples to be added to the queue of completed tasks; each thread pops one element from the queue, reads the name of the associated sample, looks for the folder containing its dumps and starts the `comparer.jar`. This component performs the comparison between the DOMs from the infected VM and the DOMs from one or more clean virtual machines, performing also a first filtering step.

3.3.3 Phase 3: Fingerprint generation

The fingerprint generation part is entirely performed by the `comparer.jar`. These processing and filtering operations are carried out in two consecutive steps, i.e. removal of benign differences, that occur also when comparing clean DOMs, and application of heuristics derived from manual inspection of the results. The output we obtain at the end of this path is a set of JSON files divided by sample. A JSON file stores information about every single difference that has been detected and not discarded as legit. Given that a file is created only for those URLs on which non legitimate differences were found, we may have empty sets for samples that do not supposedly perform WebInjects. When also the fingerprint generation is done and the `comparer.jar` terminates, the `ComparerThread` is freed and goes back to waiting for new elements to fill the queue.

3.3.3.1 Heuristics application in the fingerprint generation phase

We already discussed the use of heuristics during the DOM collection part in Section 3.3.1.1. In the fingerprint generation part we considered useful to adopt heuristics as well. We managed to identify false differences with common properties that often occurred in the output and we arranged the following heuristics (to refer to specific heuristics, the numbering resumes from the list of those in Section 3.3.1.1):

Heuristic 3: Filtering Special Attributes. Several attributes can be safely ignored, because they would not lead to new DOM nodes. We assume that if a malware attempts to forcefully inject a DOM node (e.g. `<input />`) into an attribute value, this would lead to parsing errors, and thus to a useless DOM node. Specifically, we ignore `value`, `style`, `class`, `width`, `height`, `sizset`, `sizcache`, and `alt`. The `style` atribute may be used maliciously, to inject JavaScript code. However, **Heuristic 1** prevents this case.

Heuristic 4: Filtering Text Nodes. We ignore all the text nodes, unless they are children of `<script />` tags. Text nodes are harmless, because they can only contain pure text.

3.4 System implementation

This section gives a technical explanation about how the components introduced in Section 3.3 are implemented. We will do so after a brief overview of the libraries we used to develop our system.

3.4.1 Libraries and tools

The languages we used to implement Zarathustra are two: Python and Java.

We used Python to automate and optimize the workflow. The main goal of this part is to take care of everything that deals with the virtualized environment, that is creation, snapshotting, starting and powering off the VMs, files transfer from and to the host operating system, programs execution on the VM. This part was implemented on top of the virtualization software package Oracle VirtualBox.

Java was used to implement the two JARs we mentioned in Section 3.3, i.e. the `dumper.jar` and the `comparer.jar`. Among the Java libraries Zarathustra uses, there are two that are particularly important: WebDriver and XMLUnit.

WebDriver is part of a suite of tools by Selenium. We employ it to start the browser, load an URL and get the source code as processed by the browser

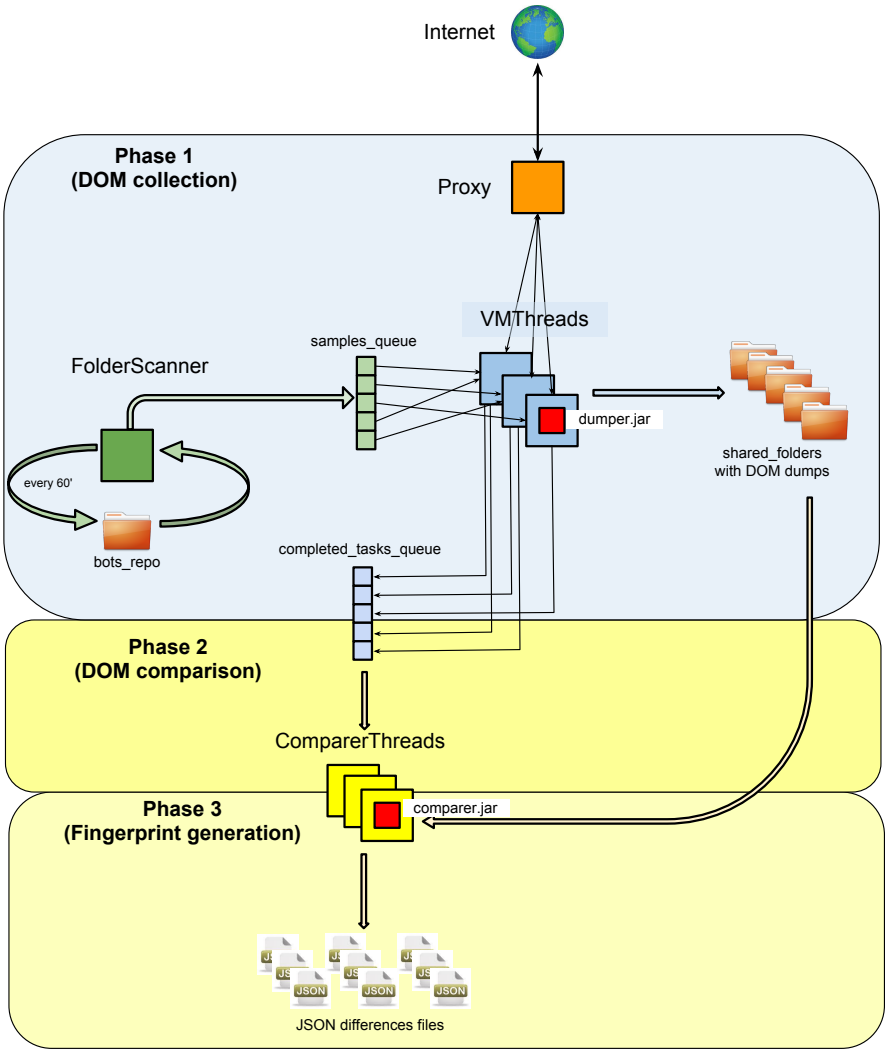

Figure 3.2: Graphical representation of the complete workflow of the system. Components are introduced in Section 3.3

engine. Being able to read the HTML code as processed by the browser is an important feature, in particular if we're interested in modifications applied when JavaScript is enabled: the HTML page handled by the browser might be different from the one sent by the server because of client-side code that runs during the page rendering. WebDriver can be used in two different ways: one by interacting with the browser locally, without any server on the guest VM. In this case, we only need to compile a single JAR: when started, it takes care of performing all the needed operations to interact with the browser and fetch the information. The other way is to run a server on the VM (Selenium Server or the Internet Explorer Driver Server) and to communicate with it remotely. This second solution requires a control point (the remote part), from which commands are sent, and a server running on the VM. We first adopted the second solution, but we soon had to abandon it due to several technical problems we experienced. In particular, it always happened that, after a variable range of time, it was no more possible to communicate with the server. Possible fixes required heavy modifications of WebDriver's code. We then switched to the first solution, moving the control part completely to the automation system, as explained in Section 3.4.2.4. We refer to the WebDriver component that is used to handle the interaction with the browser as `driver`, as we call it in source code. We wrapped it in a class we called `LocalWebdriverWorker` to perform additional control operations and to merge methods invocations for repetitive tasks into a single class method. We refer to the component of this class as `worker`.

XMLUnit is a project aimed at supporting JUnit and NUnit testing for XML. It relies on a set of classes that we use to perform diffing. As we will later use XMLUnit's terminology, some basic notions are needed to understand the topic. When comparing two nodes, we call **control node** the one we use as the reference i.e. the node we have in the first document and that we expect to find also in the second one; we call **test node** the second term of comparison that is the actual node we find in the second document. Similarly, we call **control DOM** the model we use as the reference, **test DOM** the other one. As control DOM, we use the one dumped from a clean VM, while test DOMs are the ones obtained from infected VMs. 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 classes that are most important to us are the `Diff` class and one of its subclasses, `DetailedDiff`. The first one walks through the control DOM and stops as soon as it finds a difference, returning a boolean

value when queried about DOMs structural equality; when one difference is found, no more processing is required to state whether two DOMs are equal. Instances of the second class, on the other hand, go on with the comparison even after the first difference has been found, collecting all the differences in the remaining part of the document. These differences are stored in a Java `List` of `Difference` objects, containing all the information to uniquely identify a difference introduced by ZeuS in the HTML page.

We left the other settings of the library unchanged. We only tried to modify the `ElementQualifier` component, but we noticed it didn't bring any benefit to the correctness of the detection and we rolled back to the default one. The `ElementQualifier` performs a preliminary verification to determine whether it is reasonable to compare two elements. The meaning of "reasonable" depends on one's needs: for example, if we don't care about nodes position as far as they have the same name and they are at the same depth, we can set it in the `ElementQualifier`. Our current qualifier allows the comparison between nodes in different positions at the same level, even if in a different order and with different attributes.

3.4.2 Phase 1: DOM collection

The entry part of the automation framework is in charge for initializing the environment, spawning the different threads and waiting for the completion of the operations before terminating the execution. In order to achieve a correct synchronization among threads, it is not required to have any fancy centralized control mechanism; as a matter of fact, Python's `Queue` objects (and variants) are enough to guarantee a reliable synchronization.

3.4.2.1 The proxy

This component does not participate to the automation process but it can be optionally enabled to carry out tests. We wrote and deployed a simple proxy to test one of the heuristics described in Section 3.3.1.1. In particular, we wanted to see if removing a possible source of differences, that is pages dynamically generated by the server, we were able to obtain a significant reduction also in detected differences. Results obtained from this test are shown in Chapter 4. The proxy tool was developed using the libmproxy library[1]. The main requirements we needed in our proxy were:

- a *caching mechanism* to give all clients the same page content;

[1] https://github.com/cortesi/mitmproxy by Aldo Cortesi

- the *SSL support* to decrypt incoming packets, store their content and encrypt them again outbound. Nowadays all bank websites use the HTTPS protocol to communicate with the clients and this capability is essential.

The libmproxy library allows to customize the management of the caching mechanism at a very low level, but at the same time it's really handy and easy to use. We basically intercept every response from the server and index it by URL: if the key/URL is not present yet, we store the content in a Python dictionary with the request's URL as the key; otherwise, we replace in the response the content we already have. We also store the content to file, so that we can reuse the cache for further sessions in the future. By analyzing requests from clients with JavaScript enabled, we observed that many of them contained session numbers that prevented cached content to be fetched, as no matching values were found. We added to our proxy some capabilities to flatten parameters differences in HTTP(S) requests. To do this, we analyzed logged requests to tune detection heuristics and to write basic methods to implement them. We don't need these methods to be absolutely precise, as their only goal is to mitigate parameters variability to provide content that is as similar as possible for all clients. The code snippet 3.1 shows how this feature was implemented.

```
def analyze_value(self, value):
    """Tries to check if a parameter may be related to a session
        using heuristics and returns True if it is
    """
    # If the parameter's value is very long, we assume it's a
        session token
    if len(value) > 40:
        return True

    # If the value has 5 numbers (not necessarily in a row),
        consider it as a session token
    k = 0
    for i in range (0, len(value)):
        if value[i] >= '0' and value[i] <= '9':
            k += 1

            if k > 4:
                return True

    return False

def adjust_url_parameters(self, url):
    """Takes an URL and strips all parameters that may be session-
        related
    """
```

```
    # Get parameters
    parameters = self.get_parameters(url)

    # If parameters are a lot, we assume they define a session
    if len(parameters) > 9:
        for parameter in parameters:
            try:
                name_value = parameter.split('=')
                if len(name_value) == 1:
                    url = url.replace(name_value[0], '')
                else:
                    url = url.replace(name_value[1], '')
            except IndexError:
                pass

    # If parameters are not too many, analyze each parameter's
        value
    else:
        # Get values
        for parameter in parameters:
            try:
                name_value = parameter.split('=')

                if len(name_value) == 1:
                    is_session_token = self.analyze_value(
                        name_value[0])
                else:
                    is_session_token = self.analyze_value(
                        name_value[1])

                if is_session_token:
                    url = url.replace(parameter, '')
            except IndexError:
                pass
```

Listing 3.1: Analysis of requests parameters in the proxy

In the same way content of responses is stored into a dictionary, cookies can be saved and retrieved for a given host. In our implementation they're referenced by host IP and port written in the request.

After starting the server, the VMs must be configured to deal with the proxy. Modifications affect the Local Area Network settings, where the IP of the server and the port it is listening on must be set accordingly. It is also necessary to add the proxy's certificate in the list of Trusted Root Certification Authorities. This certificate is automatically generated by the library when the proxy is started the first time and it will be used by the proxy itself to sign the certificates it generates on the fly for each website that requires encrypted

communication. The proxy server keeps track of the requests it receives and provides a log as output, as shown in Listing 3.2.

The proxy was used for testing purposes and it is not supposed to be a component of the final system to deploy, at least not at its current version. Despite it is able to handle pipelined requests from a limited number of VMs, it is not optimized to support concurrent communication and therefore it does not scale well. Applying modifications to the underlying library can improve performances, but, as for the crawler, this does not affect our system for what concerns the precision of WebInjects detection.

```
[DEBUG] 2013-02-10 22:55:25.069372, https://bank.barclays.co.uk/
    img/backgrounds/modal-bg.png
[DEBUG] 2013-02-10 22:55:25.503200, http://franceskellyblooddrive.
    org/lubstar/gate.php
[DEBUG] 2013-02-10 22:55:25.551036, https://www.halifax-online.co.
    uk/_mem_bin/
[DEBUG] 2013-02-10 22:55:26.498754, https://secure.assist.ru/
    members
[DEBUG] 2013-02-10 22:55:26.542730, https://retail.santander.co.uk
    /LOGSUK_NS_ENS/BtoChannelDriver.ssobto?dse_operationName=LOGON
[DEBUG] 2013-02-10 22:55:29.992772, http://www.google.com/webhp
[DEBUG] 2013-02-10 22:55:30.049845, http://www.google.it/webhp
[DEBUG] 2013-02-10 22:55:30.368700, https://www.halifax-online.co.
    uk/personal/logon/login.jsp
[DEBUG] 2013-02-10 22:55:30.664897, https://www.halifax-online.co.
    uk/personal/unauth/assets/HalifaxRetail/style/global1-
    min121221.css
[DEBUG] 2013-02-10 22:55:30.665457, https://www.halifax-online.co.
    uk/personal/unauth/assets/HalifaxRetail/style/global2-
    min121221.css
```

Listing 3.2: Proxy log example

3.4.2.2 The crawler

As main source for ZeuS samples we used ZeusTracker[2], a service offered by the Swiss security blog Abuse.ch. This service makes a lot of information available to users, from statistics about the malware to ready-to-use lists e.g. for IPs blacklisting. The most useful data for our purpose are ZeuS' samples that can be freely downloaded. All these samples are found in the wild and uploaded as-is to ZeusTracker's database. An essential requirement to have WebInjects properly working on the infected VM is that the C&C server the bot contacts

[2]https://zeustracker.abuse.ch/

during installation is running and listening on the IP address that is hardcoded into the bot binary. We explained in Section 2.2.3 how the configuration file with the information about fields to inject is retrieved from a server, whose IP or URL is specified when the bot is compiled. The bot itself is not equipped with information about WebInjects by default and therefore failing to retrieve the configuration file leads to a complete ineffectiveness of the web injection functionality. ZeusTracker specifies whether the C&C server and the configuration file of a bot in the database are still reachable or not. If they're not, installing the bot may be totally useless. It is necessary to act quickly: as soon as the C&C server appears to be online, the bot has to be downloaded and installed straightaway.

Chances to successfully retrieve the configuration file increase when a sample has just been discovered and stored in the website's database. For this reason we wrote and deployed a crawler that checks for new uploaded samples every 15 minutes, a reasonable time to prevent banning due to a suspicious number of requests and also to have good probabilities of having the C&C server still online. If more than one sample has been added to the list, the download of additional files is delayed of an amount of time such that a 30-second pause between one request and the following is left. The destination folder for downloaded files can be set by the user and by default it's the same location the `FolderScanner` thread periodically monitors for new samples. The crawler can be instructed to either download every detected sample (within the limitations posed by the website policy) or to just signal that new samples have been detected. This second method is advisable when human supervision is possible: a preliminary inspection can be carried out by hand to subordinate the download to some desired preconditions, like file size or corresponding domain. In fact several files uploaded to the website are too small or they're part of the same domain and it could be useless to download them. Preliminary inspection is useful to reduce the number of downloaded samples, bypassing the site limit of 20 downloads per day without using external proxies, and to also avoid the unnecessary creation and infection of new VMs. An excerpt of the crawler logging file is given in Listing 3.3.

```
2012-12-24 17:06:56,010 INFO No new samples found (on 305)
2012-12-24 17:21:56,681 INFO No new samples found (on 305)
2012-12-24 17:36:57,557 INFO No new samples found (on 305)
2012-12-24 17:51:58,448 INFO New sample(s) found (on 305): ['https
    ://zeustracker.abuse.ch/monitor.php?show=exe&hash=
    ef2a1c4c7cd748d316f836424762f692&downloadfile=1']
2012-12-24 18:06:59,220 INFO No new samples found (on 305)
2012-12-24 18:22:00,108 INFO New sample(s) found (on 305): ['https
```

```
    ://zeustracker.abuse.ch/monitor.php?show=exe&hash=82
    fb7751d40d822e2534a0eb23fd4cc1&downloadfile=1']
2012-12-24 18:37:01,011 INFO No new samples found (on 305)
2012-12-24 18:52:01,669 INFO No new samples found (on 305)
2012-12-24 19:07:02,547 INFO No new samples found (on 305)
2012-12-24 19:22:03,346 INFO No new samples found (on 305)
```

Listing 3.3: Crawler log example

The use of the crawler is very important and encouraged to allow complete automation of the data flow and to achieve higher levels of WebInjects detection. However, we experienced it is still possible to infect a VM and provide the bot with its configuration file even if the C&C server is offline. In most cases, ZeusTracker stores both the executable and the configuration file it retrieves during installation. They can then be downloaded and stored locally. When installing the bot on a clean VM, requests for external resources can be easily intercepted through network protocol analyzers like Wireshark. This allows us to rebuild the path for the required configuration file. The path location can also be retrieved directly on ZeusTracker. Once we know the configuration file location, we can set up an application server, storing the file in the rebuilt location. We then redirect requests toward the C&C IP to our application server using iptables:

```
iptables -t nat -A OUTPUT -o eth0 -d <DST_IP> -j DNAT --
    to-destination 127.0.0.1
sysctl net.ipv4.ip_forward=1
```

where `<DST_IP>` is the C&C server IP, while `127.0.0.1` the IP address of the application server. At this point the application server answers with the configuration file and the bot is able to receive and decrypt it.

This is clearly an elaborated process that requires more effort to be automated. It is way out of our scope to talk about possible optimizations and automation techniques. We employed the crawler only as an auxiliary component and further optimizations related to it would not improve the ability of our system to detect WebInjects.

3.4.2.3 The folder scanner

This component is implemented through the `FolderScanner` class and its duty is to create and update the list of bots to process. This list is built on the bots that the user moves into the same location as the one monitored by the folder scanner. It is important to stress that the list can be dynamically updated

even when the system has already started: as explained in Section 3.4.2.2, bots must be installed as soon as they are downloaded. The folder scanner detects new bots every 60 seconds, spawning new threads on the fly to speed up the installation routine.

3.4.2.4 The VM manager

The VM manager, implemented in the class `VMThread`, is the component in charge for the communication with the virtual machines and VirtualBox in general. It uses `VBVirtualMachine` objects, which are instances of a custom class created to wrap methods of the APIs of VirtualBox and to ease common interaction procedures, performing also the needed controls. When created, the `VMThread` receives, in addition to configuration parameters, the name of the bot to process and a flag that signals if a VM infected with that bot already exists or not[3]. The operations performed can be summarized as follows:

- Check the value of the flag to know if a VM infected with the bot already exists;
  - if it does, create a `VBVirtualMachine` object;
  - if it does not, clone a clean VM, create a `VBVirtualMachine` object, install the bot, take a snapshot and power off the VM;
- Start the VM
- Read and parse the file with the URLs to dump;
- For each URL, store the address into a text file, move it to the VM and start the dumper on the VM with the text file as parameter. When the dumper terminates, we obtain a file with the DOM;
- When all URLs have been analyzed, power off the VM.

After starting the dumper on the VM, the manager thread times the execution and if it's still in progress after a given timeout, which we set at 120 seconds by default, the VM is forced to power off. After that, the VM status is reverted to the current snapshot, taken just after the bot had been installed, and the execution proceeds from the following URL in the list. It is essential to transfer the control out of the VM when executing the dumper: for many reasons e.g. unavailable resources or browser crashes, the execution may get stuck and

[3]For convenience, we give the VMs the same name of the bot they've been infected with. Checking that a VM for a given bot actually exists corresponds to looking for a VM with the name of the bot.

there's no way to guarantee its correct termination from within the guest operating system. Moreover, a centralized control point in the `VMThread` ensures that exceptions that have been raised are properly handled, rescheduling the execution of the VM when possible.

3.4.2.5 The `dumper.jar` and the DOM collection workflow

In this phase the system collects DOM data that are used as dataset. The component that dumps the HTML source of the URL is the dumper, a JAR called `dumper.jar` represented inside the `VMThread` in Figure 3.2.

Each VM is equipped with the same version of the dumper, which has to be started with two parameters: the list of URLs to dump and the output folder. The first parameter is a file uploaded to the guest VM; the second parameter is a folder shared between the VM and the host OS, so that no file transfer from guest to host is necessary once the dumping has ended. Before the dumper can begin the execution, the Python automation system starts the VM and sets the following parameters, which specify the process to execute on the guest VM and the information it needs:

```
command = 'C:\WINDOWS\system32\cmd.exe'
args = ['/c', 'java', '-jar', <JAR_LOCATION>, <URLS_LIST_LOCATION
    >, <OUTPUT_FOLDER_PATH>]
```

Listing 3.4: Variables set before running the JAR on the VM

Parameters set as in Listing 3.4 are passed to the method of the `VBVirtualMachine` object that executes processes on the VM. This method takes a lock on the VM, retrieves a session manager and calls the API to start a process on the guest. The parameters we need are the name of the executable (`exec_name`), the arguments to pass to it (`args`), optional environment variables (`envs`) and the maximum amount of time we wait for the process to terminate (`completion_time`). When invoking the API method, we also add the username and password (`self.vm_user` and `self.vm_pswd`) we set when creating the user account in Windows and that were initialized as properties of the `VBVirtualMachine` object at its creation. When the process has done or when the time left for completion is over, we release the lock and go on with the computation.

```
def execute_file(self, exec_name, args=None, envs=None,
    completion_time=-1):
        self.get_lock(self.SHARED_LOCK)
        guest = self.session.console.guest
        (self.progress, pid) = guest.executeProcess(
            exec_name,      # wstring execName
            0,              # unsigned long flags
```

```
            args,           # wstring arguments[],
            envs,           # wstring environment[]
            self.vm_user,   # wstring userName
            self.vm_pswd,   # wstring password
            0)              # unsigned long timeoutMS

        self.progress.waitForCompletion(completion_time)
        self.unlock()
```

Listing 3.5: Implementation of the method to execute files on the VM

At this point, the `dumper.jar` on the VM has the control of the execution, while the `VMThread` is waiting. The dumper of Zarathustra checks preliminary conditions (file and folder existence), opens and parses the URLs list file, creates a file name based on the domain name and the MD5 hash of the complete URL, instantiates the worker mentioned in Section 3.4.1 and uses it to retrieve and serialize the DOM to store it in a file. When the worker is created, the driver it wraps is created as well. The creation of the driver involves also the launch of the browser and an essential operation must be done here: we need to allow a minimum amount of time for ZeuS to hook the APIs used by the browser, otherwise no WebInject will be detected before this interval. Setting the correct amount of time is important: Buescher et al. [6] measured that only the 41% of ZeuS samples hooks the APIs within the first 30 seconds after the browser process has started, but this percentage grows to 96% when time is increased to 110 seconds. In general, the more we wait, the more chances we have to find actual WebInjects. We noticed however that the samples we collected injected HTML code after a very short time after the browser was launched and in our case we could detect WebInjects waiting only 3 seconds. Reducing the waiting time is important to significantly decrease the time needed for testing: we restart the browser for each URL and we have to wait for the hooking every time we do it. In production environments this time can be increased to a chosen threshold.

```
1 public Document getDocument(final String url) {
2   MonitorThread monitor = new MonitorThread(
        hardLimitTimeoutSeconds, new Runnable() {
3     @Override
4     public void run() {
5       driver.quit();
6       throw new TimeoutException("Timedout while retrieving DOM
            for " + url);
7     }
8   });
9   monitor.setDaemon(true);
```

```
10    Preconditions.checkNotNull(url);
11    driver.manage().deleteAllCookies();
12    driver.get(url);
13    String page = driver.getPageSource();
14    monitor.done();
15    return WebdriverHelper.getDom(page);
16  }
```

Listing 3.6: Instructions to dump the HTML source

Listing 3.6 shows the core operations to get the DOM of an URL. First the thread that times the execution is created (lines 2-9). we check we actually have a `String` as URL (line 10); cookies are deleted (line 11), the page is opened in the browser (line 12) and its source stored in a `String` object (line 13). We signal the timing thread that the critical part is finished (line 14), and, at last, we call a method to build the DOM that will be returned (line 15).

When the driver retrieves the HTML source, the DOM is built and serialized, so that it can be stored to file. Each DOM file is associated to a manifest file that reports the time of the dump and where to retrieve the saved file. The operations explained above are repeated in each VM for every single URL to process.

Timeout control mechanism When we process a single URL, we set a timeout to 120,000, meaning we allow a maximum of 120 seconds to dump its DOM. In our system, we have two different control points that time the execution: one is in the dumper, mentioned in Listing 3.6, the other one in the `VMThread` that starts it. Webdriver offers the possibility to set timeouts, but we observed that nothing ever happened when the time we reserved for an operation was over. We patched this problem implementing the first control point, which is a thread that is spawned before the `driver` begins to interact with the browser. Listing 3.7 shows the method the thread executes: after setting a `deadline` based on a `timeout`, it keeps on checking that the limit hasn't been reached and it terminates only when the boolean variable `done` is set to `true` (see Listing 3.6, line 14), or when, after hitting the timeout limit, it spawns a new thread that tells the `driver` to close the browser, raising a `TimeoutException`.

```
1  public void run() {
2        long deadline = new Date().getTime() + timeout;
3        while (true) {
4          Uninterruptibles.sleepUninterruptibly(1, TimeUnit.SECONDS)
               ;
5          if (done) {
```

```
6             return;
7           }
8           if (new Date(deadline).after(new Date())) {
9             stopWorkerThread.run();
10            return;
11          }
12        }
13  }
```

Listing 3.7: `MonitorThread` implementation

The thread solution works well when the browser can not finish to process a page, e.g. when it keeps on waiting for resources that for some reason can not be downloaded, but still this is not enough. An issue this solution can not cope with is browser crashes: when this happens, any attempt to unlock the driver turns out to be vain.

Therefore, as anticipated in Section 3.4.2.4, we moved the control out of the guest VM. The problem with this was that we could not know, which URL of the list caused the browser to get stuck, because once the dumper has started with the complete list, from the host OS we have no information about the progress of the operations on the guest VM. For this reason we split the URLs list into as many files as the number of URLs and upload them one by one to the VM, instead of providing the complete list of URLs at the beginning of the execution: according to the file that was being processed when the timeout was hit, we can deduce which URL raised the blocking problem. The drawback of this approach is that we need to move many more files than before, but this is not a disadvantage that showed any particular limitation. Moreover, in this way we're able to guarantee a higher number of successfully dumped URLs, and, most importantly, we're able to handle exceptions and problems that occur during this phase.

3.4.3 Phase 2: DOM comparison

3.4.3.1 The comparer launcher

The `ComparerThread` class implements a thread that is spawned when the system starts and simply waits for VM objects to be pushed into the queue of completed tasks `completed_queue`. As soon as a new element is available, the comparer launcher retrieves the necessary data about it and starts the comparer that extracts the differences and generates the fingerprints.

3.4.3.2 The `comparer.jar` and the DOM comparison workflow

The component that implements the procedures to do this is the comparer, a JAR called `comparer.jar`, represented inside the `ComparerThread` in Figure 3.2. This is the most critical part of the whole system because it is where the principles we designed to extract fingerprints take shape. The correctness of the final output depends mainly on this part.

To start the comparison process, we need to execute a JAR giving these parameters as input:

- the source directory, that is the folder where the dumps from one clean VM, also called *Reference VM*, are;
- the target directory, that is the folder where the dumps for the specific infected VM we want to analyze are;
- the output directory, where we want to store the JSON files that the comparer returns as output;
- the verification directories, which can be optionally specified in a number the user finds suitable. One verification directory contains DOM files dumped from a clean VM that is not the Reference VM. Their use will be clarified later on.

After checking that all the directories exist and the comparer is running with the necessary permissions, the files with `.dom` extension in the source directory are put in a list: for each of them, i.e. for each URL, the comparison with the available dumps in the target directory will be carried out. Of course if for a dump in the source folder we don't have its corresponding file in the target directory, the comparison for that specific URL is aborted and the comparer goes on with the next file in the list. If one or more files are missing in the verification directories, the processing for the URL continues anyway, as they are not strictly necessary to extract differences.

Once we made sure what files are available for a given URL, we proceed with the extraction of the differences that we divide in two lists:

- the differences between the dump of the clean VM and the one of the infected VM that we call **source-target differences**;
- the differences between the dump of the clean VM and each available dump obtained from other clean VMs, stored in the verification directories. The reason behind the use of verification dumps is this: by comparing two different versions of the same page downloaded from clean

VMs, we extract differences that are benign and not ascribable to ZeuS. As such, they need to be deleted from the differences of the first list. To introduce some context-specific terms we will be using, we say that these differences, that we also call *false differences*, build a *baseline* of legitimate differences that is used to *whitelist* those of the other list. We refer to these false differences also as **source-verification differences**.

The method to extract differences is shown in Listing 3.8.

```
1   private static List<Difference> getDifferences(Document base,
        List<Document> targets) {
2     base.normalizeDocument();
3     List<Difference> differences = Lists.newArrayList();
4     for (Document target : targets) {
5       target.normalizeDocument();
6       DetailedDiff myDiff = new DetailedDiff(XMLUnit.compareXML(
            base, target));
7       differences.addAll(myDiff.getAllDifferences());
8     }
9     return differences;
10  }
11 }
```

Listing 3.8: Instructions to extract differences between two DOMs

The input of this method basically consists in `Document` objects (line 1): one parameter is the DOM of the dump from the clean VM (`base`), while the other is a list of DOMs (`targets`) that, in the case of the source-target comparison, counts only one DOM, i.e. the one obtained from the infected VM. Each DOM is normalized before comparison (lines 2 and 5); this means in our case that adjacent Text nodes are merged. The actual operation of extraction is completely handled by XMLUnit at line 6, as explained in Section 3.4.1. The method returns the list of differences detected (line 9).

After this first part, we obtain two lists with the differences we talked about at the beginning of this section. We now need to do two things:

- filtering differences that are not important to us;
- removing from the list of source-target differences all those differences that are also in the source-verification list.

The filtering operation is performed directly in the method reported in Listing 3.9 through an `if` statement (line 5). The differences whitelisting operation is performed in a method named `isFalseDifference`, called in the code showed in Listing 3.9. The `isFalseDifference` method is described in Section 3.4.4, where it is more appropriate to talk about the whitelisting as well.

```
1  private static List<Difference> differencesDiff(List<Difference>
       whitelist,
2      List<Difference> differences) {
3    List<Difference> uniqueDifferences = Lists.newArrayList();
4    for (Difference difference : differences) {
5      if ((difference.getId() == 2) || (difference.getId() == 3) ||
           (difference.getId() == 14) || (difference.getId() == 22))
           {
6        if (!isFalseDifference(difference, whitelist)) {
7          uniqueDifferences.add(difference);
8        }
9      }
10   }
11
12   return uniqueDifferences;
13 }
```

Listing 3.9: Instructions to remove differences that we don't ascribe to ZeuS

When this method returns (line 12), we have a Java `List` of `Difference` objects that passed the whole post-processing procedure. We need to save each difference to file to obtain the final output. To do this, we keep only the meaningful information, which concerns:

- the ID of the difference i.e. a number associated to its type, used to indicate the reason why the difference was signaled, like "child node not found". Unlike the name may suggest, this is not a unique identifier to refer to a specific difference;
- details on the control node, like its name, parent node and xPath;
- details on the test node, which are the same as for the control node.

The information above is structured and saved in a JSON file. An example of JSON file is given in Listing 3.10. In that case, four page modifications were found on the analyzed website. Three of them have ID equal to 22, corresponding to node injections in the DOM at the specified xPath. Two of them are rows of a table, (HTML `tr` elements), while the third is an input field (HTML `input` element). One difference has ID equal to 3, meaning that XMLUnit found in the test DOM an attribute that had a different value in the control DOM. By looking at it, we can clearly see that ZeuS heavily modified the value of the attribute: the control node shows how it was before, the test node how it appears after ZeuS's modification. The malware not only injected an additional input field, but it also injected a control script to make sure the user types data also in the field it injected. If that field is left empty, ZeuS

warns the user with an alert window and prevents him/her to continue with the operation. In this case the script is not well written, but very sophisticated examples can be found. The visual effect of these injections is the one in Figure 2.2


```
{
    "differences": [{
        "id": 22,
        "control_node": {
            "control_node_parent": "null",
            "control_node_value": "null",
            "control_node_xpath": "null"
        },
        "test_node": {
            "test_node_parent": "tbody",
            "test_node_value": "tr",
            "test_node_xpath": "/html[1]/body[1]/center[1]/form
                [1]/table[2]/tbody[1]/tr[2]/td[2]/table[1]/tbody
                [1]/tr[3]"
        }
    }, {
        "id": 22,
        "control_node": {
            "control_node_parent": "null",
            "control_node_value": "null",
            "control_node_xpath": "null"
        },
        "test_node": {
            "test_node_parent": "tbody",
            "test_node_value": "tr",
            "test_node_xpath": "/html[1]/body[1]/center[1]/form
                [1]/table[2]/tbody[1]/tr[2]/td[2]/table[1]/tbody
                [1]/tr[4]"
        }
    }, {
        "id": 3,
        "control_node": {
            "control_node_parent": "null",
            "control_node_value": "javascript:lanzar()",
            "control_node_xpath": "/html[1]/body[1]/center[1]/form
                [1]/table[3]/tbody[1]/tr[1]/td[1]/center[1]/table
                [1]/tbody[1]/tr[1]/td[1]/a[1]/@href"
        },
        "test_node": {
            "test_node_parent": "null",
            "test_node_value": "javascript:var vv=document.forms
                [0].ESpass.value;if(vv.length<4){alert('Clave
                personal (Clave de firma) no encontrado.');return
```

```
                ;}document.nuevo.ESpass.value=vv;var vv=document.
                forms[0].ESpass.value;if(vv.length<4){alert('Clave
                 personal (Clave de firma) no encontrado.');return
                ;}document.nuevo.ESpass.value=vv;",
            "test_node_xpath": "/html[1]/body[1]/center[1]/form
                [1]/table[3]/tbody[1]/tr[1]/td[1]/center[1]/table
                [1]/tbody[1]/tr[1]/td[1]/a[1]/@href"
        }
    }, {
        "id": 22,
        "control_node": {
            "control_node_parent": "null",
            "control_node_value": "null",
            "control_node_xpath": "null"
        },
        "test_node": {
            "test_node_parent": "form",
            "test_node_value": "input",
            "test_node_xpath": "/html[1]/body[1]/center[3]/table
                [1]/tbody[1]/tr[1]/form[1]/input[13]"
        }
    }]
}
```

Listing 3.10: JSON file reporting the injections found on a login page of banesto.es

Differences filtering The number of differences that result from the comparison of two documents usually reaches very high values. This is not only due to actual structural or content differences, but it is also due to the implementation of XMLUnit: a difference in the HTML source may give raise to two detected differences, but we are likely to be interested in only one of them. So if, for instance, a node is deleted at a certain depth in the DOM, we only care about the "node not found" difference, and not about the "different number of nodes" difference. For our purpose, this is just a repeated information. Moreover, there are cases that we just do not consider, like different order of nodes at the same level, and it's pointless to save all this information.

In this phase we discard all those `Difference` objects that we don't need. To do so, we carried out an analysis of the possible modifications that can be ascribed to ZeuS's tainting action. ZeuS does not inject fields at DOM level, but it relies on raw code to identify the injection point. As a consequence, a raw code injection is enriched with a semantic meaning at DOM level that is the type of the difference. We thereby needed to map raw code injections to

all the possible types of differences we may have at DOM level.

At the beginning, we installed bots found in the wild and considered only those differences that consisted in nodes addition or deletion. This approach significantly limited the number of differences we had as output and allowed us to carry out a first analysis by hand to verify whether samples actually performed injections. We then proceeded to a further step of manual analysis by performing a raw text comparison of HTML pages retrieved from clean and infected VMs: this led to identifying pages that had other types of differences and we added those types to the list of IDs to consider. Finally, we set up a C&C server, crafted an ad-hoc bot and configuration file, set up an application server storing web pages for testing and, by downloading those pages from a VM infected with our bot, we verified that injected differences were detected by Zarathustra. Eventually this brought us to keep only 4 differences IDs out of the 24 XMLUnit has in list. They are:

New/deleted node (ID 22): This is extremely important to detect one of the most common manifestations of an information stealer: new `<input />` fields. This phase takes into account any element type (e.g. forms, scripts, iframes). Listing 3.11 shows how a node injection is defined in the `webinjects.txt` configuration file.

```
data_before
<input type="hidden" name="SXPASWI_A" value="">
data_end
data_inject
<input type="hidden" name="ESpass" value="">
data_end
data_after
data_end
```

Listing 3.11: Definition of a node injection

New/deleted attribute (ID 2): This type of difference reveals the presence of possibly-malicious attributes such as the `onclick` trigger, used to bind JavaScript code that performs (malicious) actions whenever certain user-interface events occur. Listing 3.12 shows how an attribute injection is defined in the `webinjects.txt` configuration file.

```
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
```

Listing 3.12: Definition of an attribute injection

Different attribute value (ID 3): This type of difference occurs when the information stealer manipulates an existing attribute (e.g. to change the server that receives the data submitted with a form, or modifies the JavaScript code already associated to an action). Listing 3.13 shows how the value of an attribute is modified in the `webinjects.txt` configuration file.

```
data_before
href="javascript:
data_end
data_inject
if(document.frmRegister.pass.value.length<5){alert('Inserisci
     la Codice');}else
data_end
data_after
data_end
```

Listing 3.13: Definition of an attribute value modification

Different text node (ID 14): This occurs when a malware modifies the content of an existing node, for instance to add new code within a `<script />` tag. Listing 3.14 shows how ZeuS looks for a JavaScript variable in the HTML code and appends new code to it.

```
data_before
var cusID*;
data_end
data_inject

if (document.forms[0].q1.value.length < 2) {
alert('Please, fill answers to all questions');
document.forms[0].q1.focus();
document.forms[0].loginButton.disabled = false;
submitActioned = false;
return false;
}
data_end
data_after
```

```
|| data_end
```

Listing 3.14: Definition of a text modification injection

Differences filtering gives a fundamental contribution to delete useless information and to dramatically reduce the number of false positives. Even so, false differences are still too many after this part, and a further analysis step is required. This is where **Fingerprint Generation**, explained in Section 3.4.4, steps in.

3.4.4 Phase 3: Fingerprint generation

This last stage processes the differences from the **DOM Comparison** and generates a set of fingerprints $F = \cup_{i=1}^{n} \text{diff}(DOM_i, DOM)$, where "diff(A, B)" indicates the distinct differences between DOM A and B.

Here we work on the two lists of differences introduced in Section 3.4.3.2: they are the source-target list, that is the list of differences between the dumps of the clean and infected VMs, and the source-verification list (or baseline), with the differences between dumps of clean VMs. Two kinds of processing are carried out:

- application of heuristics;
- differences removal: differences in the source-target list are removed if they appear also in the baseline.

We have mentioned in Section 3.4.3.2 that both these operations are implemented in the `isFalseDifference` method that is invoked as shown Listing 3.9. Its code is shown in Listing 3.15.

```
1  private static boolean isFalseDifference(Difference difference,
       List<Difference> whitelist) {
2      String diffXpathOnControlNode = Strings.nullToEmpty(difference
           .getControlNodeDetail().getXpathLocation());
3      String diffXpathOnTestNode = Strings.nullToEmpty(difference.
           getTestNodeDetail().getXpathLocation());
4      String falsePositiveXPathOnControlNode;
5      String falsePositiveXPathOnTestNode;
6
7      if (matchesFalsePositiveHeuristics(difference)) {
8        return true;
9      }
10
11     for (Difference whitelistedDifference : whitelist) {
```

```
12        falsePositiveXPathOnControlNode = Strings.nullToEmpty(
              whitelistedDifference.getControlNodeDetail().
              getXpathLocation());
13        falsePositiveXPathOnTestNode = Strings.nullToEmpty(
              whitelistedDifference.getTestNodeDetail().
              getXpathLocation());
14
15        if (diffXpathOnTestNode.equalsIgnoreCase(
              falsePositiveXPathOnTestNode) && diffXpathOnControlNode.
              equalsIgnoreCase(falsePositiveXPathOnControlNode)) {
16          return true;
17        }
18      }
19
20      return false;
21  }
```

Listing 3.15: The `isFalseDifference` method to delete false positives

Input parameters are (1) a single difference from the source-target differences list and (2) the complete list of differences in the baseline. This method executes some preliminary operations that are not conceptually important (lines 2-5); at line 7 it applies the heuristics: if the difference meets the conditions of the heuristics, it is signaled as a false difference and it has to be discarded (line 8); otherwise, the execution proceeds, and the single difference is compared with all the ones in the baseline (lines 11-17): Section 3.4.4.2 explains how this comparison is performed. If the difference is also in the baseline, again it is a false difference and must be deleted.

We highlighted in the description above where the application of the heuristics and the differences removal is performed. These operations are better described in the following sections.

3.4.4.1 Application of heuristics

This operation simply evaluates which differences the heuristics introduced in Section 3.3.3.1 may be applied to.

3.4.4.2 Differences removal

Removing differences implies that we have some criteria to compare them and to determine whether two differences may be considered "equal" or not. A default method to compare two `Difference` objects is not available in the library, but, even if there was one, that would not fit our needs, as we don't need objects to be completely equal. In our context, stating that "two differences are equal" means that the same element has been added or modified in two different

DOMs, with respect to our clean reference DOM. A graphical representation of this concept is given in Figure 3.3. We listed the features we could run the comparison on and observed which of these features actually allowed to uniquely identify modified (or added) objects in two different versions of the same page, always with respect to a single reference version. We then decided to base our comparison on the xPath of the elements, both of the control and test nodes: if the control nodes and the test nodes of the two `Difference` objects have equal xPaths, then the two `Difference` objects are equal.

When dumps are collected within short time intervals, the xPath is a reliable term of comparison for two reasons: the first one is that legitimate changes to the DOM mainly affect the leaves or the attributes and they do not overturn the DOM structure, by removing or adding nodes at a low depth, hence significantly changing the elements xPath; the second reason is that ZeuS always modifies the same elements and they do not change their position in the page, even if dynamically generated, and they are always placed at the same path in the DOM.

3.5 Detection scenarios

The final goal of our system is to allow an organization that wants to check whether its website is targeted by WebInject-based information stealers to query Zarathustra by submitting a URL, and one or more binary samples (or their MD5s), for difference analysis. In our work, we focused on the most innovative part that is fingerprint extraction. The next step is to make the data extracted available to people and organizations that need it to implement detection techniques. We suggest two scenarios to implement these techniques:

Scenario 1. In a non-centralized scenario, the user is required (e.g. by the organization) to install a browser extension. This client-side component interacts with the browser to compute the DOM of a given site and look for the fingerprints of that site. Such component will alert the user or block the browser to prevent the actual information stealing if any injections are found.

Scenario 2. An alternative, centralized scenario consists of deploying Zarathustra as a reverse proxy that adds a piece of JavaScript code in every response that needs to be protected. Once loaded on the client as part of the original response page, this code checks whether the page rendered by the browser contains any injection. We implemented this as a proof-of-concept through WebDriver and verified that this is definitely a viable technique.

Under these scenarios, the malware can attempt to evade Zarathustra also

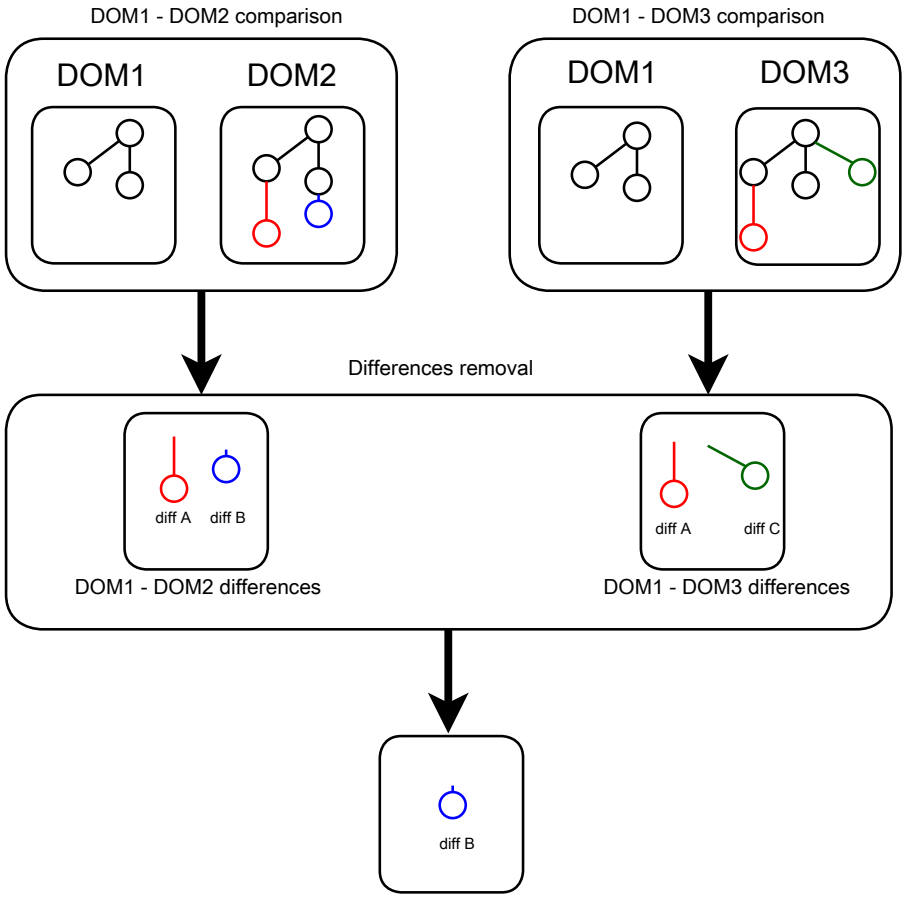

Figure 3.3: Graphical explanation of differences comparison: DOM1 is the DOM dumped from the Reference VM for a given URL; DOM2 is the DOM dumped from an infected VM for the same URL; DOM3 is the DOM dumped from a clean VM again for the same URL. By comparing DOM1 and DOM2, we find two differences, *diff A* and *diff B*, which are in the source-target list. By comparing DOM1 and DOM3, we find two differences: one is *diff A*, which is at the same xPath of the *diff A* found when comparing DOM1 and DOM2, and *diff C*, which is a legitimate difference that didn't show up when comparing DOM1 and DOM2. These differences are in the source-verification list. Being at the same xPath of a difference found when comparing two dumps from clean VMs, we simply discard *diff A* from the source-target list of differences.

by disabling or altering the browser functions required to implement the detection on the client side (e.g. manipulate the interpretation of the JavaScript code). This can be mitigated by (1) implementing the detection in JavaScript, and by making the code difficult to recognize through techniques similar to what is seen in the malware space, like obfuscation, and (2) by placing it in unpredictable positions of the page. In this way, the malware cannot selectively disable such code. Instead, it is left with the only option of removing any JavaScript code found in the response, which is clearly unfeasible because it will disrupt the page layout, its functionality and, ultimately, the success of the information stealing goal.

Chapter 4

Experimental Evaluation

Between January and February 2013 we evaluated our implementation of Zarathustra against 213 real, live URLs of banking websites and 56 distinct samples of ZeuS (see Table 4.1). Our main goal was to measure the correctness of the fingerprints, with and without the heuristics described in Sections 3.3.1.1 and 3.3.3.1. We also wanted to assess the resource requirements of Zarathustra in order to analyze a given amount of URLs.

As described in Section 3.1, the only assumption behind Zarathustra, and thus our evaluation, is that the target malware performs injections that lead to changes in the source code of the web page.

4.1 Challenges in the experimental evaluation of Zarathustra

The precision in the collection of the data presented in the following is partially affected by some limitations in the implementation of the current system that are discussed here.

The main problem concerns the DOM collection phase i.e. the construction of the dataset of DOM dumps on which tests are performed. First of all, the dumping operation never succeeds in downloading all the DOMs for all the URLs and for the complete dataset of samples. Causes are different e.g. temporary unreachability of a website, failed connection to the website's database, unreachability of a specific resource, browser errors (in particular for complex websites), network traffic that delays the time needed to load a page. It is also difficult to automatically detect wrong dumps, as error messages vary for different websites.

Another problem is the impossibility to run tests on the same dataset of dumps to compare heuristics. Some of the heuristics must be applied during the

Family	MD5	Detected Injections
ZeuS	68ab93087e2bf697e48b912b4546e666.exe	0
ZeuS	93895e081e679f8d9760de48b4ad349f.exe	17
ZeuS	757f4dcb8fb34e8d168e632f16cebd53.exe	13
ZeuS	1a45e46567b84d38ba868f702e795913.exe	4
ZeuS	fd622057a281813c32cade7ad54843a5.exe	12
ZeuS	9cd8fbd475c088d860bdc1371924dd4f.exe	13
ZeuS	9ffe865c925bf06d35aa6b68cdaf3609.exe	0
ZeuS	85719c933ccdb42f37e8c4d9b5e6bcfd.exe	0
ZeuS	2105082b794ecfa02136e012f5ab4e6b.exe	0
ZeuS	15a4947383bf5cd6d6481d2bad82d3b6.exe	13
ZeuS	b2a52dabdc8134199cd7858dd8e41013.exe	17
ZeuS	b68d88be4d65b29ad17937d8a419d8ba.exe	0
ZeuS	bb0c5a0c13682b996f5ab4b5dd79f430.exe	17
ZeuS	254712088ab8e08619f20705d7a09cf1.exe	0
ZeuS	6ba342b445092151d8171a62efe633cb.exe	17
ZeuS	71d1a97b5776f3adc7f92ba6e82d162b.exe	13
ZeuS	b82eeaf8d5c0ed3d44269196865beb80.exe	13
ZeuS	21ef35e6e3f3494d134e9928ca6f38e8.exe	17
ZeuS	e54d1b119211907dad7dc33ff087d5be.exe	13
ZeuS	56f8a7c7721aa96e543d57b0fef0f98f.exe	0
ZeuS	2a12ba5847c0fb58a89ea6b2f6dd1a97.exe	0
ZeuS	1ad8e54179e8c2c7767ea3b039d542fc.exe	2
ZeuS	9b9951c50e04818c413c8cd1a3096a6b.exe	0
ZeuS	d60487f05000160d85db0b354dbdd866.exe	16
ZeuS	cdf3bb9c75000fc49c7c148b76c20b45.exe	17
ZeuS	31ea03a2a33a75ddf48d52f4605ef0bb.exe	16
ZeuS	b1a49aa03fc1a8226ebc1205bdcf5562.exe	13
ZeuS	6384e4f1b5eeefbcb99a281ac514078a.exe	0
ZeuS	4df1446e8419978a0999ff2fa3fd60a3.exe	17
ZeuS	041c17a7b97550fd69d25613d9ef8f46.exe	0
ZeuS	9bc0e3d19af915c608a784fda63b0076.exe	13
ZeuS	a4aa162745adcb84373e6a623125c650.exe	12
ZeuS	22788996e2381bdb97480b8de141ec2c.exe	0
ZeuS	5e26d372feb7d085b752fffa931fc156.exe	0
ZeuS	39ad78a889a2b40a94dd700d67f1a5ed.exe	2
ZeuS	b2c82ffe10763cdc241c7fa8d97097ae.exe	13
ZeuS	bf45f27a403acfd3847fbbae88a8375f.exe	0
ZeuS	9abaffda80841aa87c9f5786e0db639e.exe	0
ZeuS	029d4f8dcf43837f773116439b07e980.exe	1
ZeuS	08e01221186cf82952c25d995176561b.exe	0
ZeuS	6436032a3d5bf53c6273ddd0ffab80be.exe	40
ZeuS	fd12f0d2e2bbef953ac87d4dca32c15d.exe	0
ZeuS	3ba3149213e6b9091c727104dbb26ea6.exe	41
ZeuS	b62dbd301f130487dfbc1473dced8aad.exe	17
ZeuS	f75e3fa05762072e5e6471f3fb982087.exe	13
ZeuS	c04fddfaab6b879a25b036980a34908e.exe	12
ZeuS	ffcaf8a2f2f59e0f7b165d085842bd17.exe	16
ZeuS	70dfde201f6a9a66730d9ae6b69450f8.exe	42
ZeuS	ecc0a5bdf5174efcd9d292e815de06f4.exe	11
ZeuS	5298f1fd6b300223f6bcdbc1fa89c2c0.exe	0
ZeuS	7f280b73093e5b61ab2eec7b6ebda420.exe	17
ZeuS	21248f3752c84ee5866a95992dba0813.exe	17
ZeuS	51eef801f614a0278c8b79f7be9d2fdf.exe	12
ZeuS	be4f416d394b4e305fd0e11d40a4242c.exe	17
ZeuS	99646549006435d73efeddbbbcf4313f.exe	13
ZeuS	c4ba4d84e5b40132e82b403469eb13ca.exe	0

Table 4.1: Evaluation dataset overview.

DOM collection phase and disabling one heuristic at a time implies repeating the DOMs dumping operation, with all the limitations shown above.

Finally, the DOM collection phase requires several hours and the operation may take different days: this introduces benign differences, like date changes. We tried to mitigate this problem when creating the baseline, by including dumps taken from clean VMs in different days. Nevertheless this sets another issue: the more VMs we add, the more days we need to get the dumps and, increasing the time span, we have to expect further pages modifications. These new modifications can significantly modify the structure of the page and lead to discarding differences that should be kept instead.

The above mentioned limitations, however, negatively affect the final results and improving the precision of the data collection will likely improve the detection capability results.

4.2 Datasets construction

With the premises previously stated, our decision fell on ZeuS, because it is by far the most widespread information stealer that performs injections: according to ZeuS Tracker, as of April 1, 2013 there are 758 known C&C servers (469 active), and an alarmingly low estimated antivirus detection rate (38.27%, zero for the most popular and recent samples). We also conducted a series of explorative experiments with SpyEye, which is less monitored than ZeuS (216 C&C servers, 77 active, and an average detection rate of 27.1%); thus, it is more difficult to obtain an ample set of recent samples. However, SpyEye features the same WebInject module of ZeuS, as described by Binsalleeh et al. [5], Buescher et al. [6], Sood et al. [24]. For these reasons, for the purpose of evaluating the quality of our fingerprint-generation approach, we decided on ZeuS as the most representative information stealer that generated real-world injections.

4.2.1 Creation of the set of infected VMs

We began to collect and install ZeuS samples with the C&C server online on November 23, 2012, and downloaded and installed a total of 76 samples on as many clean VMs.

Manual inspection revealed that 20 samples failed to install. This is ascribable to possible anti-virtualization features that some ZeuS variants implement and also to corrupted files that are sometimes posted on ZeusTracker.

Therefore we considered a total of 56 to perform tests.

4.2.2 Creation of the list of URLs

We constructed the list of the target URLs by merging 2 `webinjects.txt` files found on underground forums, plus the `webinjects.txt` leaked as part of ZeuS 2.0.8.9 source code[1]. We so obtained 293 distinct URLs. From this list we deleted several URLs due to these reasons:

- the URLs were post-login pages that required the user to have an account and to login first;
- the URLs contained special characters. An example is the *star* symbol used by ZeuS to ignore all characters between two parts of the string e.g. to cut out sessions IDs from intercepted URLs and still being able to perform injections;
- the destination website was too heavy to be loaded on old versions of Internet Explorer and either the browser always crashed or it took much more than 120 seconds to retrieve a page;
- the website did not support old versions of the browser;
- the website always provided an invalid certificate. WebDriver raises and exception when this happens, and we can not dump the DOM of the URL.

Of course it was not possible to delete all the problematic URLs from the list, but we instructed our system to handle all the possible exceptions and errors we observed, so to guarantee the prosecution of the execution.

Eventually we selected 213 URLs (143 organizations) among the URLs that were active at the time of evaluation. Reducing the initial number was required to make it feasible to manually verify the results in a reasonable time.

4.2.3 Creation of the ground truth

Creating a ground truth to evaluate experimental data was a critical part of our system. For each sample we needed to determine which WebInjects were performed and on which websites. This posed two major problems: first of all, WebInjects are configured in a ciphered configuration file. As mentioned in Section 2.4.1, there are decryption tools to decipher configuration files, but we tested them and they never worked for ZeuS versions equal to or higher than 2.1.0.1, which are the great majority in our dataset. Secondly, even assuming that we were able to obtain a deciphered file for each sample, we should

[1] `https://bitbucket.org/davaeron/zeus/`

have manually analyzed it, going through the thousands of lines of configured injections, and verifying if each WebInject was successfully performed in the current version of the website. Of course this should have been repeated for each sample, compromising the automation feasibility.

We opted for another approach: we configured Zarathustra with all the heuristics enabled and then manually analyzed the results to ensure that no false positives were found. Those that were found were always caused by a faulty dump of the page. Adopting such an approach allowed to manually analyze a much lower volume of data and also to identify common injections that could be unmistakably linked to specific WebInjects in the unencrypted `webinjects.txt` files we had at our disposal. As for false negatives, we assumed that by performing a deterministic comparison between two documents it is not possible that actual differences are not detected. Further manual inspections on dumps confirmed this hypothesis.

An alternative approach could have been to craft a proper `webinjects.txt` file as the ground truth. However, we wanted to test Zarathustra on injections found in the wild.

Another methodology we adopted to evaluate performances on false positives consisted in using a clean VM to carry out tests as we do for infected VMs. Ideally no differences should be detected on this VM and each difference is a false positive.

The granularity we use to evaluate our system is not at the difference-level, but at the URL-level: we are interested in signaling an URL as infected or not, independently from the number of differences found. Therefore we only distinguish between two cases: zero differences, corresponding to a clean URL, or higher than zero, corresponding to an URL with WebInjects.

4.3 Environment and deployment

We deployed Zarathustra on a 1.6GHz, 4-cores x86-64 Intel machine with 16GB of RAM. We installed Windows XP SP3 (with Internet Explorer 6) on each VM and granted outbound and inbound Internet access. Zarathustra required 256MB of RAM and 2 to 5GBs of disk space per VM; in our experiments we used 2 to 35 VMs.

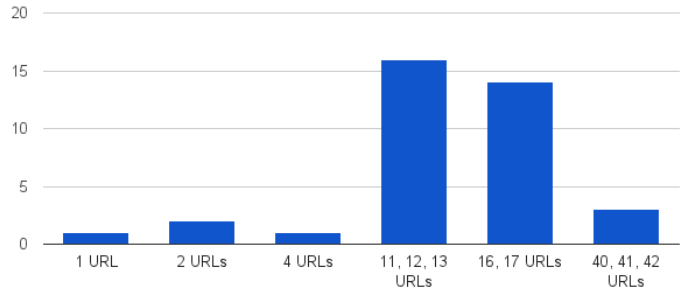

Figure 4.1: Samples grouped by number of different URLs on , which injections were found: different samples have many common injections and the targeted websites are often the same.

4.4 Experiments

4.4.1 Detection capabilities evaluation

Table 4.2 summarizes the top-ten domains where Zarathustra correctly recognized injections caused by ZeuS. Some samples perform zero injections, or they require a longer time to hook the browser APIs. Usually we found around 1 to 9 injections per distinct URL of the same domain.

Table 4.3 summarizes the influence of each heuristic on the detection capabilities: we disabled one heuristic at a time and ran the same experiment. The last row reports the detection capabilities when all the heuristics are enabled: we manually verified the presence of actual injections and set this as the ground truth for the experiments reported in the above rows. Overall, Zarathustra correctly detected that ZeuS was performing an injection in 23.48% of the URLs. The second column is the most important one. It shows the fraction of URLs where Zarathustra correctly detected that a specific sample was performing an injection. We notice that the contribution of the first heuristic is fundamental, because such fraction of URLs decreases to 39.58% (on average) when disabled. The second heuristic also provides a significant contribution to the detection, whereas the last two heuristics are not particularly influential in our dataset.

4.4.2 False positives evaluation

A false positive occurs mainly when Zarathustra detects a legitimate, benign difference in a website. A curious fact is that this could happen a) on an infected machine (if Zarathustra wrongly detects a change in a non-injected website), as well as b) on a clean machine. Although both are, strictly speaking,

Effective TLD	Injections			
	min	max	tot	avg
ybonline.co.uk	0	28	952	9.0667
cbonline.co.uk	0	45	699	2.6885
lloydstsb.com	0	23	677	4.3121
bbvanetoffice.com	0	14	312	5.7778
virginmoney.com	0	279	279	5.6939
if.com	0	77	231	4.2778
banesto.es	0	10	194	0.7239
rbkmoney.ru	0	8	112	2.1132
accessmycardonline.com	0	31	93	1.7547
smile.co.uk	0	29	87	1.6415

Table 4.2: Top ten websites in our dataset. The no. of injections are calculated and averaged over the set of ZeuS 56 samples, and on the URLs within each domain.

Heuristics	Avg. Correct (± Var.)	%URLs
2,3,4	**39.58 ± 11.53%**	52.17%
1,3,4	74.98 ± 15.42%	23.48%
1,2,4	97.97 ± 0.069%	22.61%
1,2,3	98.42 ± 0.124%	23.04%
All	100.0%	23.48%

Table 4.3: Contribution of each heuristic on the detection capabilities. The second column reports the fraction of URLs with correctly-identified injections (this fraction is averaged over the set of 56 samples). The last column reports the fraction of URLs where at least one sample was detected while performing an injection, including false detections. False positives are analyzed separately in Section 4.4.2.

false positives, evidently the false positives found on infected machines are less problematic than on a clean machine. The rationale is that raising an alarm usually leads to some reaction (e.g. investigation, removal), which would waste time and resources on a clean machine; on the other hand, a false alarm raised on the wrong site is beneficial anyways if that occurs on an infected machine.

In our experiments, we obtained zero false positives when using all the heuristics on the entire dataset.

Next, we analyzed the influence of **Heuristic 1**, which was the most effective at eliminating false positives, as the first row of Table 4.3 shows. For this, we disabled **Heuristic 1**; then, on all the URLs in our dataset, we calculated the fingerprints between an increasing number $n \in [1, 35]$ of clean VMs and a) four VMs, infected with four distinct ZeuS samples, and b) four clean VMs. In this way, we can assess how well Zarathustra can tell legitimate differences and true positives apart when using a sufficiently large number of emulated clean clients.

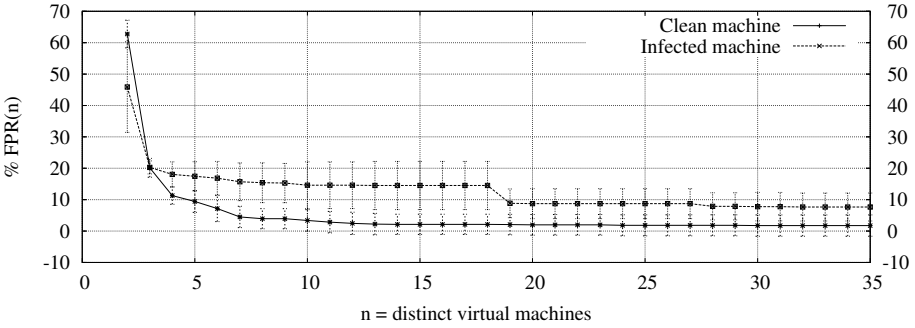

Figure 4.2: False positives due to legitimate differences decrease for an increasing number, $n \in [2, 35]$, of clean VMs, until it reaches 1.0%. We used 206 distinct URLs, rendered on a machine infected with ZeuS (MD5 a4aa162745adcb84373e6a623125c650).

As Figure 4.2 shows, both the false positives, a) and b), decrease while n increases. More importantly, the false positives on b) clean VMs become almost zero (1%) if at least 35 clean VMs are used. These results show that, according to our dataset, **Heuristic 1** is still necessary to achieve zero false positives. However, when **Heuristic 1** must be disabled for certain sites, increasing the number of clean VMs can provide a reasonable substitute.

We manually observed that the vast majority of the remaining false positives in case a), at $n = 35$, was caused by JavaScript-based modifications, which lead to highly-dynamic DOMs. Thus, when deploying Zarathustra to protect from injections on web pages that have a dynamically-generated DOM, it is recommended that either **Heuristic 1** is enabled, or a large number of VMs is used.

4.4.3 Speed and scalability

We measured the execution time of Zarathustra and observed that, as Figure 4.3 shows, it scales well: with 10 VMs running in parallel we are able to process 1 URL for all the samples in less than 3 seconds. However, increasing the number of parallel threads generally results in a lower number of successfully dumped URLs, because all the traffic flows through the single interface between VirtualBox and the host OS, and the proxy represents a bottleneck in our implementation. The time required to match the fingerprints is negligible, because the fingerprints are indexed by URL and the verification of each fingerprint is implemented as an XPath query. Moreover, the architecture of Zarathustra has no central node nor any dependency that prevents full parallel operations: as a result, its capacity scales directly with the amount of resources

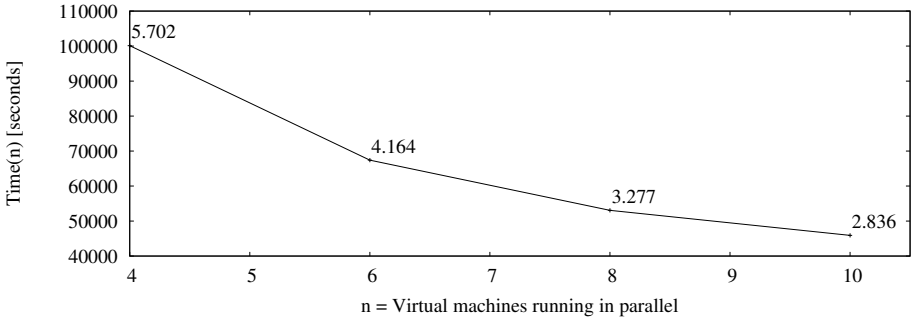

Figure 4.3: Scalability of Zarathustra: Time required to process 213 URLs with 76 samples (including crashing samples). the labeled points indicate the time to process 1 URL.

available.

Chapter 5

Conclusions

We have presented Zarathustra, an automated system that detects on the client side the activity of banking trojans that perform WebInjects. Zarathustra extracts the DOM differences by comparing web pages as they are rendered in an instrumented browser running on clean and infected virtual machines. It builds a model of legitimate differences that we called baseline, and then extracts a fingerprint of the modifications introduced by the malware sample.

Our system has the advantage of requiring no reverse-engineering effort: the only requirement is a binary sample of the malware. Fingerprint generation is completely automated and independent from the malware family. In this sense it is a major improvement over the current state of the art.

Our evaluation of Zarathustra against 213 real, live URLs of banking websites and 56 distinct samples of ZeuS show that, in all the cases, our system detected all the injections correctly. The false positives (about 1%) were caused by legitimate differences in the original web pages. We have developed specific heuristics, which can be safely enabled under realistic conditions that can reduce such false positives to zero. Zarathustra scales well, and can generate fingerprints for 1 URL in less than 3 seconds on average (the time required to match the fingerprints is negligible) even on our modest infrastructure. Furthermore, as fingerprint generation and matching can be performed independently on samples and URLs, the process is fully parallelizable and scales directly with the available resources.

We believe that the approach implemented in Zarathustra, that is the observation of differences in clean vs. infected machines, can be applied to other data that the information stealers need to manipulate. Although simple, our approach has the great advantage of being completely agnostic with respect to the source of the differences: as long as the manipulated data is observable, our approach can be generalized to create further "difference modeling" techniques

that can be used to characterize the activity of an information stealer from other observation points.

5.1 Limitations

Our current implementation of Zarathustra assumes that the banking website is an oracle. For reasons that fall outside our attacker model (e.g. client-side malware), an injection may match exactly with a benign difference. For example, this happens if the banking website is updated with a new form input that has the very same DOM representation of an injection. Not only this is very unlikely to happen, it is also very easy to remedy, by leveraging feedback from the bank whenever their site is updated, or possibly by requesting an update of the fingerprints for that domain. It is indeed reasonable to imagine Zarathustra deployed within a bank information system: this use case would erase most, if not all, the venues for false positives as a full up-to-date model of the clean website would always be available.

Another obstacle that Zarathustra has to face are evasion mechanisms employed by the malware to fool dynamic analysis. However, we do not rely on debugging or introspection tools, against which modern malware adopt specific countermeasures (e.g. refusing to execute). We rely on virtual machines solely for ease of implementation and flexibility during evaluation: Zarathustra works perfectly, and even faster, on bare metal. Hence, this obstacle is easily circumvented by adopting the method proposed by [13] to obtain a virtual-machine-equivalent snapshots on physical hardware. In this way, no malware can possibly recognize that it is running in a controlled environment.

5.2 Future work

As described in Kharouni [12], some attackers are shifting toward more advanced uses of WebInjects, operating more subtle changes, which do not result in user-visible DOM modifications. Although some of these usages can be directly detected using the approach described in this paper, some interact at a lower level with the libraries of the malware, resulting in advanced manipulations of the HTTP requests to divert monetary transactions to a bank account under the attacker's control. The respective HTTP responses (e.g. page that confirms the result of a transaction), and all the subsequent interactions with the banking website, are also modified such that the true recipient of malicious wire transfers is masqueraded (e.g. replaced with the intended recipient's name). This threat will require modifications to Zarathustra, because the in-

jections may occur in pure text nodes. Thus, the set of heuristics will need to be refined to cope with these corner cases.

In Zarathustra we showed that the DOM is a simple yet effective observation point. However, we believe that other aspects of the browser behavior can be observed and compared on infected vs. clean clients, to assess whether the information stealers cause side effects in the browser that can be used as a detection criteria.

Bibliography

[1] Cyber Banking Fraud. Global Partnerships Lead to Major Arrests, January 2010. URL http://www.fbi.gov/news/stories/2010/october/cyber-banking-fraud.

[2] Banking trojans: Understanding their impact and how to defend your institution against trojan-aided fraud. Technical report, Symantec Corporation, 2011. URL https://www4.symantec.com/mktginfo/whitepaper/user_authentication/21195180_WP_GA_BankingTrojansImpactandDefendAgainstTrojanFraud_062611.pdf.

[3] Reversal and Analysis of Zeus and SpyEye Banking Trojans. Technical report, IOActive, 2012.

[4] State and trends of the "russian" digital crime market 2011. Technical report, Group IB, 2012. URL http://group-ib.com/images/media/Group-IB_Report_2011_ENG.pdf.

[5] H Binsalleeh, T Ormerod, A Boukhtouta, P Sinha, A Youssef, M Debbabi, and L Wang. On the analysis of the zeus botnet crimeware toolkit. In *Privacy Security and Trust*, pages 31–38. IEEE, 2010.

[6] A Buescher, F Leder, and T Siebert. Banksafe information stealer detection inside the web browser. In *RAID '11*, pages 262–280. Springer, 2011.

[7] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, Stefano Security Zanero, and Privacy SP 2010 IEEE Symposium on. Identifying Dormant Functionality in Malware Programs. *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[8] Nicolas Falliere and Eric Chien. Zeus: King of the bots. Technical report, Symantec Corporation, 2009.

[9] Max Goncharov. Russian underground 101. Technical report, Trend Micro Inc., 2012. URL http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-russian-underground-101.pdf.

[10] C Grier, L Ballard, J Caballero, N Chachra, C J Dietrich, K Levchenko, P Mavrommatis, D McCoy, A Nappa, and A Pitsillidis. Manufacturing Compromise: The Emergence of Exploit-as-a-Service. In *ACM conference on Computer and Communications Security*, 2012.

[11] Mario Heiderich, Tilman Frosch, and Thorsten Holz. Iceshield: Detection and mitigation of malicious websites with a frozen dom. In *RAID '11*, pages 281–300. Springer, 2011.

[12] Loucif Kharouni. Automating Online Banking Fraud. Technical report, Trend Micro Incorporated, 2012.

[13] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareBox: efficient malware analysis on bare-metal. In *ACSAC '11: Proceedings of the 27th Annual Computer Security Applications Conference*. ACM Request Permissions, December 2011.

[14] Amit Klein. Citadel Trojan Targets Airport Employees with VPN Attack, August 2012. URL http://www.trusteer.com/blog/citadel-trojan-targets-airport-employees-with-vpn-attack.

[15] Brian Krebs. SpyEye Targets Opera, Google Chrome Users, April 2011. URL http://krebsonsecurity.com/2011/04/spyeye-targets-opera-google-chrome-users/.

[16] Brian Krebs. Police Arrest Alleged ZeuS Botmaster "bx1", January 2013. URL http://krebsonsecurity.com/2013/01/police-arrest-alleged-zeus-botmaster-bx1/.

[17] Martina Lindorfer, Alessandro Di Federico, Paolo Milani Comparetti, Federico Maggi, and Stefano Zanero. Lines of Malicious Code: Insights Into the Malicious Software Industry. In *Annual Computer Security Applications Conference*, October 2012.

[18] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. In *International Symposium on Recent Advances in Intrusion Detection*. Springer-Verlag, October 2009.

[19] Louis Marinos and Andreas Sfakianakis. ENISA Threat Landscape. Technical report, ENISA, September 2012.

[20] Tyler Moore and Richard Clayton. Examining the impact of website take-down on phishing. In *the anti-phishing working groups 2nd annual eCrime researchers summit*, pages 1–13, New York, New York, USA, 2007. ACM Press.

[21] T Ormerod. An Analysis of a Botnet Toolkit and a Framework for a Defamation Attack. 2012.

[22] M Riccardi, R Di Pietro, and J A Vila. Taming Zeus by leveraging its own crypto internals. In *eCrime Researchers Summit*, 2011.

[23] Sergei Shevchenko. Config decryptor for zeus 2.0. Technical report, ThreatExpert, 2010. URL `http://blog.threatexpert.com/2010/05/config-decryptor-for-zeus-20.html`.

[24] Aditya K Sood, Richard J Enbody, and Rohit Bansal. Dissecting SpyEye – Understanding the design of third generation botnets. *Computer Networks*, August 2012.

[25] Kevin Stevens and Don Jackson. Zeus banking trojan report. Technical report, Dell SecureWorks Counter Threat Unit, 2010. URL `http://www.secureworks.com/cyber-threat-intelligence/threats/zeus/`.

[26] James Wyke. What is Zeus? Technical report, SophosLabs UK, 2011.