Politecnico di Milano
Scuola di Ingegneria dell'Informazione

Polo territoriale di Como
Master of Science in Computer Engineering
Sound and Music Engineering

Apollo: Eliciting and Analyzing Advanced WebInject-based Malware

Advisor: Prof. Federico MAGGI
Co-Advisors: Andrea CONTINELLA, Prof. Stefano ZANERO

Master Graduation Thesis of:
Samuele RODI
Matricola N. 817854

Academic Year 2015-2016

Abstract

Financial trojans, a particular kind of information-stealing malware, are one of the prevalent Internet threats. Their purpose is to automatically commit fraudulent transactions by silently stealing users' credentials to bank accounts of infected machines. Their level of sophistication has steadily grown in the last few years, keeping up at the same pace with reinforced security measures introduced by financial institutions. The attack schema is devious, as, in many cases, it produces no traces of the attack, leaving the victim unaware of the fraud, often, for a long period. These attacks leverage the API hooking techniques, to install a malicious payload in the victim's browser, in order to steal user credentials or modify web-pages inserting new content (so called web-injection).

We propose an automated system, Apollo, capable of extracting web-injection signatures from financial trojans by analyzing two different versions of the same visited web-page, prior and after the malicious injections, and identifying the portions of the original page source that trigger the malicious behavior of the malware under analysis. The system is able to elicit the malware's behavior on specified web-pages as well as to extract the web-injection targets through dynamic memory inspection.

We evaluated Apollo against a dataset of working financial trojan samples showing that our method successfully extracts correct web-injection signatures together with the corresponding URL targets.

Contents

1 Introduction

Financial gain is the major motivation behind most cybercriminal activities and it is very unlikely for this principle to change in the future. Information stealing trojans (also known as banking or financial trojans) have been for the last few years one of the most effective tools used by cybercriminals to steal banking credentials from unfortunate users, and derive financial gain by committing fraudulent transactions. In 2011, the FBI reported a loss of over USD $85 million for US financial institutions due to account takeovers and wire transfers piloted by banking trojans [1]. Zeus alone, the progenitor of financial trojans, is estimated to have caused damages, in only 2 years, worth USD $100m since its inception in 2007 [2]. In 2014, it was registered a record of more than 4 millions active machines infected by banking trojans. Nowadays, on-line banking frauds are continuously on the rise, and in 2015 a large portion of the total amount is still to be attributed to financial trojans [3].

Financial trojans operate inside the most widely used web-browsers and steal credentials from users as they log-in onto websites of financial institutions. With the stolen credentials, the attacker can commit fraudulent transactions. This type of threat is particularly dangerous because the victim is often unaware of the threat and, even after the transaction is committed, he might receive no feedback or hint about the occurred event. This is because the malware operates in a stealth mode and hides the traces of his actions, for example modifying the displayed account balance after the fraudulent transaction.

Nowadays, countermeasures to such type of threat are still based on prevention mechanisms geared by anti-virus software, and they very often fall short in the detection of new type of threats. Despite the existence of automated techniques for malware analysis, manual reverse engineering is still often necessary in this field, which is a tedious process not always capable of determining the range of action of a malware sample. In particular, in case of financial trojans, the main efforts hinge on correctly recognizing the malware sample version, while very few is known or established about the trojan's targets or main activity.

In the present work we set the goal to propose an automated dynamic-analysis framework able to characterize the behavior of financial trojans, and extract useful information that allows to study the ecosystem in which this kind of malware lives. One of the biggest challenge in malware dynamic analysis consists in triggering the sample's malicious behavior. This allows to understand the malware "modus operandi" and operating conditions. As explained in Chapter 3, we adopt an eliciting approach to trigger the malicious behavior on a targeted website. Based on the findings of previous work [4], which demonstrated the feasibility of web-page differential analysis, we propose a much more usable and efficient approach to extract the modifications that the malware sample performs on the target web-page in order to accomplish its fraudulent scope. As explained in Chapter 4, we trace the modifications performed by a malicious sample using API hooking techniques, ironically, the same used by malware authors, in order to extract both a clean and an injected version of the web-page and deduce DOM modifications.

In Chapter 5, we provide evidence and the proof of concept that our method is viable and returns the expected results under given circumstances. We encountered difficulties when testing on real samples, because of the lack of a verified ground truth on the samples being tested as well as a nearly total absence of information on the sample dataset. This does not determine the invalidation of our method, but it simply assumes conditions that are only arguably met in the testing scenario performed on real unknown samples.

In conclusion, the project reorganizes the previous work on Prometheus [4] and Zarathustra [5] under a different method, with the goal of providing more accurate and less biased results than in previous experiments. We verify the validity of the proposed approach introducing some original concepts:

- An innovative procedure: we stimulate the malware's malicious behavior on a target web-page through the modification of the original page source, in order to spot the hooking points of the injection.
- We propose a method to dynamically inspect the malware memory to retrieve sensitive information.
- We wrapped the whole systems in a dynamic and versatile solution, capable of executing automated analysis.
- We performed experiments on a custom set, successfully validating the web-injection signatures and proving the potential of the adopted approach.

2 Motivation

In this section we expose the overall problem of financial trojans, from their inception to the current state of the art. In particular, we focus on their most-dangerous component, the WebInject module, and its operation schema.

2.1 The widespread scenario of digital banking frauds

Financial trojans have been one of the prevalent threats on the Internet over the last decade. The dawn of this type of threat has its origin in 2007 with the introduction of the Zbot, mainly known as Zeus, in underground forums. Zeus, a malware package that allows full control by an unauthorized remote user to an infected machine, has been conceived with the primary function of financial gain by stealing credentials such as email, on-line banking credentials, or other passwords, and (as of 2009) it could be purchased in black markets for as low as 700$ [6]. Originated in Russia and continuously evolving over time, Zeus, have threatened worldwide financial institutions until 2010, when its source-code was leaked and the "rights" to sell the kit were given to the then biggest competitor and descendant, the creator of the SpyEye Trojan.

After that, an ever increasing number of banking trojan families developed in more and more complex forms and are being constantly updated and adapted to thwart modern defense mechanisms. Cybercriminals have now taken things a step further with the help of automatic transfer systems (ATSs). They consist of complex code injected in the web-page capable of performing automatic wire transactions directly on the infected machine or check account balances using the victim's credentials, without alerting the user. ATS scripts can even modify account balances after illicit operations in order to completely hide traces of the malware activity on the victim's machine [7]. The automated on-line fraud schema also drastically eases attackers' life as they no longer need user intervention to obtain money.

In this context, many attackers are involved not only in simply participating

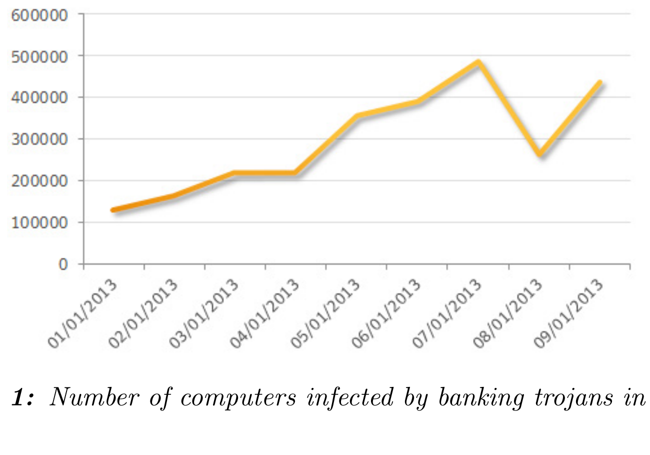


Figure 1: *Number of computers infected by banking trojans in 2013 [8]*

to financial frauds, but are now also actively dedicated to creating tools to facilitate these activities. Attackers can leverage third-party services to operate more efficiently and can even outsource the cash-out process. Compromised banking accounts are traded for 5 to 10 percent of their current balance [9].

Information-stealing trojans are a sophisticated threat, which has grown steadily until 2014 as shown in Figure 1. Financial trojans compromised millions of computers and targeted user accounts of over 1000 financial institutions [10]. When looking at the targeted regions with the highest financial trojan infection rates, the US have always ranked first within the last five years with the largest portion of total infections count. In the top 10 positions it is also possible to spot many European countries, Japan, India and Canada (see Figure 2).

The main issue related to banking trojans is that they are easily purchasable in online black markets, consisting of darknet/individual websites, forums and chat rooms. Underground marketplaces provide all the required resources to build a custom sample version, as a service industry under the exploit-as-a-service model. In this scenario, anyone, independently from its technical competences, can perform financial frauds by buying malware kits which get sold in a range between 100$ to 3000$. Cybercriminals often offer also paid support and customization, like advanced configuration files for end users to include in their custom builds. The Russian underground economy of cybercriminals is very active and dangerous, hosting several types of "crime services", and it has been estimated to be worth in total $2.3 billion dollars [11].

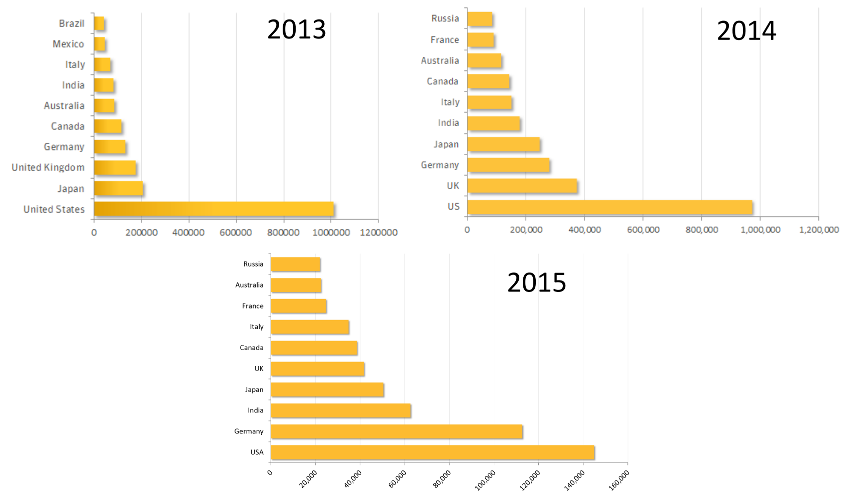


Figure 2: *Number of computers infected by banking trojans per each country in the last three years [10]*

2.2 The malware-driven information stealing architecture

The attack schema deployed by financial trojans is very complex compared to other common type of frauds and requires a strong knowledge of computer systems as well as a successful matching of several fortuitous conditions. For these reasons it is recognized as one of the most sophisticated financial fraud architecture and often disregarded because of its excessive complexity.

2.2.1 The fraud schema

On-line banking services are active since 1994. Using a Web browser, it is possible for customers to log into the banks' secure websites and view their personal account balance or perform financial transactions. Since then, on-line banking has encountered a fast increasing popularity among customers of global financial institutions and this new trend has rapidly caught the attention of cybercriminals. Financial gain became a concrete opportunity for cybercriminals, that had lived until that time mainly seeking after notoriety and fame. Originally, attacks involved a variety of methods including simple keylogging trojans or phishing emails able to intercept user credentials and by-pass most of the security measures which were not yet strongly enforced. As financial institutions enhanced cybersecurity and fraud detection systems, cybercriminals had to adapt and attacks gradually required more and more sophisticated and functional trojan software [12].

The European Network and Information Security Agency (ENISA) advises financial institutions to adopt security measures that assumes user devices are compromised. This global communication led institutions to introduce the use of transaction authentication numbers (TAN or OTP) and two-factor authentication (2FA) methods. These methods requires a legitimate user to provide evidence of owning a secondary factor or device in order to get the authorization for a transaction. In many implementations, the second factor is represented by an external key (Figure 3) provided by the financial institution handed personally to the customer, which continuously generates one-time-passwords (OTP) valid for a single operation and expiring in a short period of time (usually 1 minute).

As a consequence, financial trojans evolved, aiming at stealing also the OTPs

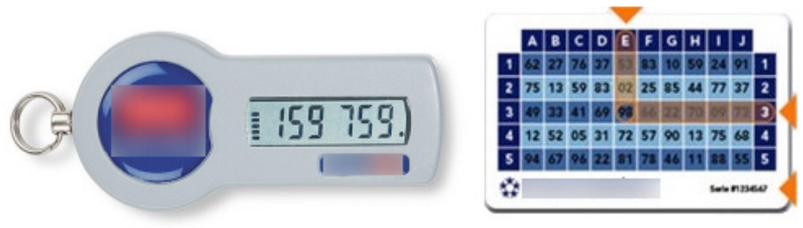


Figure 3: *Example of weak authorization mechanism with OTP-token (or TAN) [6]*

required to perform fraudulent transactions. Financial trojans are, indeed, able to install themselves into the victim's browser and modify on-the-fly the content of web-pages using the so called man-in-the-browser paradigm. In a typical attack vector, whenever a user on an infected computer access a targeted banking website, upon a customer login, the trojan injects into the bank web-page a field requesting the OTP, usually in conjunction with a message stating that the additional field was recently introduced to enforce security measures. When the unfortunate users insert both the credentials and the OTP, the trojan intercept the data and sends them to the C&C server, providing the attacker with all the pieces necessary to commit an authorized transaction. Most advanced implementations use ATS scripts injected in the web-page which automatically perform transactions directly on the infected client and modify the account balance or the transaction specifications displayed on the browser so to remain completely undetected by the unsuspecting user.

Recent web-injection configurations often also target the departments in banks that deal with corporate customers, in order to fish for high quality accounts. Ready-made configurations can be found for less than US$100 in underground markets [9].

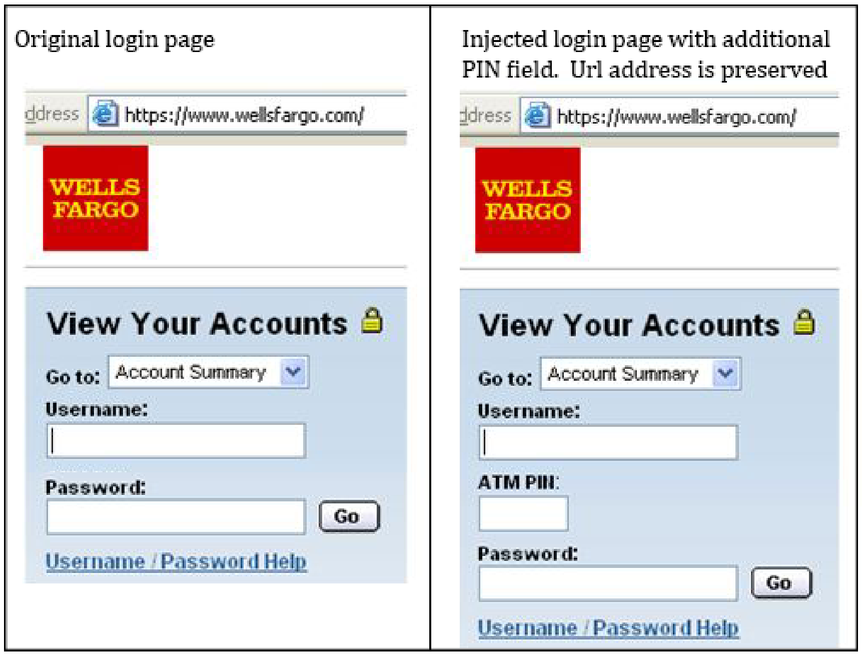


Figure 4: *Example of browser web-injection in login page and comparison between clean machine (left) vs. infected machine (right)*

2.2.2 The Man-In-The-Browser Attack

Still nowadays, the most common and well documented method used by malware for financial frauds is the Man-in-the-Browser (MitB) attack. It allows the trojan to locally modify all the traffic from and to the browser. This idea was first presented by Agusto Paes de Barros in 2005 and adopted from 2007 for financial frauds [12]. MitB exploits the API hooking techniques and involves a WebInject module (the malicious component responsible for web-injection) to inject into the browser process with the goal of manipulating data before it is displayed. It usually targets the most popular browsers, such as Internet Explorer, Firefox, and Chrome. Upon infection, the WebInject module finds its place between the browser's rendering engine and the API functions that allow to send and receive HTTP(S) data. By hooking high-level API communication functions in user-mode, the WebInject module can intercept the traffic without raising any sign or suspicious alert to the user since installation in kernel mode is not necessary, and thus performing more conveniently than traditional keyloggers [4]. The peculiarity of the WebInject module is its effectiveness even in case of an HTTPS connection which is, on the contrary, resistant to Man-in-the-Middle attack. As displayed in Figure 5, the user hooks attach itself right after the SSL communication decryption so that all the traffic intercepted through this channel results in a clear state. A Man-in-the-Browser attack shows up only at the browser presentation layer, i.e. right before the page is rendered and any script executed on it. There is no obvious indication of malicious activity; the domain is legitimate and the security certificate has not been tampered with, which all adds credibility to attacker requests and can end up fooling the user.

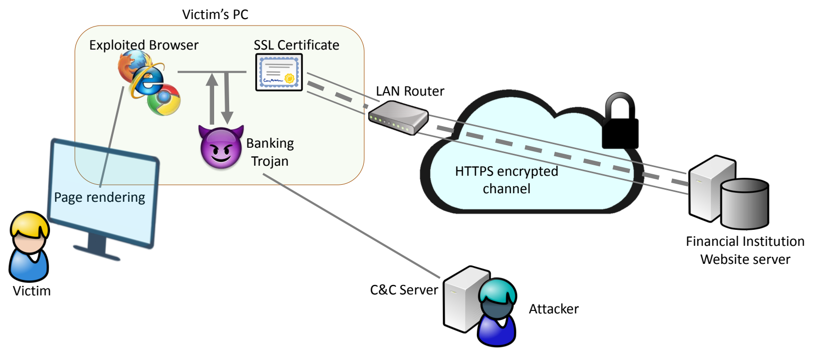


Figure 5: *Schematics of the Man-In-The-Browser paradigm. The trojan lives in the browser intercepting user credentials before they get encrypted by HTTPS protocol.*

2.2.3 The WebInject module

In the MitB paradigm the WebInject module plays a fundamental role. Installed in the victim's browser, it occasionally communicates on a scheduled basis with the C&C server of the Botnet in order to retrieve the necessary configuration files which are only temporary stored on the victim's machine in an encrypted format. The configuration for webinjection resides on the server and is built from a simple *webinject.txt* file that the attacker can modify to specify custom target websites. Often, the *webinject.txt* file comes pre-compiled within the purchased kit with a set of URL patterns targeting worldwide financial institutions. The target list included an average of 56 patterns per sample as of 2014 against an average of around 283 URL patterns in 2015, targeting around 93 different institutions [10].

Below is an example of configuration block defined in the *webinject.txt*:

```
set_url http://www.abankwebsite.com/login.php GP

data_before
name="email"*</tr>
data_end

data_inject
<tr><td>PIN:</td><td><input type="text" name="pinnumber" id="pinnumber"/></td></tr>
data_end

data_after
data_end
```

Listing 1: *Sample WebInject Configuration Block*

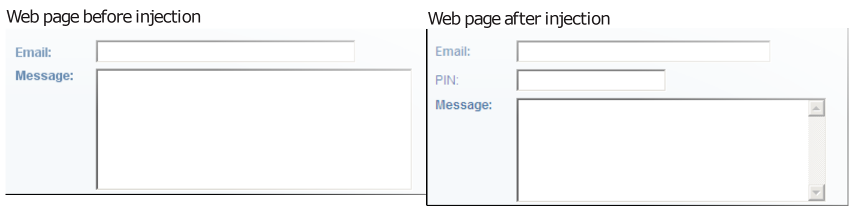


Figure 6: *Visual result of the web-injection performed in Listing 1*

The present example operates on the Web page matching the URL `http://www.abankwebsite.com/login.php`. The HTML defined by *data_inject* represents the text to be injected in the page, to be inserted after the string represented by *data_before* field. The syntax also allows for HTML to be replaced by specifying the *data_after* field. When this field is specified, then the HTML specified by *data_inject* will replace the HTML content between *data_before* and *data_after* [13]. The given configuration file produces the result depicted in Figure 6.

2.3 The state of the art of financial trojans

Banking trojans have represented one of the major threats for financial institutions in the last decades and their infection rate has grown steadily until March 2014, counting more than 4 millions of compromised computers in that year [9]. However, after that peak, numbers have changed due to several factors which include botnets takedown, enforced security measures and a market change.

2.3.1 The current state of adoption of information stealing malware

In the last two years the infection rate of information stealing trojans has sensibly decreased resulting in a decline of 73% as of 2015 and it has now reached infection levels similar to that of 2012 [10]. Unfortunately, this is a misleading findings as someone might think the problem is going away, but, instead, the drop is to be attributed to different causes.

The drop followed various takedown operations and malware author arrests performed by the FBI and the European Cybercrime Task Force, which were carried out in early 2014. The downtrend referred especially only to the Zeus family with all its descendants, which all in all represents the largest portion of financial trojans, counting a drop from nearly 4 million infections in 2014, to just under 1 million in 2015. This is an indication that cybercriminal groups are moving to other, more advanced, financial malware families with similar features, like Dridex and Dyre, experiencing a drastically smaller share but upward trend (Figure 8). Dridex infections increased by 107 percent in 2015, making it the fastest growing family of financial trojans [14].

The downward statistics are also very likely determined by the rapid rise of ransomware, a different type of malware, which encrypts user personal data and asks to pay for a ransom to get the decryption key. This prominent type of market have very likely allured a good number of cybercriminal groups which

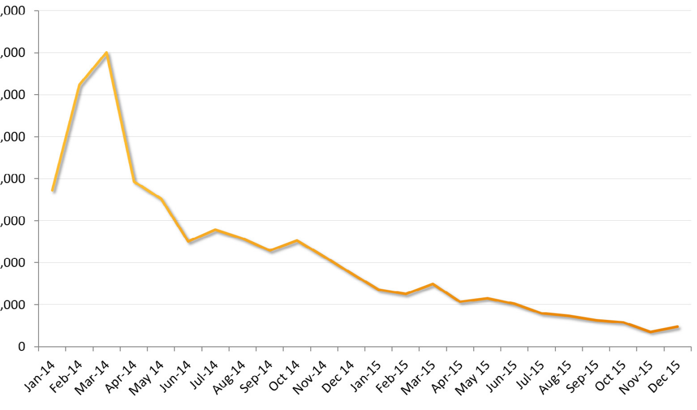


Figure 7: *Showing major drop of computers infected by banking trojans between 2014 and 2015 [10]*

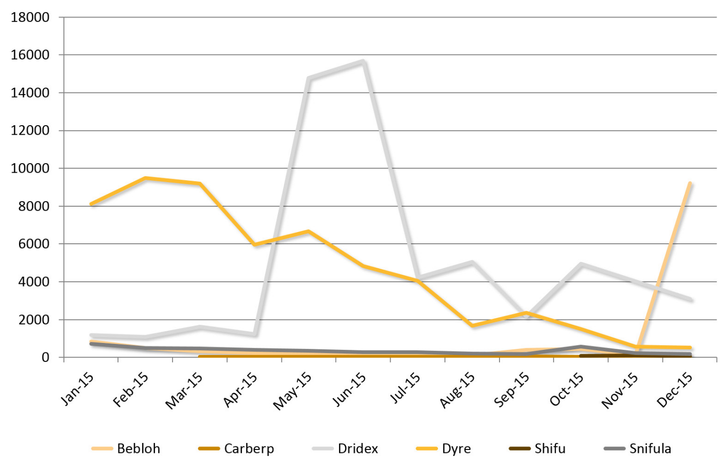


Figure 8: *Computers infected in 2015 by the top financial trojan families excluding Zeus family [10]*

stopped to send out financial trojans to focus onto other money making schemes through this promising type of malware, capable of spreading panic amid the population [10].

The tendency might also be the reflection by the ever stronger security measures due to the introduction by several banks of the two-factor authentication using One Time Passwords (OTPs) sent by SMS, as a method to counteract financial trojans and the MitB paradigm. For this reason, in the last years most of the banking trojans have evolved and include in their toolkits a mobile component. This mobile component works in pairs with the PC version and can access all the information in the user's phone, intercepting SMS, and sending it to its C&C server. This attack scheme is also known as "Man in the Mobile" (MitMo) [4]. As depicted in Figure 9, once the victim's PC is infected, when the victim visits his online banking website the trojan steals his credentials and inserts a message in the web-page that invites the user to download and install a new mobile application to be able to access his account from his smartphone. This step is usually performed inserting in the web-page a QR code that points to the malicious application's download. As the victim downloads and installs the mobile malware, his phone gets compromised. The mobile malware can now intercept all the SMS, silently avoid the system notifications and remove them after they have been sent to the C&C.

Also in this circumstance we might find a possible reason of the downtrend of banking trojans infections in recent years. Indeed, despite malware attack evolution, such security measure has made attacker's life harder as traditional banking trojans need to compromise two devices, in such scheme, to commit a fraudulent transaction. At the same time, the interest has reasonably moved directly onto the mobile platform, with malicious mobile applications capable of information stealing directly onto the mobile device. Indeed, with the diffu-

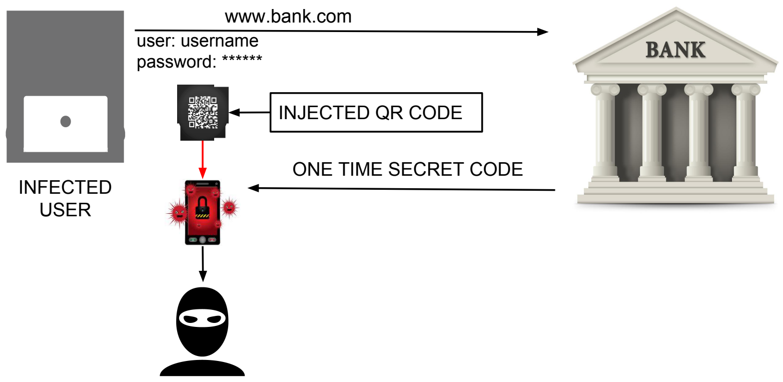


Figure 9: *MitMo scheme. The victim downloads the malicious mobile application through the QR code, allowing the attacker to intercept OTP sent through SMS [4]*

sion of smartphone technologies, the number of malicious Android applications designed to steal financial data rose almost 500% in half a year in 2013, from 265 samples to 1321, with steady growing pace up to present [15].

On the contrary, Kaspersky Lab in 2015 observed an increase in financial trojan infections and predicts that trend to continue in 2016, meaning that the downtrend is very likely just a temporary change of the market [16]. Financial gain is still one of the major motivations behind most cybercriminal activities and there is little chance of this changing in the near future. For such reason financial trojans still represent a major threat. In fact, customer data could be even under greater threat in future, as some banks are having discussions about removing the use of 2FA for smaller transactions to save costs and incentive usability [10].

2.3.2 Research advancements and related work

Anti-virus software is constantly fighting information stealing malware, but its approach to malware principally focuses on virus detection rather than description and correct classification. Most anti-virus vendors are mainly interested in detecting and removing a threat rather than analyze and determine its modus operandi. Also, they can offer an acceptable detection rate only after a signature is generated from a given malware observation, while they often fail in identification of new malware samples. This allows cybercriminals to evade signature detection by updating regularly their samples' executable or even by simply packing and obfuscating each sample with different routines during execution [17].

The malware analysis is often based on reverse engineering techniques which poses the objective to statically analyze malware binaries and extract its functionality even if they are encrypted or obfuscated. Despite its effectiveness, reverse engineering is a tedious and time-consuming practice which very often lacks of generality and the results only applies to the currently analyzed sample. In a different fashion, a more reliable automatic classification can be achieved only by exploiting dynamic behavioral signatures based on the interaction between an application and the operating system. This is also suggested by the huge amount of malicious samples, generated by the continuous releases of different families or versions, customized or simply differently obfuscated, which make it practically impossible to perform manual reverse analysis on each one.

In the direction of automatic dynamic analysis, a certain number of researches have already been attempted. Buescher et al. [18] effectively attempts to detect the illegitimate software manipulation to the browsers' networking libraries, which information stealing trojans use as part of the MitB paradigm. The current project recalls the objective of a previous work [4]. In the precedent study, Prometheus detected injections to generate signatures through memory forensic techniques and through the comparison in DOM differences between the page sources downloaded from a clean and an infected machine. Also the work of Kapravelos et al. [19] is related to ours as they propose an eliciting approach to detect malicious behavior of browser extensions. In particular, they leverage the use of "HoneyPages" and the principle that some extensions activate based on the content of a web-page rather than the URL, in order to extract injection elements. In this context, HoneyPages are custom crafted web-pages fed to a malicious extension to elicit its malicious behavior and detect on which page patterns it performs dynamic modifications.

2.4 Goals and research challenges

Our project deals with the analysis of Web-Inject based trojans and sets the objective of extracting information about their operation mode on an infected machine. In particular, we focus our scope on the detection and deduction of the behavior of information stealing malware on websites hit by web-injection. As better explained in Chapter 3, we propose to derive the content of the WebInject configuration file (*webinjects.txt* or any derivative) by using an eliciting approach, looking at the evidence of injections directly inside the retrieved page source. We focus on the extraction of triggering patterns for a web-injection and injected strings inside the page.

The approach we adopt aims at extracting pieces of information in a completely dynamic fashion, at malware run-time. This is a necessary consequence of the WebInject module architecture. Indeed, using a simple static reverse analysis, it would be impossible to extract the WebInject configuration files as these are downloaded and decrypted by the sample at run-time and remain persistent only on the C&C server. In this context, studying the dynamic behavior is particularly hard because it is impossible to predetermine the state of activation of a sample until an experimental observation is made on it. In most cases, information stealing trojans need to be active with working communication with the C&C server and they also need to be up-to-date for a dynamic analysis to be effective, which are requirements seldom met in real case scenarios. This is especially true during the analysis phase that usually exhibits limited time resources during which the examined sample might lie dormant for an undefined activation period.

The objectives of this work are the following. First, we want to develop a platform for analyzing banking trojans at a high level of abstraction, independent of the malware implementation, which derives the WebInject signatures of the analyzed sample. The key idea, is to inspect the visible modifications that malware causes in the page source of a given website. Differently from previous works, we want to detect such modifications by comparing two versions of the web-pages, extracted on the same infected machine with a single Internet request. This innovative approach aims to filter out the large number of false positives detection caused by modern dynamic web-pages, which render differently on different machines, or even simply at different request timing.

Further, we want to combine the web-page differential analysis with dynamic memory inspection in order to recover, at least partially, the valuable content of the encrypted configuration file. The memory inspection is a fundamental step to determine the operational range of the malware, useful, in a later stage, for the differential analysis. Differently from previous attempts, we perform memory inspection completely at run-time and we define hooks to a very small region of memory where we suspect the analyzed trojan to write sensitive behavioral configuration data. This is a novel approach which guarantees performances and speed and also diminishes false positives detection by focusing the research scope on a restricted target area.

Finally, our ultimate goal and challenge is to perform a generic analysis which do not leverage any malware-specific component or vulnerability, so to tie their applicability to any kind of WebInject-based malware.

3 Approach

In this chapter, we aim at providing an overview of the approach adopted in order to detect the web-injection performed by a malware sample on an infected machine, both in a generic fashion and a more detailed one.

3.1 General overview

The malicious code injected in a targeted website, is responsible of modifying the requested web-page by inserting some additional fields, like an OTP (One-Time-Password) input, necessary to conclude an on-line transaction. In this attack type, it is also possible to delete or change elements of the page in order to override any protection mechanism or substitute an unwanted element. Data, that an unconscious user inputs, are intercepted by the trojan and generally sent to the C&C server which uses this to take illicit actions, like committing a transaction. The main purpose of the present project is to detect a web-injection performed by a malware sample on a given web-page, and determine how the page was exactly modified. For this purpose, we adopted an eliciting approach aiming to tease the malware until it exhibit an interesting behavior. We repeatedly propose to the analyzed sample a continuously modified version of the web-page in order to detect the expressions which trigger the malicious behavior. The other goal of the project is to retrieve a feasible list of URLs for each sample representing all the websites where the malware is likely to perform web-injection, with the goal of determining the range of action of the malware.

3.1.1 The web-page extraction

The method proposed to detect web-injections relies on a dynamic analysis which leverages the usage of an elicitation paradigm. The analysis is performed in a controlled and automated environment where the infected machine visits a web-page which has been a-priori determined to be one of the analyzed sample's targets.

The first objective of the analysis is to extract the web-page source, also referred to as DOM (Document Object Model) as its intrinsic representation, in its integrity and not yet compromised by the malware. In a similar manner, the page source needs to be extracted also after the malicious injection in order to carry out a comparison. It has been proven that, in order to get a consistent and repeatable representation of the DOM and to perform a trusty comparison between the original and infected versions, it is necessary to retrieve those under the same exact conditions (same machine, same environment and session). Nowadays, it is well-known that most of the web-pages are dynamically generated, enhanced with customized content, like advertising, so that the same URL request could produce two different DOM results on two different machines, different browsers or even under different request timing (like different timestamps). For this reason, in our case we chose not to employ different machines to extract the two pages (a clean and an infected one) as done in previous experiments [4]. Instead we adopted a different approach, adding a layer of complexity to the problem. Figure 10 represents a high-level abstraction of the page extraction and preliminary detection procedure.

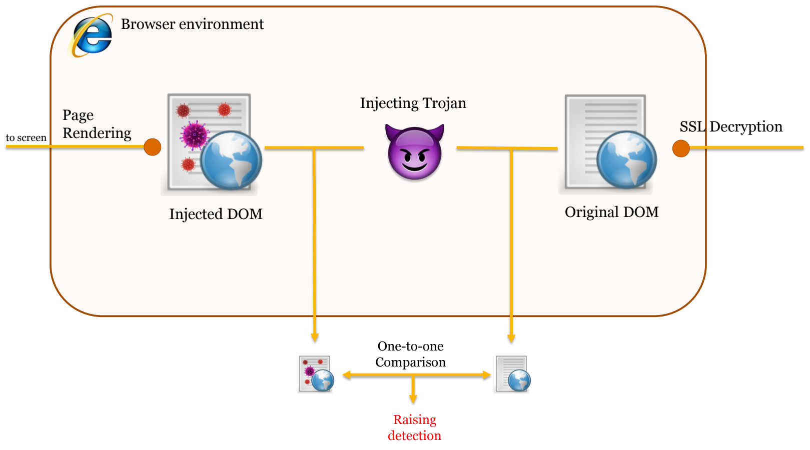


Figure 10: *Extraction schema and detection. The original DOM is retrieved prior and after the malware interception and a comparison is performed in order to detect any web-injection*

3.1.2 The injection detection through web-pages comparison

The page source is intercepted prior and after the malicious injection, representing respectively a clean and an infected state of the web-page. The two versions are compared to locate the injection (Figure 10), as an added, modified or deleted field in the page. Our purpose is also to determine the triggering expressions for each web-injection, i.e. the portion of HTML page that triggers the malicious injection of code, which are defined by the $data\_before/data\_after$ fields in the WebInject configurations file. Here, the elicitation paradigm proposes to repeatedly return to the malware a modified version of the page source until the sample either executes or not the observed injection. The elements in the page mostly sensitive to the discontinuous behavior of the sample are assumed to be the triggering patterns for the analyzed web-injection. The goal of the elicitation approach is to derive the minimum set of characters or expressions which trigger the malicious injection in the page.

3.2 Technical overview

The proposed approach is based on the following technological aspects.

3.2.1 Injecting the browser

Several types of malware show the feature of attaching themselves to legitimate running processes with the purpose of stealing information or simply messing up program execution. This execution mode is named thread injection and allows for a thread to execute in the space of an already running process while sharing the same memory space.

In the Windows environment, thread injection is possible thanks to DLL (Dynamic-link library) execution. DLLs are essentially compiled libraries which get linked only at run-time and extend or implement a program functionality. DLLs are comparable to normal executable, with the difference that they do not have an entry point so that it is not possible to directly execute a DLL, but it requires an EXE (Windows executable) for the operating system to load it. The practice of DLL usage have several advantages in software development. For example, it provides a mechanism for shared code and data, allowing for example for code modularity. Modularity allows changes to be made to code and data in a single self-contained DLL shared by several applications without any change to the applications themselves and without requiring applications to be re-linked or re-compiled.

It can be said that DLLs are the core of the Windows environment as they are extensively used by programs, but despite their advantages, they are also the key to several vulnerabilities. Indeed, DLLs execute in the memory space of the calling process and with the same access permissions which means both little overhead in their use but also no protection for the calling EXE if the DLL has any sort of bug or malicious functionality. In the case of banking trojans, they exploit the thread injection technique to attach themselves onto a running browser process with the goal of accessing their memory space and modifying data.

For the present project, the same approach has been adopted by using a custom injector which loads the DLL detection module onto the Internet Explorer process and actively "listen" to any DOM modification or other type of activity.

3.2.2 API hooking and DOM interception

Modern malware components make extensive use of function hooking techniques to implement several of their features. Function hooking consists in intercepting program function calls or messages passed between software components. Code that performs such interception can freely execute anything prior or after the hooked function call, including copying, modifying the function arguments, or even totally redirecting the method. In practice, a function hook subscribes itself as the legitimate function that is being intercepted and gets executed on behalf of it. This technique has been conceived for several purposes, including debugging or extending program functionality.

The hooking practice is a technique which can be theoretically extended to any function within a program, however, under real circumstances, the exact address of the target function needs to be known, and this represents a strong requirement when dealing with compiled or proprietary code. For this reason hooking is mainly used to intercept system APIs or external library methods, that are dynamically loaded in the target program.

In the current context, the thread injected in the browser implements hooks to all the browser functions responsible to handle the communication between two end-points. In the case of Internet Explorer, all the Windows API residing in the WinINet library are those responsible of requesting a URL, opening a connection or downloading a file, including a website page source. Through the interception of those APIs in Internet Explorer, it is possible to inspect and/or modify the content of a page source since the DOM is represented by an embedded parameter within some of the WinINet functions.

3.2.3 Detecting DOM modifications

For the present approach, in order to actively detect any modification to a given page source, a comparison between an original and an infected version of the same page is fundamental. This means that, the page source needs to be extracted twice from the browser process, precisely, once before the malicious web-injection and once after.

By acknowledging the fact that trojans use basically the same mechanism to perform web-injection, it is important to control the hooking process so to get all the hooks in the correct order, i.e. firstly the one retrieving the original page version, secondly the malware hook performing the injection, and lastly the one extracting the infected version. Once the correct hook order is established, the web-injection can be detected by comparing the two versions

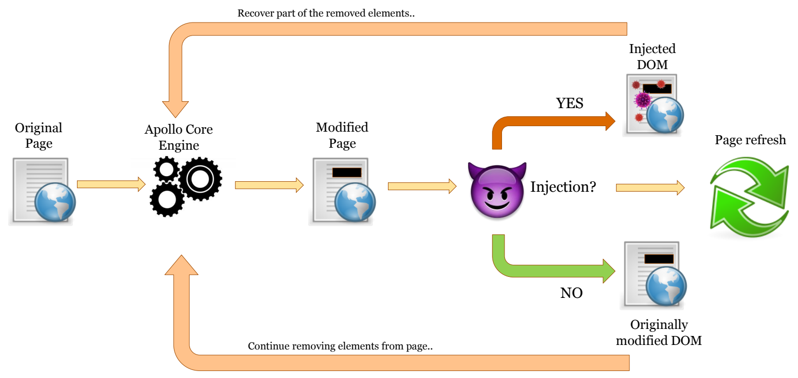


Figure 11: *Signature extraction schema. The page is refreshed repeatedly and the original content modified according to the presence of the injection*

of the page extracted by Apollo. Beside the exact injection, we are interested also in spotting the triggering pattern of the injection. As described in section 2.2.3, an injection gets executed whenever an element of the page matches the field in $data\_before$ or $data\_after$. Therefore, to extract the content of these strings we decided to elicit the malware behavior by re-proposing the same page in a modified version where few characters in the region around the detected injection were removed. By doing so, if the removed characters constituted part of the triggering expressions, then the malware would not find any matching pattern and the injection would not get executed. In this way, it is possible to address such set of characters to the triggering expression. The process is thus iterated until all the relevant characters in the nearby region of the web-injection are analyzed so to derive the full matching pattern. Figure 11 depicts the entire work-flow. The procedure is first carried out analyzing the $data\_before$ field, i.e. the injection begin, and then it is repeated in nearly the same manner for the $data\_after$ field, i.e. the end of any data replacement.

3.2.4 Extracting the WebInject targets

The present approach makes the strong assumption that the analysis be conducted on a URL which is known to be one of the sample targets. This requirement, even if quite trivial in principle, represents a big hurdle in phase of testing when it is unknown on which website is important to check the presence of an injection. Also, it is far from being trivial the problem of extracting a feasible URL list from a malware sample, wrapped in a generic solution for any malware family. We thus proposed a solution to the problem in an experimental form which could allow us to extract a plausible list under few circumstances.

During the analysis of Zeus code, we observed that, in order for the sample to perform injection on a website, a necessary check of the visited URL against a target list is performed by the malicious sample. This comparison occurs intuitively upon page loading, like in Zeus family, and in particular during the malicious hook to the Send request function. This is, indeed, the function which opens up the connection with the chosen website and incorporates the URL parameter. In Internet Explorer, which uses the WinINet library, the send request is generally called by the *HttpSendRequestW* function.

The comparison with the target list is, for good reasons, performed locally and the list stored in memory in clear right before the comparison. The target list, a dynamic element of unknown size, is likely loaded and unloaded in the Heap memory during each send request in order not to leave permanent static access to the malicious resources. Also, it is very often reloaded, after a download from the C&C server, in order to keep the malicious functionality up-to-date. Given such considerations, we decided to track the memory allocations during the send requests. We hook the HeapAlloc and HeapFree Win32 API so to read the entire memory region in the precise time interval of the send request, looking for URL patterns. Considering that Zeus is not only the mostly diffused banking trojan, but also the only one with a leaked source, we based the current approach on these findings, derived from the study of its source code.

4 Implementation Details

In the following chapter we inspect the process work-flow, focusing in detail on each one of the analysis steps. We identified 4 principal phases in the analysis as in Figure 12. In the first, we launch the browser, injecting into the process the module used to perform a dynamic memory dump. In the second phase, we look for URLs in the memory dump and validate the whole list using a search engine. In the third we push further the analysis by detecting web-injections performed on any of the URL validated in phase 2. In the last phase we analyze each web-injection detected, in order to derive the exact injected string together with the triggering patterns.

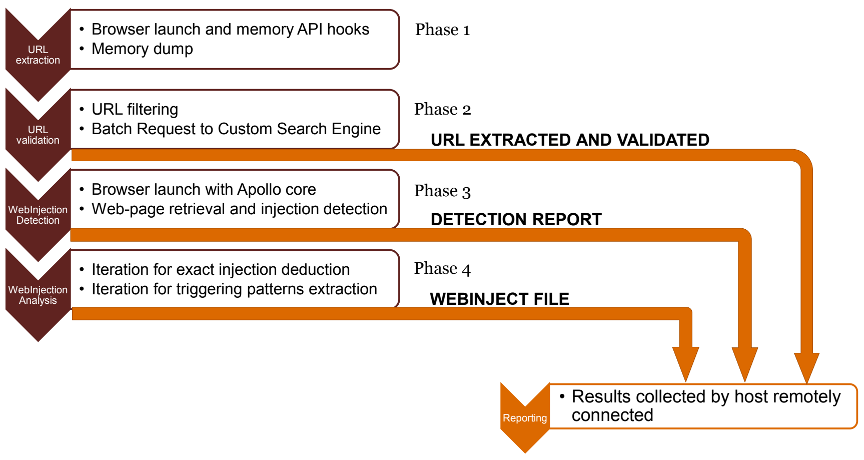
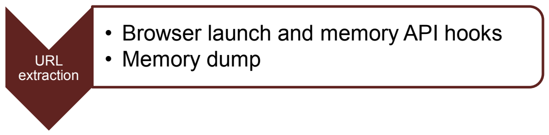

URL extraction
• Browser launch and memory API hooks
• Memory dump
Phase 1
URL validation
• URL filtering
• Batch Request to Custom Search Engine
Phase 2
URL EXTRACTED AND VALIDATED
WebInjection Detection
• Browser launch with Apollo core
• Web-page retrieval and injection detection
Phase 3
DETECTION REPORT
WebInjection Analysis
• Iteration for exact injection deduction
• Iteration for triggering patterns extraction
Phase 4
WEBINJECT FILE
Reporting
• Results collected by host remotely connected

Figure 12: *Overall process work-flow*

4.1 Phase 1: Dynamic Memory Dump



The first phase of the analysis consists in the dump of the browser dynamic memory performed during the connection request to any website. This task is accomplished in order to retrieve a list of likely targeted URLs where to perform the subsequent analysis. Supported by the study of the Zeus malicious code we realized that a malicious sample is necessarily needy to compare the visited URL against a set of URL targets, in order to know where to perform web-injection. This list is very likely loaded dynamically into the browser in the Heap Memory location. In the present analysis, we assume that the sample loads and unloads the list every time a send request is issued to a remote website. This behavior is necessary to the malware as, in the case of the ZBot, the list is saved encrypted in a region of the registry, updated regularly by the malware core and only during a connection request the list is deciphered and the matching performed against the visited URL. In the current implementation we also make the strong assumption that the web-injection module stores, in the Heap memory, the URL list directly in clear and does not make any modification to that memory region before deallocation.

Given such considerations we decided to hook the Heap Memory functions in order to intercept any activity performed to memory and dump all the memory allocation involved in the operation. In particular, we filtered out all the possible legit function calls in a temporal domain, by attaching and detaching the hook to memory prior and after the send request call.

Upon implementation, in order to perform the API hooks we used Detours[1], a Microsoft software package for re-routing Win32 APIs underneath applications. The module to extract URL gets injected into the browser after launching and hooks permanently all the network API of the WinINet library. The Internet Explorer 8 version running under Windows 7 uses the HTTPSendRequestW function in order to request an Internet resource through a URL on a remote server. For this reason, we implemented our temporal filter by attaching the memory hooks prior to this function that we used as a reference. In case a malware is performing web-injection on Internet Explorer, the HttpSendRequestW is very likely the place where matching with the requested URL is performed. Indeed, this function is the preamble to the InternetReadFile, which performs a batch download of the resource, and a malware very likely adopts it to understand if the web-injection onto that site will be necessary, as Zeus does. In this time frame, when we call the $Real\_HttpSendRequestW$, we are instead calling the malware hook to the same function, and we also start the interception of all the interesting memory function. We are therefore sure that the malicious WebInject module executes with the memory hooks attached. Below is a commented excerpt of the hook to the HTTPSendRequestW:

```
//Hook to the HttpSendRequestW
BOOL __stdcall Mine_HttpSendRequestW(   HINTERNET hRequest,
                                        LPCWSTR   lpszHeaders,
                                        DWORD     dwHeadersLength,
                                        LPVOID    lpOptional,
                                        DWORD   dwOptionalLength)
{
 ...                                 //request check. IRRELEVANT
 if (!attached){
  urlMemPointerStart = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, MAX_HEAP_BLOCK_SIZE);
                                     //Allocates memory for the dump
  urlMemPointer=urlMemPointerStart;   //Memory address used to save the dump
  urlMemSize=0;                      //Control variable

  ...                                //Detour Transaction Preamble. IRRELEVANT
  ATTACH(HeapFree);                  //Hook to memory functions: HeapFree
  ...                                //Commit transaction and error checking. IRRELEVANT
```

[1]'Detours: Binary Interception of Win32 Functions': `https://www.microsoft.com/en-us/research/project/detours/`

```
 }
   //Call real HttpSendRequestW API
   BOOL result = Real_HttpSendRequestW(hRequest,
                                       lpszHeaders,
                                       dwHeadersLength,
                                       lpOptional,
                                       dwOptionalLength);
 if (attached) {
  ...                                //Detour Transaction Preamble. IRRELEVANT
  DETACH(HeapFree);
  if (DetourTransactionCommit() != 0) {
   attached=TRUE;
   ...                            //Commit transaction and error checking. IRRELEVANT
  } else {
   attached=FALSE;
  }
 }

 return result;
}
```

Listing 2: *HTTPSendRequestW Hook*

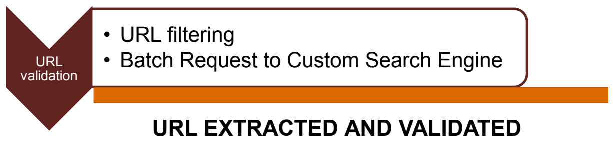
The relevant memory function hooks are represented mainly by the HeapFree function (which deallocates the memory area) since our goal is to copy the full block of memory only after any allocation or modification to it. Arguably, the most significant exploitable memory function for this purpose is the CopyMemory, but we discovered trough reverse engineering of several samples that very often this function is statically reimplemented by each malware as an evasion measure. The hook to the HeapFree, instead, does pretty much the same task but unfortunately it is only able to see the last modification to the memory area analyzed. The memory hook itself is responsible of copying all the content of the memory in a temporal memory space used later for dumping, and gets executed any time the malware deallocates space from memory during the HTTPSendRequestW. Below is an excerpt of such function hook.

```
//Hook to the HeapFree function
BOOL __stdcall Mine_HeapFree(HANDLE hHeap, DWORD  dwFlags, LPVOID lpMem)
{
 if (lpMem!=NULL) {
  SIZE_T blockSize = HeapSize(hHeap, dwFlags, lpMem); //Get size of memory area
  urlMemSize+=blockSize;                              //Control variable

  if (urlMemSize>MAX_HEAP_BLOCK_SIZE) {
   ...      //control checks and dump directly to file for large memory block. IRRELEVANT
  }

  CopyMemory(urlMemPointer, lpMem, blockSize);        //Intercept all the memory region
     pointed by the memory function
  urlMemPointer= (LPVOID) ((LPBYTE) urlMemPointer
                              + blockSize);           //Control variable to write result
 }
 //Call real Heap Free
 return rv = Real_HeapFree(hHeap, dwFlags, lpMem);
}
```

Listing 3: *HeapFree Hook*

The memory dump module is a simple DLL which gets injected into the browser process several seconds after the browser start. The time interval before the memory module injection is necessary to guarantee that the malware injects as first comer into the browser process, and subscribes itself as hooker in first place. During the implementation we force the call to the send request by visiting a set of 5 websites, from the Internet Explorer browser, right after the memory module gets injected.

4.2 Phase 2: Url extraction and validation


URL validation
• URL filtering
• Batch Request to Custom Search Engine
URL EXTRACTED AND VALIDATED

The second phase of the process is represented by the memory dump analysis. The goal of such phase is to scan through the memory dump generated by phase 1 searching for any URL pattern which could represent a website where the sample performs web-injection. In fact, we expect to find in the memory dump the entire list of URLs targeted by the malware. We prepared a custom regular expression with the goal of inspecting the dump and filtering for any URL-like pattern. In this phase we made the consideration that the URL targets defined in the WebInject configuration file are very rarely full-fledged URL strings pointing exactly to the Internet resource. More often, the URL targets are wildcarded expressions, which extensively use the character '*', representing an indefinite number of possible target resources. This fact gives a lot of freedom to the attacker which is not only able to perform injection on multiple pages at once, but also guarantees the possibility of injection also on dynamically generated and/or customized URLs which accept input parameters. It also partially prevents the attacker from rewriting the configuration file any time a web-page gets relocated, a folder renamed or moved on the targeted web-server. Provided such considerations, we looked through the memory dump using the following custom regular expression:

```
((http[s*]?://|*)[*\.0-9a-zA-Z-#]*\. [0-9a-zA-Z-\._~:/?#[]@!$&'()*+,;=%]*)
((\.[a-zA-Z]*)   | * | /)
```

Listing 4: *URL pattern regular expression*

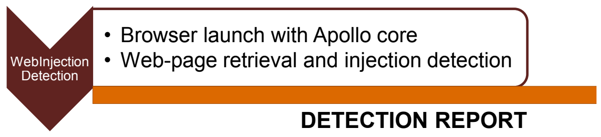
The regular expression built this way retrieves any string which either start with an *http* statement or a '*', and it looks for any string with URL valid characters ending with a set of characters (representing an extension), a '/' or a '*'.

The list extracted this way represents a set of URL patterns which very often do not point to a specific single Internet resource. In order to detect web-injection with the chosen approach we need a definite set of URL addresses to visit and, for this reason, we have to validate and cast the URL patterns into complete Internet Resource Locators. For the validation we used the Google

Search Engine, specifying the parameter 'allinurl' which only matches the queried element against the searched website URLs. From the results of the query we then extract only the top four searched websites. The validation process is quite effective most of the time, since it actually returns in average a single result when starting from a well-defined URL pattern. On the contrary, a really loose expression might return multiple websites of very different nature, but this very rarely happens in real case as the attacker usually targets specific websites, and the related expression is quite well-formed.

The filtering process is performed within the main Python script which structures the analysis phases and controls the browser execution. The URL filtering is implemented by analyzing the memory dump of the browser directly within the Python script, while the validation process uses the Google CustomSearch API Client Library for Python[2], which sends a batch request to the Google servers, containing the set of URL patterns to validate.

4.3 Phase 3: Web-injection detection with Apollo core engine


WebInjection Detection
• Browser launch with Apollo core
• Web-page retrieval and injection detection
DETECTION REPORT

In order to detect a web-injection, as already explained in Chapter 3, it is necessary to compare the injected page source with its original or clean version. To accomplish this, we developed a custom build of the Zeus malware, representing the Apollo core engine. The purpose of the custom build is to attach itself to the IE browser upon its launching before any other process and, while hooking all the WinINet functions, intercept the DOM beforehand, at the root to any other subsequent modification. This goal is achieved by Zeus, representing one of its main functional characteristics. For this reason the core engine was developed starting from Zeus source code and it attaches itself to the browser process using 4 different methods [20]:

1. CoreInject_injectToAll function
2. CoreHook_hookerNtCreateUserProcess function hook
3. CoreHook_hookerNtCreateThread function hook
4. CoreHook_hookerLdrLoadDll function hook

[2] 'CustomSearch API Client Library for Python': `https://developers.google.com/api-client-library/python/apis/customsearch/v1`

The *CoreInject_injectToAll* function (1) simply runs on malware start-up and performs a widespread injection to all the running processes including winlogon.exe (or explorer.exe if the user does not own Administrator privileges [6]) which is the responsible for launching any other user created process. The other three methods are used to inject into every newly created process which evades the mass injection provided by method (1), and they are all similar alternatives operating at different levels. The basic idea consists in hooking the functions responsible for instantiating a new process, and loading, by default, the malicious DLL prior to all the libraries requested by the process. Indeed, every new user process gets created by explorer.exe or one of its descendants, which, in turns, is injected with Apollo core module during the mass injection. The functions hooked are the *NtCreateUserProcess* (2),*NtCreateThread* (3) and *LdrLoadDll* (4), which are undocumented low-level functions exported from the ntdll.dll library, used and wrapped by some more familiar Win32 API functions, which handle the creation of a new process and they are respectively the *CreateProcess*, *CreateThread* and *LoadLibrary* from the Kernel32 library. In this context, the purpose of the core engine is to get loaded into the browser space, prior to any other dynamic library, including the malicious module of the malware sample being analyzed. In this way, Apollo hooks as first comer the WinINet library functions of the browser, such that it has full access to the original page source through its HttpGrabber and WebInject module. The Apollo core build differs from Zeus parent as it just re-implements the features used for web-injection and it dumps the original visited page source in a virtual folder every time a website is reached. In addition, the Apollo core version has been deprived of few operational signatures and regular checks performed upon installation which prevented Zeus from being installed on top of another malware sample of the same family.

For the experiment, the Apollo core runs fully on local with the C&C endpoint pointing to *localhost*, and the local server configured with XAMPP. In the server folder it is possible to access the sample executable, the malware builder and the configuration builder together with all the configuration files needed. It is also present the *webinject.txt* file describing the DOM modifications, necessary for Phase 4. In order to extract the infected version of the website page source, a small portion of the Apollo WebInject module has been imported into a simple DLL (named *http32.dll*) equipped only with the necessary hooks to the WinINet library functions, and tweaked to dump the infected source to the browser virtual folder located in the "Temporary Internet Files" of the current user ("%USERPROFILE%/AppData/Microsoft/Windows/Temporary Internet Files/Low").

In the execution time-line, as in Figure 13, Apollo is launched only once a list of feasible URLs to inspect is available, while the analyzed malware is running in background. It is launched after the malicious sample in order to guarantee its first place as hooker. Some time after the Apollo core engine execution, the IE browser is launched and waits few seconds for all the components to load. Once the browser is ready and both the Apollo WebInject module and the malware sample module are supposedly loaded into the browser process,

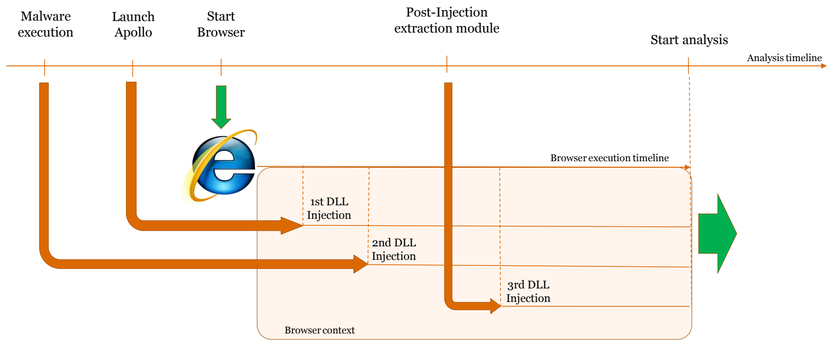
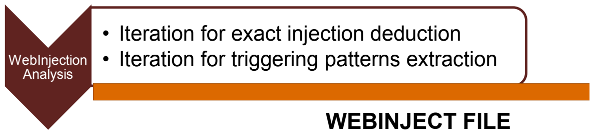


Figure 13: *Browser Injection process. The browser starts instrumented with the Apollo's core engine DLL and the malicious WebInject module before any other dynamic library is loaded. After start-up the post-injection module is attached in order to retrieve the infected DOM*

also the *http*32*.dll* gets injected in order to extract the infected version of webpages. At this point, the browser visits each URL in the list one at a time, and extract both the original page source and the post-injection source. The python script retrieves the sources from the dump folder and compares the two files potentially raising an injection detection. For performance reasons, instead of a char-by-char comparison between the two sources, the files extracted are compared through their md5-based hash representation. Any raised detection alarm passes the analyzed URL to Phase 4, which implements the injection analysis and the extraction of the signature.

4.4 Phase 4: Injection analysis, elicitation and triggering pattern deduction


WebInjection Analysis
• Iteration for exact injection deduction
• Iteration for triggering patterns extraction
WEBINJECT FILE

Whenever an injected page has been detected, the goal of the experiment is to deduct for each injection three elements in the page source representing as close as possible the fields of the configuration file on the C&C:

1. *data_before*: element in the source triggering the injection;
2. *data_inject*: injected field in the page source;
3. *data_after*: field in the source triggering the eventual deletion of unnecessary elements.

It is important to underline that, from the simple source comparison in Phase 3, it is not possible to derive a single solution to all these three elements of the problem. For each injection, multiple solutions exist both for the field *data_before*, *data_after* and also for the *data_inject* field. Indeed, from the hacker's point of view, it is possible to achieve the same injection using definitely different configurations as illustrated in the example below:

```html
<div>
    <h1>Example Page</h1>
    <p>This is the original page</p>
</div>
```

Listing 5: *Original Page Source*

```html
<div>
    <h1>Example Page</h1>
    <p>This is the original page</p>
    <p>I am the injection</p>
</div>
```

Listing 6: *Injected Page Source*

Solution A	**Solution B**
data_before: original page</p>	**data_before:** Example Page
data_inject: <p>I am the injection</p>	**data_inject:** </h1> <p>This is the original page</p> <p>I am the injection</p>
data_after:	**data_after:** </div>

To tell apart between different solutions and extract the exact injection field it is necessary to elicit the malicious behavior by exposing the malware to modified versions of the same page and deduce the injection field or the triggering patterns from the resulting page. In particular, the procedure is split in 3 logical steps: the first determines the exact *data_inject* string (abbr. $di), the second determines the *data_before* sequence (abbr. $db) and the last derives the *data_after* (abbr. $da).

The first step determines the extreme characters of the string $\$di$, also representing the last character of $\$db$ and the first of $\$da$. Given that, the injection location is fully determined by the last character of $\$db$, by removing such character the injection would disappear or simply appear in a different location from what expected. Thus, from the original page source, the algorithm removes one char at a time backward, starting from the original position of the injection detected and refresh the page until the injection disappears or gets relocated. Below we propose the pseudo-code of the procedure:

Algorithm 1 Extract injection beginning

```
Result: Find the exact start position of the injection
originalInjectionDetected = (string) injection Detected in Phase 3
db_end = (char) elem Of Source @(originalInjectionDetection[0] - 1);
while True do
    remove element db_end from source;
    refresh page;
    if isPresent(inj) and inj==originalInjectionDetected then
        db_end = (char) elem Of Source @(pos(db_end) - 1);
    else
        break;
    end
end
di_start = (char) elem Of Source @(pos(db_end) + 1);
```

In a similar but opposite fashion also the end of the injection is determined by evaluating the first character of $\$da$ if present, proceeding forward and starting from the end of the injection originally detected.

Step 2 and 3 follow a similar procedure and aim to determine the exact $\$db$ and $\$da$ patterns which trigger the injection. Below is the pseudo-code for retrieving the $\$db$ field.

Algorithm 2 Derive exact *data_before* field

```
Result: Find the exact start position of data_before field
exactInjectionDetected = (string) injection Detected in Step 1
db_end = (char) elem Of Source @(exactInjectionDetected[0] - 1);
db_start = (char) elem Of Source @(exactInjectionDetected[0] - 2);
source_to_remove=pageSource[0:pos(db_start)];
while True do
    remove element source_to_remove from source;
    refresh page;
    if notPresent(inj) or inj != exactInjectionDetected then
        db_start = (char) elem Of Source @(pos(db_start)-1);
        source_to_remove=pageSource[0:pos(db_start)];

    else
        break;
    end
end
db_start = (char) elem Of Source @(pos(db_start) + 1);
db=pageSource[pos(db_start):pos(db_end)];
```

The current algorithm adopts the same approach of Step 1 in an inverse manner. The page, from beginning to the detected injection position, is fully deleted from the source such that the injection necessarily disappears or gets relocated, through the deletion of the original $*db* field. The removed region is then progressively shrunk starting from the injection position and the page refreshed until the injection appears back where expected. At this point the characters left between the removed region and the injected string represent the $*db* field. Since the $*db* field could represent a very long string this would imply a lot of page refreshes. Thus, for performance reasons, the real algorithm uses a dynamic delta to compute the number of characters to remove at each refresh, which progressively either duplicates or halves down if the injection has already been observed once or not. In a very similar but opposite fashion also the full $*da* field is derived and the full injection signature extracted.

For the experiment, the entire analysis procedure is automated within the main Python script implementing a basic web-crawler with Selenium 2.0 WebDriver API for Python. Selenium is a browser automation library, often used for any task that requires automating interaction with the browser it supports[3]. Additionally, in order to make the necessary modifications to the page source to extract the triggering patterns, we exploited the Apollo core build which is capable of returning the original version of the page and to directly modify it prior to the malware sample injection. Apollo core has also been properly setup to provide a timely response to the configuration changes. Modern browsers makes extensive use of page caching to improve network performances, which

[3] 'Selenium HQ Browser Automation': http://www.seleniumhq.org/

would mine the working principle of the current experiment. For this reason all the Internet Explorer caching options has been disabled. Under these circumstances, we are able to make a full page refresh every 2-3 seconds and for each web-injection an average of 20-30 refreshes are necessary to extract the full injection signature, resulting in an operational time of around 2 minutes per each web-injection.

4.5 Automated analysis and anti-evasion

The whole experiment has been wrapped into a unique solution in a virtualized environment with a 32 bit version of the Windows 7 operating system running on the free and open-source hypervisor VirtualBox by Oracle. In order to automate the procedure we used Cuckoo Sandbox, which is a malware analysis system capable of performing automated dynamic analysis of provided Windows binaries[4]. It also returns comprehensive reports on key API calls, network activity and full memory dumps. In order to launch our analysis tool we patched the agent file (*agent.py*) provided by Cuckoo with the goal of launching Apollo after the execution of the binary inside the VM. To avoid any early interruption of the analysis we enforce a timeout of over an hour in the Sandbox. To counteract some evasion mechanisms, generally adopted by malware binaries, we adopted few countermeasures. Our virtual machine was firstly hardened with a batch script which deletes the traces left by VirtualBox during the VM installation. The script modifies some VM configurations and Windows register entries in order to prevent the fingerprinting by the analyzed sample of the virtualized environment. Among these, it modifies the names of the graphic adapter, which by default is *VBoxGraphicAdapter*, and other virtualized device drivers names starting with "VBox". As a common practice, Apollo was configured to start with a delay of nearly half an hour after the binary execution. This is, as well, an anti-evasion practice because many malicious binaries usually undergo a long initial activation period generally meant to look less suspicious to the unfortunate user and also to evade malware analysis.

After the experiments, the analysis results are written in the form of text files and sent, over the LAN network, to the host machine, which receives them through a mini-server script. The results collected in the *data* folder of the log server contains respectively:

- A log file listing all the websites analyzed by Apollo indicating for each one if any injection were detected.
- (if present) Two url files listing all the url extracted from Apollo, both prior and after validation
- (if present) A *webinjects.txt* file representing the solution for the configuration file computed by Apollo

[4] 'Cuckoo Sandbox: Automated Malware Analysis': `https://cuckoosandbox.org/`

5 Experimental Validation

In the current chapter we expose the results from the malware analysis collected in different experiment scenarios.

The first is a custom made experiment used as proof of concept for the elicitation approach and it is based on a custom binary properly configured to perform web-injection. The second scenario extends the first experiment, but embraces a rather different objective, and aims at testing the overall functionality of Apollo. In the third experiment we step into the validation of the results by testing the tool in total absence of malicious activity and demonstrating the low false positives detection ratio. In the last experiment we analyze a set of real samples downloaded from VirusTotal to test the performance of Apollo under real circumstances.

5.1 Eliciting triggering expressions on a custom sample

5.1.1 Goal

The experiment was prepared in order to test the proper working and functionality of Apollo. In particular, this experiment aims at proving the validity of the eliciting method as a mean to deduce the triggering patterns and the injected strings in a web-page, which is consolidated target of web-injection. For the first experiment we want to show that by providing the tool with a targeted URL it is possible to detect the web-injection and derive its signature in terms of $data\_before$, $data\_inject$ and $data\_after$ fields.

5.1.2 Experimental Setup

The first experiment was launched using a Zeus bot. An Ubuntu 12.04 LTS VM was set to host the C&C server for the sample and the network settings adjusted in bridged mode. The server was prepared to host the Zeus control panel, which is a nice web-interface that allows to remotely control all the active bots of the given botnet. It was installed the latest versions of Apache, MySQL and PHP, then the Zeus cp (control panel) through the user manual [13]. The Zeus malicious binary was built using the default builder with the following static configurations:

```
entry "StaticConfig"
  ;botnet "btn1"
  timer_config 1 1
  timer_logs 1 1
  timer_stats 1 1
  url_config "http://192.168.2.101/zeus/config.bin"
  remove_certs 1
  disable_tcpserver 0
  encryption_key "123456"
end

entry "DynamicConfig"
  url_loader "http://192.168.2.101/zeus/bot.exe"
  url_server "http://192.168.2.101/gate.php"
  file_webinjects "webinjects.txt"
  entry "AdvancedConfigs"
    ...     <!--irrelevant-->
    ...
end
```

Listing 7: *Static Configuration File*

The configuration binary was built using the same Zeus builder and we specified the injection list and web-inject behaviour through the *webinjects.txt* file as follows:

```
set_url *google.it/?gfe_rd=cr* GP
data_before
<body *>
data_end
data_inject
<h1>Hi! I am Zeus</h1>
data_end
data_after
data_end
```

Listing 8: *WebInject file used in first experiment*

To be noticed is the fact that in the *webinjects.txt* file it is possible to define regexp, using wildcards like ’*’, also for triggering patterns like $data\_before$, as in the experiment. Detection of wild-carded expressions falls outside our purposes, while it does not introduce any limitation for the experiment. The sample binary together with the configuration binary file was located in the apposite server folder.

On the client, a Windows7 32-bit virtual machine, Apollo core engine was configured to run in local on top of the XAMPP installation. The client is set up with two network adapters, a host-only adapter and a NAT adapter in order to allow the communication between the two virtual machines (the C&C server and the client), the host operating system and the Internet. The file properties of Apollo are set to point to all the needed files located on the machine like the core installation folder and the folder of the browser memory dumps. The Internet Explorer cache option is disabled and all the security levels set to the minimum. We launched the first experiment by specifying the URLs to visit, precisely `https://www.google.it/?gfe_rd=cr`.

5.1.3 Results

We first launched the Zeus bot and we observed the injected page by visiting the URL as in Figure 14.

We launched Apollo and waited for the results. It correctly detected the injection in the web-page and started the pattern analysis, which returned an inject file very similar to the original configuration file.

```
set_url https://www.google.it/?gfe\_rd=cr G
data_before
<body bgcolor="#fff">
data_end
data_inject
<h1>Hi! I am Zeus</h1>
data_end
```

```
data_after
data_end
```

Listing 9: *WebInject file result of first experiment*

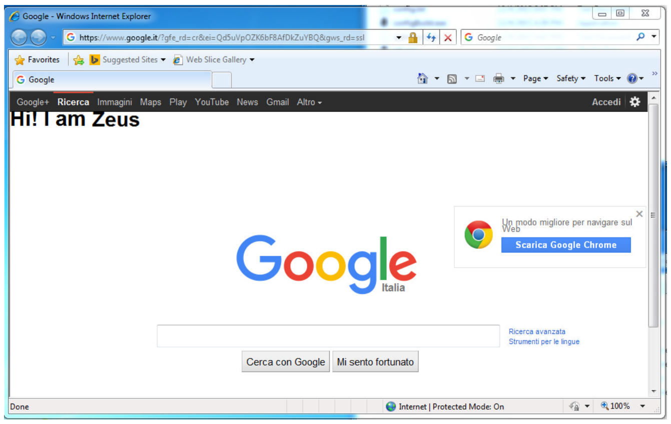

Google - Windows Internet Explorer
https://www.google.it/?gfe_rd=cr&ei=Qd5uVpOZK6bF8AfDkZuYBQ&gws_rd=ssl
Favorites
Suggested Sites
Web Slice Gallery
Page
Safety
Tools
Google+ Ricerca Immagini Maps Play YouTube News Gmail Altro
Accedi
Hi! I am Zeus
Un modo migliore per navigare sul Web
Scarica Google Chrome
Google Italia
Ricerca avanzata
Strumenti per le lingue
Cerca con Google
Mi sento fortunato
Done
Internet | Protected Mode: On
100%

Figure 14: *WebInjection detected on Google page*

The only noticeable difference from the real *webinjects.txt* located on the server is the wildcard '*' used in the *data_before* field, replaced by the corresponding set of characters in the visited page source. The presence of regexp is not a relevant aspect for the triggering patterns extraction. In fact, we deal in practice with real page sources, and in order to extract a working signature for a given malware sample, it is only necessary to determine where an injection is observed on that given page. Anyway, with the eliciting method, it is theoretically possible to detect the presence of wildcards using a more fine-grained logic which iterates on single character removal of the page source rather than on block of characters. In practice, this drastically reduces performances by increasing exponentially the necessary iterations used for pattern extraction, and for this reason was considered out of scope.

The current injection was extracted using 16 refresh iterations of the webpage with a net operating time of around 80 seconds. According to the implemented logic, this is estimated to be an average result for injections that have only a simple *data_before* field and no *data_after*.

5.2 Signatures and URL extraction on a custom sample

5.2.1 Goal

The second custom experiment aims at verifying the correct functionality of Apollo, focusing, in particular, on the URL extraction problem. In this experiment we want to show that it is possible, using the proposed approach, to extract a list of URLs, targeted by the malware, from a closed-source binary, and which "modus operandi" is not exactly known a-priori.

5.2.2 Experimental Setup

For the second experiment, we used nearly the same configuration of the first experiment, but instead of a Zeus build, we deployed a leaked version of the Citadel builder. As a descendant of Zeus, Citadel exposes nearly the same interface, builder and configurations. We used the same C&C server of Zeus experiment, and we additionally installed the Citadel control panel. As in the first experiment, we placed on the apposite server folder the sample and the configuration binary generated with the Citadel builder. For this experiment, while the static configurations were nearly the same, we used the *webinjects.txt* file provided within the builder kit. In the file, containing by default 115 web-injections, we added a few injecting blocks and also updated some of those already present blocks since many of them were pointing to old web-pages and were not working any longer because of obsolete triggering patterns.

Finally, we launched the experiment directly by submitting the sample via Cuckoo Sandbox[5] in order to test the overall functionality of Apollo.

5.2.3 Results

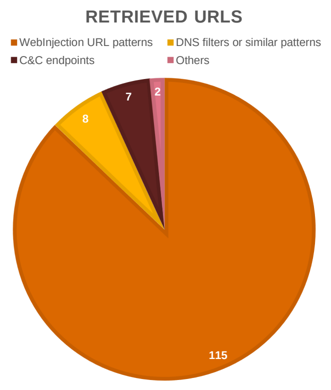
Since no URL was specified, Apollo called the URL extractor upon launching, which performed a dynamic memory dump through the hook to the memory functions. On browser launch, after all the DLL module injections, a set of web-sites are visited in order to enforce the allocation and de-allocation on memory of the targeted URL list and the memory dump. The analysis tool found traces of 132 URLs in memory and correctly retrieved all the addresses defined in the *webinjects.txt* file, plus few other URLs that we identified as relevant to the malware, representing some of the C&C endpoints or the DNS filters used by the sample as in the list below:

```
http://192.168.1.101/citadel/server/files/webinjects/injects.txt
http://*.com/*.jpg
*facebook.com/*
http://192.168.1.101/citadel/server/gate.php
*payment.com/*
https://www.wellsfargo.com/
https://www.us.hsbc.com/*
...
```

Listing 10: *Extracted Url list of second experiment*

The subsequent validation of the extracted URLs returned 137 elements. In the extracted set, around 25 elements returned no results as these were simply obsolete URLs or not indexed by the search engine (as the C&C endpoints), while some other strongly wild-carded expressions returned more than a single result. After validation, the analysis continued to detect web-injections in the set of validated URL addresses. Apollo correctly identified and reconstructed

[5] 'Cuckoo Sandbox Automated Malware Analysis': `https://cuckoosandbox.org/`



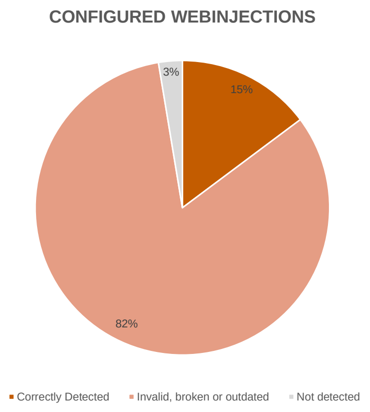


Figure 15: *URL extraction (left) and web-injection detection (right) relative to known configuration file*

17 web-injections together with their triggering patterns. All in all, we detected a small number of web-injections compared to the number of retrieved URLs, but after a manual verification we estimated that the tool was identifying nearly all the active and working injection, while the rest of the validated URLs were not malware targets or simply represented old signatures no longer working. In the set, we counted only 3 supposedly active web-injections that Apollo did not detect, and this was due to either an uncaught URL instance during validation or the page unresponsiveness during the detection analysis.

The experiment produced as outcome a *webinjects.txt* file which differs sensibly from the original only in a structural form, while it is very similar on a conceptual basis as in the excerpt below:

```
set_url https://www.us.hsbc.com/* GL
data_before
<table id="fiveBoxDisclosureHygiene" *>
data_end
data_inject
data_end
data_after
</table>
data_end

...


...
```

Listing 11: *Portion of original webinjects.txt*

```
set_url https://www.us.hsbc.com/1/2/home/
     personal-banking G
data_before
<table id="fiveBoxDisclosureHygiene" summary
     ="Investments,␣Annuity␣and␣Insurance␣
     Products␣Disclosure" style="font-weight
     :bold;">
data_end
data_inject
data_end
data_after
</table>
data_end
...
```

Listing 12: *Portion of generated webinjects.txt*

Also in this case, we detected that the main conceptual difference is determined by the presence of wildcards, both in triggering expressions and in the URL targets. This difference is irrelevant to our goals and it does not embody any significant limitation. In addition, in case of URL targets, it is still possible to easily identify the exact URL pattern by recovering the base matching regexp from the list of raw URLs extracted before validation.

In the current experiment, for each page where an injection was detected, Apollo took in average 120 seconds to retrieve the triggering patterns with about 25-28 refresh iterations per page. The number of refresh needed is proportional to the length of the triggering expressions. The algorithm, uses an incremental exponential algorithm which basically flattens out the requirements for page refreshes for very long patterns. Indeed, in the tested set we measured a maximum number of 35 refreshes in case of long expressions against a minimum of 19 iterations for very short patterns, with an exact average of 26,4. The results collected from this experiment can be represented by the charts in Figure 15, while below we provide an approximated table of Truth for both the URL extraction and the web-injection detection problem for the current experiment.

URL extraction Truth		
.	True	False
Positive	115	22
Negative	0	-

WebInjection detection Truth		
.	True	False
Positive	17	0
Negative	3	95

5.3 False positives detection

In the third experiment we validate Apollo against a case where no malicious activity is present, aiming to demonstrate the low false positives detection ratio.

5.3.1 Goal

The experiment has the purpose of testing Apollo against false positives and negatives both for the URL extraction method, and for the web-injection detection. The goal for such task is to help in quantifying the impact and frequency of a wrong detection (false) of the tool. The first mini-experiment aims at quantifying the frequency of false positives during the analysis of the memory dump and URL retrieval. This experiment also aims at defining a good set of web-pages to visit in the preamble of the analysis used to enforce the memory dump during the send request, with the scope of mitigating the presence of false positives. The second mini-experiment also aims at determining the bearing of false positives of the web-injection detection procedure which runs only after a URL list set has been extracted.

5.3.2 Dataset

In order to test the URL extraction module we launched Apollo 25 times with no malicious samples running in background. The analysis, by default, visits a set of 5 predefined web-pages in order to trigger the send request function. On the experiment set, we analyzed the file dumped by the browser looking for any URL pattern. Any detection under these circumstances is considered a false positive. For testing the web-injection detection, instead, we identified a list of 100 websites taken from the Alexa Top Sites rank[6] and we fed those to Apollo, which tried to detect any injection on these resources, always with no active malicious payload running.

5.3.3 Results

In the first part of the experiment we detected 1 URL in 25 experiment repetitions, representing a single false positive instance. Very likely, the detected false positive was triggered by some control script performed by one of the visited web-page (`google.com`). In this experiment we do not have a base ground truth, i.e. a URL list to test against, thus, when referring to false negatives, we instead refer to the analysis test, which did not report detection. From the result, we considered the list of websites rather acceptable for the analysis on unknown sample.

URL false detection test	
.	False
Positive	1
Negative	24

On the other side, the test on false web-injection detection, did not reveal any injection, and this is rather straightforward from the implementation point of view as we are comparing two exactly equal strings dumped at a nearly null time difference, with no interceptor on the way.

WebInjection false detection	
.	False
Positive	0
Negative	100

5.4 Analysis results on real samples

In the last experiment we analyzed a set of real samples to test the performance of Apollo under real circumstances. The experiment revealed several difficulties,

[6] 'The top 500 sites on the web': `http://www.alexa.com/topsites`

while testing on real unknown samples, and this is very likely to be attributed to several factors that we examine. Among these, the biggest problem we encountered was the total absence of a dataset with a known ground truth usable for validation. Instead, we only had the chance to test on a set of samples of completely unknown nature. This fact has reasonably impacted on the outcome.

5.4.1 Goal

The goal of the experiment is to prove the validity and functionality of Apollo in a real case scenario where an unknown sample is submitted for analysis. For the experiment, we provide a dataset of malware samples which are supposedly performing web-injection, and our purpose is to detect and identify any relevant injection or suspicious activity.

5.4.2 Dataset

The analysis has been launched on a set of 200 binaries downloaded from VirusTotal[7], a free virus and malware online scanning service, which also offers a huge repository of samples. Each binary is hashed and tagged with annotations provided by several different anti-virus software companies. For our experiment, we limited the malware research focusing on Web-Inject based families defined using specific annotations. We used as reference for the search the tag provided by microsoft, considered rather trustworthy in matter of banking trojans. We derived the annotations used for the search filter, by submitting few known samples to VirusTotal. Then, we manually verified the tag results and used them to perform further queries. We also validated the retrieved annotations through the Microsoft Protection website[8], which provides an exhaustive description of the family for each tag. During this kind of research we observed similar annotations belonging to banking trojan families. We added the most sensitive tags, representing modern malware binaries performing web-injection. Our filter was hence composed by sample tagged with the following annotations:

PWS:Win32/Dyzap	→ Dyre family
PWS:Win32/Zbot	→ Zeus family and descendants
Win32/Banker	→ Generic Information-Stealing Trojan
Win32/Drixed	→ Dridex family
Win32/Tinba	→ Tinba Trojan
Win32/Vawtrak	→ Vawtrak Trojan
Win32/Injector	→ Generic Browser Injector or keylogger

Finally, the dataset was comprehensive of the most recent samples submitted on VirusTotal, divided quite evenly among the malware families represented by the tags used in the search filter.

[7] 'VirusTotal': `https://www.virustotal.com/`

[8] 'Malware Protection Center': `https://www.microsoft.com/security/portal/mmpc/default.aspx`

5.4.3 Results

The validation performed on a real dataset returned little, yet significant results. Out of 200 tested samples, we extracted a set of URLs only from 10 samples, with an average of 23 URLs per sample. Of the 227 URLs extracted, many did not survive validation and on the remaining we did not observe any web-injection. This represents an unfortunate, but not completely unexpected outcome, and the results led us to investigate deeper on the causes of the scarce findings.

Results Data	
Analyzed samples	200
Active samples detected	10
URLs extracted	227
URLs extracted per active sample (avg.)	23
URLs validated	48
URLs validated per active sample (avg.)	5
Detected WebInjection	0

Performing analysis on malware is a particularly challenging task. Precisely, testing on real samples of completely unknown nature is very difficult due to the absence of a verified ground truth, which is essential for testing.

First of all it is essential to state that our approach only applies to active malware, and, to be considered as such, a sample needs to dialog with a running C&C server, needs to exploit the WebInject module and to own an up-to-date WebInject configuration. As a public and freely accessible service, VirusTotal does not always host active samples. Quite often, instead, as soon as a sample is submitted for analysis on VirusTotal, the C&C endpoint of the malware gets rapidly shut down by the same cybercriminals to evade controls and leave no traces of their criminal activity. In this context, VirusTotal is considered a good source of samples, but not really a source of good samples. The research itself on VirusTotal is not easy since, often, annotations provided by different anti-virus show mismatching results. Malware code can hardly be attributed to a single macro-family by a simple automated static analysis. Indeed, malware does not disclose its behaviour until an observation is made, and, even if samples exhibit a certain functionality, as WebInject, it is a-priori uncertain whether that piece of functionality is actually active, not exploited, dormant, or the server has been shut down.

In addition to such considerations, nowadays, a large number of malicious samples is also capable of detecting sandboxes and virtualization environment and these numbers have drastically increased since 2014 of nearly 2000% meaning that at least 17% of malware samples adopt evasion techniques to detect analysis environment [21]. On the other side, applying strong countermeasures is not straightforward and requires independent researches. Additionally, the general downtrend in banking trojan adoption observed from 2014, gives quite well the idea about the complexity of an effective (and active) samples research.

Despite these considerations, we tried to quantify the extent of the problem by manually analyzing a subset of the given dataset.

To determine whether a sample is communicating with the C&C server we analyzed a subset of 20 samples verifying, using Cuckoo's sandbox, the VM's network connections performed during the analysis. In case of an active infection we expect to find at least a request to a remote suspicious Internet resource. In fact, we verified that, out of 20 samples, only 7 were actually performing at least an unexpected network request, representing the 35% of the total.

Beside general considerations on the samples activity, Apollo exposes certain implementation limitations, which might also be the cause of the sub-performance. The method used to perform web-injection detection is strongly interconnected and dependent on the URL extraction problem, and comes in only after this in the pipeline. Our approach to the URL extraction problem is based on the hook of WIN32 APIs (in particular HeapFree), and implements a schema of action which is rather generic but which also makes few important assumptions, which are:

- The sample imports the HeapFree function in the executable.
- The sample performs a dynamic allocation and matching of the target URLs during a resource request (as in Zeus and descendants).

To quantiy this fact, we reversed through decompilation a subset of 20 samples in order to verify whether the WIN32 APIs, in particular the HeapFree function, were dynamically (or statically) imported or the malware simply used other similar libraries. We found evidence that only 9 samples out of 20 were actually importing the wanted memory functions representing the 45% of the total.

Eventually, by extending the gathered statistics to the whole dataset, we can assume that, out of 200 samples, we were effectively testing Apollo against

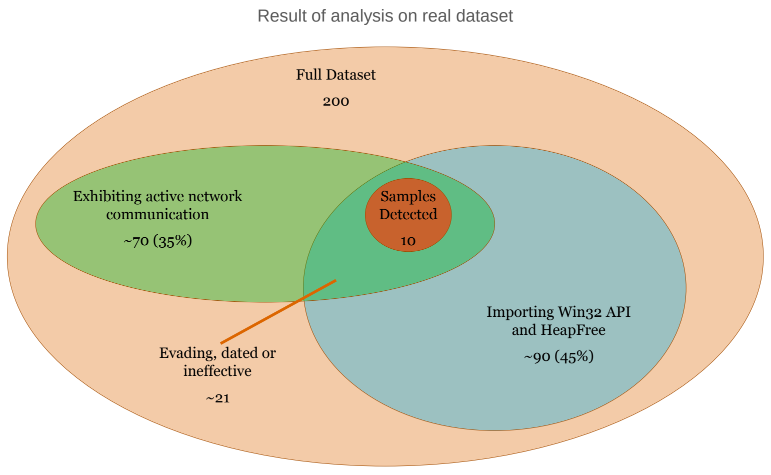

Result of analysis on real dataset
Full Dataset
200
Exhibiting active network communication
~70 (35%)
Samples Detected
10
Importing Win32 API and HeapFree
~90 (45%)
Evading, dated or ineffective
~21

Figure 16: *Analysis result on collected dataset and samples statistics*

only about 30 active and valid samples, as depicted in Figure 16. Among these, chances exist for the samples to be dated, not configured for web-injection or they might also adopt strong evasion mechanisms. In this situation, our results gain in significance and provide a substantial hint in the definition of guidelines for further improvement.

In addition, we deepened our analysis on the few detected samples in order to collect as many information as possible on the working set. We verified that only one element in the set was represented by a false positive result. Indeed, we relaunched multiple times the analysis on the subset returning always similar detections on 9 samples out of 10. By verifying the annotations of each of those samples on VirusTotal, we observed that 6 out of the 9 detected samples were belonging to the Vawtrak family, as shown in Figure 17. This could mean both a remarkable recent activity of the Vawtrak malware and a particular sensitiveness of that family to the analysis tool.

By manually inspecting the set of extracted URLs we also noticed that many URLs were closely related to the set of web-pages which Apollo visits at the preamble of the analysis in order to enforce the memory dump. We speculate that such set of URLs is the result of the malware activity inside the browser and represents an attempted comparison of the currently visited resource against a target list. In this context, we recognized that Apollo did not extract the WebInject targets, yet it recorded an important part of the malicious activity inside the browser. We hypothesize that the WebInject targets might have passed undetected due to a shutdown C&C server or improper WebInject configurations, which keep the trojan active inside the browser, but return no target elements to the bot to compare against for web-injection.

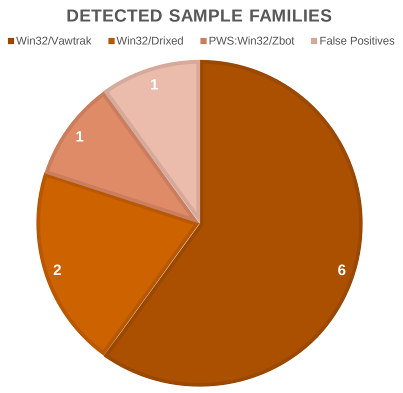


Figure 17: *Distribution among malware families of the samples detected during the analysis*

To validate our hypothesis we also analyzed the detected samples using Prometheus [4], which computes DOM differences by comparing web-pages from two different machines, a clean and an infected one. Also from the analysis with Prometheus we observed no significant web-injections performed by this subset. Instead, we only detected DOM differences that we confirmed, at a good confidence level, being of false positive nature.

To sum up, the malware research and the implementation limitations make the task of testing on real samples extremely difficult. This is particularly critical in the circumstance when neither a good dataset not its corresponding ground truth is available. Despite this, we verified that Apollo is able to detect malicious activity inside the browser even though the analyzed malware does not perform web-injection.

6 Discussion of Limitations

In this chapter, we discuss the limitations and points of weaknesses of the project. It is important to highlight that the points of discussion below derive mostly from experimental observations and critical reasoning while testing Apollo in phase of development. The tool has proven effective on custom tests, and, since gathered data on real samples did not show very generous results, we were able to draw a few conclusions about the operational limitations of Apollo.

6.1 WebInject targets extraction

The biggest limitation for the current project is the insufficient testing due to the absence of real known working samples and an established ground truth to the problem. The main problem derives from the difficulty of extracting a valid URL set where the sample might perform web-injection. This is the most puzzled problem since, there exist only few and never generic solutions.

In a situation where a plausible set of targeted URLs is totally absent, the only chance left for testing Apollo is to blindly analyze a given predetermined set of websites, which could be potentially targeted by banking trojans, that could either be a banking website or any other website with confidential and sensitive data. This methodology, even if potentially effective in terms of final outcome, radically changes the perspectives of the analysis, turning its philosophy in a website-based action-schema rather than a malware-based one. Using such different approach, it would mean, in practice, to shift the focus of the analysis not on the malicious sample, but on a list of certified websites, acting as a security warranty for these. This approach, beside being of a different philosophy, it is also practically unfeasible by a tool performing extensive analysis, which should visit thousands of different websites for every single analyzed sample, but it might be an effective mechanism for punctual investigation on a single sample.

By considering the malware-based approach currently adopted for extracting URLs, it is necessary to make some considerations. The only reason why there exists a method to extract a targeted URL list is because a malware usually stores such list and makes the comparison with the visited URL in clear. This behavior generally derives from the possibility for the attacker of specifying a URL also in a regexp form (e.g. $*.facebook.com/*/payment$). This allows an attacker to lazily define URLs which might accept variable parameters or might change over time on the targeted website after a version update. In fact, in the case this feature is not provided, a really naughty malware sample could easily hash the target list all the way from C&C server to client and make the comparison of such list with an hashed version of the visited URL. This implementation mode would preclude nearly any possibility to the URL extraction problem in absence of the hashing algorithm or encryption key used by the sample. This consideration makes evident the fact that there exists no general solution to such problem, and, any unfolding which might come out is, more or less, dependent on the malware family or the implementation of such feature.

Once acknowledged this fact we can, thus, exploit the principle that the malware needs to have stored somewhere in memory (temporarily or not) the targeted URL list during a page request, in order to make the needed comparison with the visited URLs to perform web-injection. Therefore, the most intuitive method, in this context, for extracting such list becomes a memory trace approach. Our adopted procedure to this problem uses this philosophy in a lightly malware-dependent fashion, but in section 6.3.1 we propose a more compelling solution which was also the most generic one we could think of for the current problem.

6.2 DOM extraction

Beside the main problem of URL extraction, during development phase we examined viable evasion mechanisms which led us to pinpoint some weaknesses of the adopted approach for DOM inspection. This section especially deals with all the issues observed during injection signature extraction.

6.2.1 The API hooking detection evasion

The proposed approach for injection detection relies on the API hooking technique which is an incredibly powerful instrument for myriads of purposes, if used properly. However, this method operates at the same abstraction level of malware code which performs web-injection. Malware, indeed, use the same technique to perform the Man-In-The-Browser attack. This fact not only raises difficulties in the control of the three phase injection work-flow (explained in Chapter 4), but also exposes the analysis tool to the malware operational range, making it potentially detectable by a careful malicious sample. Even though there exist several methods to do API hooking, there also exist many countermeasures that generally allow to detect hooking activity and seldom even to remove it[9]. In addition, even though the Apollo core does not show any relevant stability issue, its needed module is still closely embedded in the malicious components such that it is difficult to guarantee the absence of unwanted features, which might generate conflicts during the installation on top of another unknown malware sample. In face of such considerations, even if this does not constitute a fundamental blocking issue, it is a strongly advised idea, as a future work proposal, to extract from Apollo the module performing web-injection in a completely new solution and apply to it a stronger and less detectable API hooking method. Alternatively, it is possible to apply a different approach to extract the original page sources as for example using a proxy (see section 6.3.2).

[9] 'C++:Detecting and Removing API Hooks using C++': https://www.unknowncheats.me/wiki/C++:Detecting_and_Removing_API_Hooks_using_C++

6.2.2 The inherent ambiguity of text injection and the limit of the eliciting approach

Beside the controlled process injection problem there exist a critical evasion mechanism to the eliciting method, which represents a limitation for the proposed approach. The problem arises from the fact that, for an accurate injection signature extraction, it is necessary to refresh the page multiple times in order to elicit the malicious behavior. This is true for both the injection data and the triggering patterns, and this derives from the fact that injected text in a string is intrinsically ambiguous, i.e. multiple solutions exist for the same result, as shown in section 4.4. In this context, there exists a feature defined by financial trojans which allows to perform web-injection on a given website only once per day (and in particular this is valid for both Zeus and Citadel). Indeed, by relying also on browser cached results, in common situations, it is quite unlikely that a user makes multiple refreshes on the login page of a banking website several times per day. And, even when this happens, the browser (unless differently specified as in the case of the experiments) loads the results directly from the cache which might have already been infected by the malware sample. With such feature enabled, using multiple refreshes to extract a signature becomes a limitation and, by disabling cache options, on every subsequent refresh the web-injection would get lost and the approach falls short.

On the other side, it is also true that this evasion mechanism could be mitigated by extracting a less accurate injection signature "all in one shot". By enhancing the analysis of the page source upon injection detection and relying on the fundamental structures of the source programmatic language, it should be possible to extract with good approximation both the injection and the triggering patterns. For example, for an HTML source, it could be a good practice to make the assumption that the injection and the triggering patterns start and end with a tag element (e.g. `<div>`) as this represents the most practical and intuitive solution for an injection also from the perspective of the cybercriminal. In addition, it is also important to remark that the accurate injection behavior is not a key aspect in the generation of a signature. In the signature genesis, we are mainly interested in the observation of the injection and its observed location on a real web-page from the victim perspective rather than the exact triggering expression provided by the attacker.

To sum up, the feature of "web-injection once a day", that common malicious samples embrace, is the worst enemy to the eliciting approach. However, in this field, we expect fair good approximated results also through the adoption of a different analysis method which would also become a lot faster in signatures retrieval. For these reasons, the present observation, could turn, from a limitation, into a point of strength if implemented with due precautions.

6.3 Future Work

The problem of trojan web-injection extraction represented a complex challenge involving several aspects. In our project we crossed the whole topic in a horizontal manner, approaching the problem from several perspectives. In this situation, we acknowledged all the limitations that our methods expose, but, by learning from our mistakes, we also want to shed light on the track which is recommended in our opinion to follow for future development. In this chapter we rise few proposals for further research, that we will also assume as our next starting points to carry on our mission.

6.3.1 URL extraction through taint analysis and memory tracing

Our main issue throughout the project has been the lack of a ground truth for testing Apollo due to the absence of recognized working samples and due to the difficulties of extracting a reasonable list of URLs where the malware is likely to perform web-injection. For this reason, we give first priority to improve the URL extraction problem, in order to shape the URL extraction module in a more generic and malware-independent solution.

Our idea and proposal for future work is to perform dynamic taint analysis on the URL passed to the browser to request a web-page. Dynamic taint analysis consists in the attempts to identify variables that have been 'tainted' or mobilized with user controllable input and trace all the memory address (i.e. other variables) that have had any kind of relationship with the tainted variable. The tainted memory region progressively spread and the goal of the analysis is to identify all the memory addresses which contains values that have been, anyhow, influenced by the original tainted variable.

In this context, the idea is to use a dynamic binary instrumentation tool, like Intel PIN[10], and track all the memory addresses associated to the URL provided to the browser for a page request, by intercepting any form of string comparison or memory copy function, possibly at the machine instruction level. The approach would be quite similar to the one used in the current project, but, instead of performing a taint to the memory during the API call by hooking the HeapFree function, the interception should be performed on all the useful instructions at machine instruction level, that perform comparison with elements in the tainted memory area.

We assume and we also expect from observations on previous experiments that such comparison usually occurs during the send request API (that in the *WinINet* Library is the *HTTPSendRequestW* function), which establishes the connection with the chosen web-page. With this information it should be possible to confine the memory analysis in a temporally restricted region of time, so to guarantee an important preliminary filter. After the page request has been returned, the whole tainted area (and maybe also adjacent regions) gets dumped. The whole dump is then queried with a regular expression to extract

[10] 'Pin - A Dynamic Binary Instrumentation Tool': `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`

all the possible traces or footprints of URLs. It is important to underline that this approach still has its validity due to the principle that URLs are generally stored in clear in memory as explained in 6.1, or it would add no significance to the project.

6.3.2 Proxy implementation to intercept DOM

The second problem we identified is to correctly determine the process injection order performed by the malicious sample and Apollo, during the browser start-up. We realized that defining a good hooking anti-detection strategy to be invisible to the malware while controlling the process injection requires a not indifferent effort. Given the number of difficulties that we identified in this process (as explained in Chapter 6.2.1), we came up with the first proposal of migrating in an utterly new solution the Apollo core, which is responsible for retrieving the original page source and performing preliminary modification needed for the elicitation paradigm. The idea is to detach the core logic from the abstraction level at which the analyzed malware sample is also operating, and bringing it at a lower level of operation.

In practice, instead of using the user function hooking technique we propose to introduce a proxy implementation between the analyzed VM and the Internet, which acts as a Man-In-The-Middle (MITM), capable of intercepting all the Internet traffic between the two speakers and freely modify its content. This method in principle sounds like a flawless solution to this problem, however, in practice exposes some implementation difficulties. One of these is that nearly all the modern web-servers only accept connections through the HTTP over SSL Protocol (or HTTPS), especially in case the website handles sensitive data like banking websites, principal targets of web-injection. The HTTPS Protocol is, by design, resistant to the man-in-the-middle attack, which means that it

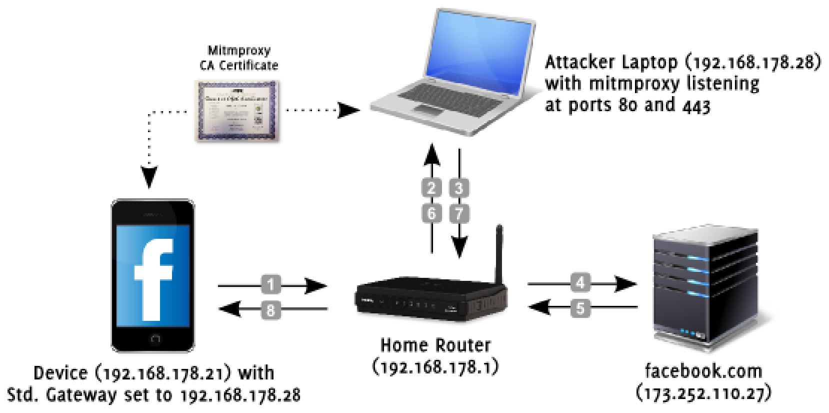


Figure 18: *Schema of Man-In-The-Middle implementation extended to work with SSL Protocol*

is impossible for an intermediary to modify or even listen to the conversation between two speakers (client and server) without letting the client know about the presence of the man in the middle. In such case, the connection is immediately interrupted and the user informed about the risk. To elude this security feature it is possible to setup on the proxy server a certificate authority (CA) which generates SSL certificates to whatever host-name is needed for a connection as illustrated in Figure 18. Provided that the client knows and trusts the proxy CA, then it is possible to perform a man-in-the-middle attack and have the full read/write access to the page source.

6.3.3 Alternatives to proxy implementation and general optimization

As an alternative solution to the proxy implementation there is the concept of working on an improved browser injection mechanism. The idea is to re-implements the Apollo core in a basic module performing browser injection in a controlled fashion. As hypothesized in section 6.2.1, in such module would be fundamental the use of a stealth API hooking method, like for example hooking lower level kernel functions. Further studies are, however, necessary in this sector and this is not the preferred direction since, in spite of the big necessary effort, this might lead to uncertain results.

In the background, with less priority, there is also a number of planned activities leading to the general improvement of the analysis tool in terms of performances and stability. For example, it is an interesting idea to optimize the signature extraction algorithm by parallelizing the elicitation task for a multi-injected page or deducting the triggering patterns using a static page analysis, without refreshing the page.

Another important planned work is the reinforcement of anti-evasion techniques. A certain number of anti-evasion measurements have already been employed, but modern malware adopt a lot of evasion mechanisms in order to detect sandboxes [21], and, for this reason, continuous research and improvement on this field is mandatory to guarantee best results.

7 Conclusions

We explored and examined the problem of Web-Inject based malware understanding the operation principles with the goal of proposing a novel framework for dynamic malware analysis capable of extracting web-injection signatures.

We presented an automated system method, Apollo, aiming to observe the behavior of financial trojans that perform web-injection. It generates signatures in terms of triggering patterns comparing different versions of the same webpage retrieved in a virtualized environment infected with a malware binary. The differences in the DOM versions are analyzed and a purposely modified instance of the web-page is fed to the malware based on the elicitation paradigm which aims at triggering the malicious behavior.

We tested our method on a custom data-set returning excellent results and detecting nearly all the custom-built injections. We defined a procedure to extract a feasible URL targets list in order to test Apollo on real samples supposedly performing web-injection. The testing procedure on real samples did not provide all the wanted results, but we address the lack of web-injection detections mainly to the absence of a solid functioning dataset with a certified ground truth. In fact, the lightly bitter outcome should not simply be associated to the incompatibility of Apollo with the malware samples being tested but also to the very likely circumstances of the samples of being inactive, dormant, not exploiting web-injection, trying to communicate with a shut down C&C server or adopting an evasion mechanism that we could not counteract.

The proposed method sets the goal to be as generic as possible and malware-independent but in the current implementation state it exposes a few limitations. The eliciting approach works effectively in the current solution, but it shows a potential vulnerability which, might be exploited by malware to evade analysis. Despite its limitations, we proposed concrete solutions already scheduled in consistent future works aiming to strengthen Apollo, which has all the prerequisites to become a strong and generic detector for web-injection signatures.

Finally, in this thesis we showed the potential of the API hooking technique and the proposed approaches in the field of the malware analysis. The built tool represents a proof of concepts for the analysis of financial trojans laying the basis for further research and development.

References

[1] G. M. Snow, “The cyber threat to the financial sector”, Federal Bureau of Investigation, Tech. Rep., 2011.

[2] “Zeus malware: Threat banking industry”, Unisys Stealth Solution Team, Tech. Rep., 2010.

[3] K. Worobec, “Fraud the facts 2016”, Financial Fraud Action UK, Tech. Rep., 2016.

[4] A. B. Andrea Continella, “Prometheus: A web-based platform for analyzing banking trojans”, Master’s thesis, Politecnico di Milano, 2014.

[5] C. Criscione, “Zarathustra: Extracting webinject signatures from banking trojans”, 2014.

[6] N. Falliere and E. Chien, “Zeus: King of the bots”, Symantec Security Response, Tech. Rep., 2010.

[7] L. Kharouni, “Automating online banking fraud. automatic transfer system:the latest cybercrime toolkit feature”, Trend Micro Incorporated, Tech. Rep., 2012.

[8] C. W. Stephen Doherty Piotr Krysiuk, “The state of financial trojans 2013”, Symantec Security Response, Tech. Rep., 2013.

[9] C. Wueest, “The state of financial trojans 2014”, Symantec Security Response, Tech. Rep., 2015.

[10] C. Wueest, “Financial threats 2015”, Symantec Security Response, Tech. Rep., 2016.

[11] M. Goncharov, “Russian underground 101”, *Trend Micro Incorporated Research Paper*, 2012.

[12] S. D. Piotr Krysiuk, “The world of financial trojans”, Symantec Security Response, Tech. Rep., 2013.

[13] *Zeus user manual*, 2010.

[14] D. O’Brien, “Dridex: Tidal waves of spam pushing dangerous financial trojan”, Symantec Security Response, Tech. Rep., 2016.

[15] K. L. Report, “Financial cyber threats in 2013”, Kaspersky Lab, Tech. Rep., 2014.

[16] J. v. d. W. e. a. Maria Garnaeva, “Kaspersky security bulletin 2015. overall statistics for 2015”, Kaspersky Bulletin, Tech. Rep., 2015.

[17] J. O. e. a. Michael Bailey, “Automated classification and analysis of internet malware.”, *Springer*, 2007.

[18] F. L. Armin Buescher and T. Siebert, “Banksafe information stealer detection inside the web browser”, *Springer*, 2011.

[19] N. C. e. a. Alexandros Kapravelos, “Hulk: Eliciting malicious behavior in browser extensions”, 2014.

[20] D. Schwarz, “Citadel’s man-in-the-firefox: An implementation walk-through”, Arbor Security Report, Tech. Rep., 2013.

[21] D. Keragala, “Detecting malware and sandbox evasion techniques”, *SANS Institute InfoSec Reading Room*, 2016.

List of Figures

Listings