Journal of Computer Security 0 (2017) 1
IOS Press

Prometheus: Analyzing WebInject-based Information Stealers

Andrea Continella [a], Michele Carminati [a], Mario Polino [a], Andrea Lanzi [b], Stefano Zanero [a], Federico Maggi [a]

[a] *Politecnico di Milano, E-mail: {andrea.continella,michele.carminati,mario.polino,stefano.zanero,federico.maggi}@polimi.it*
[b] *Università di Milano, E-mail: andrea.lanzi@unimi.it*

Abstract. Nowadays Information stealers are reaching high levels of sophistication. The number of families and variants observed increased exponentially in the last years. Furthermore, these trojans are sold on underground markets along with automatic frameworks that include web-based administration panels, builders and customization procedures. From a technical point of view such malware is equipped with a functionality, called WebInject, that exploits API hooking techniques to intercept all sensitive data in a browser context and modify web pages on infected hosts.

In this paper we propose Prometheus, an automatic system that is able to analyze trojans that base their attack technique on DOM modifications. Prometheus is able to identify the injection operations performed by malware, and generate signatures based on the injection behavior. Furthermore, it is able to extract the WebInject targets by using memory forensic techniques.

We evaluated Prometheus against real-world, online websites and a dataset of distinct variants of financial trojans. In our experiments we show that our approach correctly recognizes known variants of WebInject-based malware and successfully extracts the WebInject targets.

Keywords: WebInject, Banking Trojan, Info-stealer

1. Introduction

Malware is still a growing threat [23], with Trojans being one of the most common and dangerous types of malware [18] [33]. A particular type of Trojans, known as Information-stealers or Banking Trojans, allows malware operators to intercept sensitive data such as credentials (e.g., usernames, passwords) and credit cards information. Such malware uses a Man-in-the-Browser technique that infects a web browser by hooking certain functions in libraries and modifying web pages.

More precisely, they use two main attack techniques: (1) modification of network Windows APIs (2) and modification of web pages requested by the web client for specific websites (e.g., banks). The first technique intercepts at network API level any sensitive data that are processed by the browser even in case the connection is encrypted. The second technique typically adds new form fields to the web pages, in order to steal target information such as One-Time Passwords. Each attack module relies on an encrypted configuration file that contains a list of targeted URLs (e.g., well-known banks URLs) in form of regular expressions along with the HTML/JavaScript code that should be injected to a particular website.

Given its flexibility, WebInject-based malware has become a popular information-stealing mechanism nowadays [32].

Different works have been done regarding the analysis and detection of banking trojans [6], [10], [13], [24], [25], [26] but, as explained in Section 3.1, most of them are dependent on a specific malware family or version, and require a considerable effort to be constantly adapted to new emerging techniques. In this paper we aim to precisely characterize WebInject behaviors without relying on the implementation details of a malware but mainly on its own injection behavior.

More precisely, based on findings of our previous work [11], which demonstrated the feasibility of web page differential analysis, we propose Prometheus, an automatic framework for analyzing banking trojans. The proposed system works independently from the implementation details of the malware and the key idea is based on the fact that actions of malware must eventually result in changing the document object model (DOM). Comparing DOMs downloaded in clean machines with those downloaded in infected machines allows us to generate signatures and extract the WebInject configuration file. Our approach also exploits the artifacts left in memory by malware during the injection process and uses them for validating the results of the DOMs analysis. It is important to note that our detection mechanism exploits a different angle of malware behavior and it can be deployed as an additional detection layer along with others approaches [1,2,3] (e.g., AVs, IDS etc.). Our system is orthogonal to any other detection techniques and offers another protection layer that can be used to improve the detection capabilities.

We evaluated Prometheus on a dataset of 135 distinct samples of ZeuS analyzing 68 real, live URLs of banking websites. We manually verified the results of our experiments and we show that Prometheus is able to generate signatures correctly with a negligible fraction (0.70%) of "false differences," which are those legitimate differences wrongly detected as malicious injections. Our experiments also show the efficiency of our system, which is able to analyze one URL every 6 seconds on average.

In summary, this paper extends our previous work Zarathustra [11] with both original concepts, discussions and new results. We make the new following contributions:

- We proposed a new generalization technique that considers multiple infected DOMs from different infected machines along with an improved set of heuristics, which allow our system to guarantee a low false difference rate.
- We combined the web page differential analysis with a memory forensics inspection technique to validate the generated signatures.
- We performed experiments on a new larger dataset, and provided insights from a new data analysis point of view (i.e., classification of the URLs where injections occur typically) that is used for validating our approach.

2. Background on Information Stealers

Information-stealing trojans is a growing [9], sophisticated threat. The most famous example is ZeuS, from which other descendants were created. This malware is actually a binary generator, which eases the creation of customized variants. For instance, as of February 23, 2016, according to ZeuS Tracker[1], there are 8,149 distinct variants that have yet to be included in the Malware Hash Registry database[2].

[1]`https://zeustracker.abuse.ch/statistic.php`
[2]`http://www.team-cymru.org/Services/MHR/`

Notice that this is an underestimation, limited to binaries that are currently tracked. This high number of variants results in a low detection rate overall (40.05% as of February 23, 2016).

Lindorfer et al. [21] measured that trojans such as ZeuS and GenericTrojan are actively developed and maintained. These and other modern malware families live in a complex environment with development kits, web-based administration panels, builders, automated distribution networks, and easy-to- use customization procedures. The most alarming consequence is that virtually anyone can buy a malware builder from underground marketplaces and create a customized sample. Lindorfer et al. [21] also found an interesting development evolution, which indicates a need for forward-looking malware-analysis methods that are less dependent on the current or past characteristics of malware samples or families. This also relates to the fact that the source code is sometimes leaked (e.g., CARBERP, ZeuS), which leads to further creation of new variants [12].

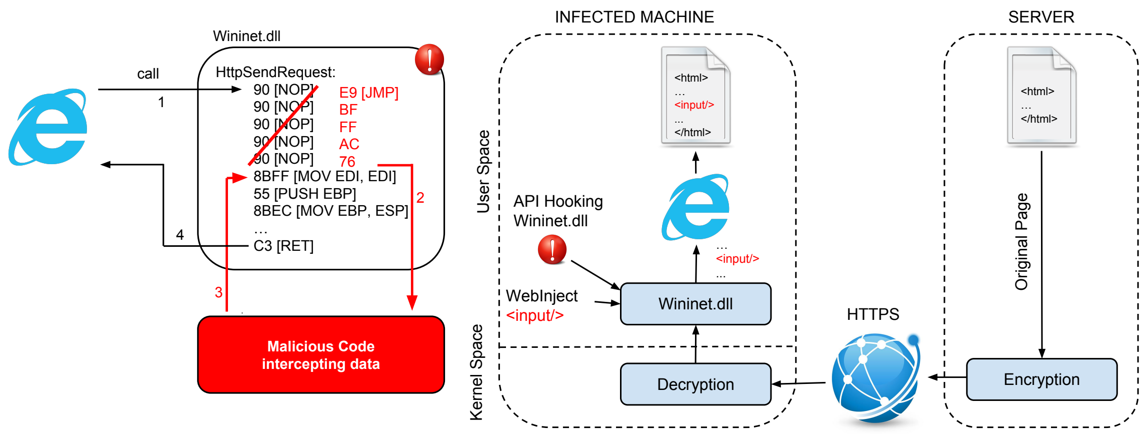
2.1. The underground economy

One of the banking trojan problems is that anyone, independently from their skill level, can perform financial frauds, as the underground marketplace is active and provides all the required resources, like a service industry. For example, Goncharov [14] estimated for the only Russian underground economy a 2.3 billion dollars market.

Grier et al. [15] investigated the emergence of the exploit- as-a-service model, showing how attackers pay for exploit kits to infect victims and propagate their own malware through drive-by downloads. Therefore, even with little or no expertise or ability to write a malware, anyone can simply purchase these "kits," and follow detailed guides and video tutorials sold online. The trojan samples and services available on the underground markets vary, and their price depends on the features. Typically, it starts from 100$ for an old, leaked version, to about 3,000$ for a new, complete version [12]. Furthermore, cybercriminals offer paid support and customization, or sell advanced configuration files that the end users can include in their custom builds. Custom WebInjects can be also purchased for 30-100 [32].

2.2. Man in the Browser attacks and WebInject

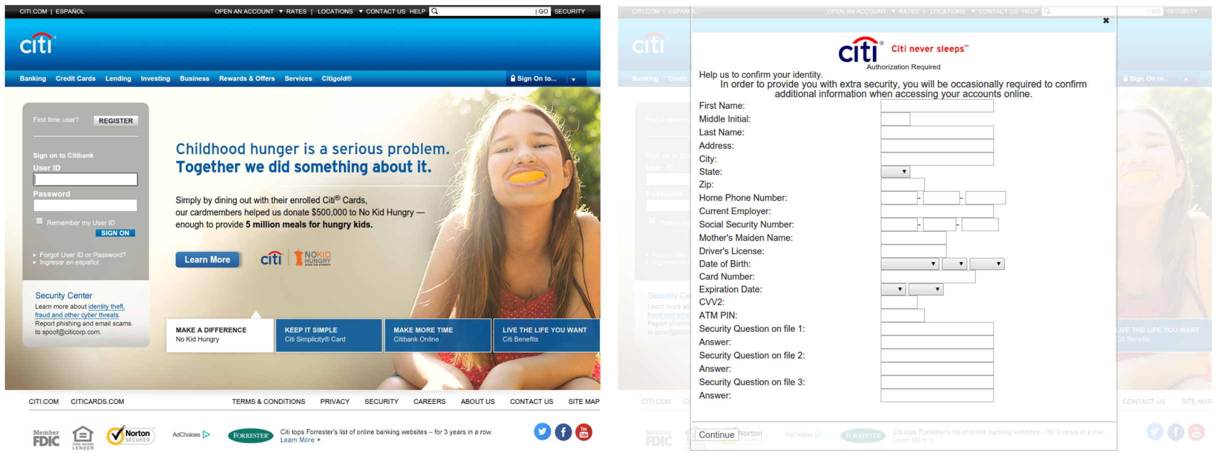
Financial trojans use man-in-the-browser (MitB) techniques to perform attacks. These techniques exploit API hooking and, as the name suggests, allow malware to be logically executed inside the web browser and to intercept all data flowing through it. Also, modern banking trojan families commonly include a module called WebInject [32], which facilitates the manipulation and modification of data transmitted between a web server and the browser. Once the victim is infected, the WebInject module places itself between the browser's rendering engine and the API networking functions used for sending and receiving data. By hooking high-level API communication functions in user-mode code, the trojans can intercept data more conveniently than traditional keyloggers, as they can intercept data after being decrypted. Therefore, the WebInject module is effective even in case an HTTPS connection is used. Figure 1 shows a high level view of the injection mechanism, and an example of inline hooking, in which the trojan overwrites the first five bytes of the *HttpSendRequest* function with a jump to malicious code. Exploiting the high level configuration interface of the WebInject module, cybercriminals can effectively inject HTML code that adds extra fields in forms so as to steal sensitive information. The goal is to make the victim believe that the web page is legitimately asking for a second factor of authentication or other sensitive information (as illustrated in Figure 2). In fact, the victim will notice no suspicious signs (e.g., invalid SSL certificate or different URL) because the page is modified "on the fly" right before being displayed, directly on the local machine.



(a) Example of API hooking.

(b) High level view of the Injection mechanism

Fig. 1. Scheme of the WebInject Hooking mechanism.


citi
Childhood hunger is a serious problem.
Together we did something about it.
Learn More
Citi never sleeps™
Authorization Required
Continue

Help us to confirm your identity.
In order to provide you with extra security, you will be occasionally required to confirm additional information when accessing your accounts online.

First Name:	
Middle Initial:	
Last Name:	
Address:	
City:	
State:	
Zip:	
Home Phone Number:	
Current Employer:	
Social Security Number:	
Mother's Maiden Name:	
Driver's License:	
Date of Birth:	
Card Number:	
Expiration Date:	
CVV2:	
ATM PIN:	
Security Question on file 1:	
Answer:	
Security Question on file 2:	
Answer:	
Security Question on file 3:	
Answer:	

Fig. 2. Example of a real injection on the home page of `online.citibank.com`.

The WebInject module loads an encrypted configuration file. This file contains the list of WebInject rules, which include the target URLs, or regular expressions that match more than a single URL, and the HTML/JavaScript code to be injected into specific web pages. For each rule, the attackers can set two hooks (`data_before` and `data_after`) that identify the portion of the web page where the new content, defined by the `data_inject` variable, is injected. An example of a real WebInject rule is shown in Figure 3.

3. State of the Art and Research Challenges

New families and new versions of info-stealing trojan samples are frequently released [7] [29] [31] and each specific trojan can be customized and obfuscated, generating new, distinct executables. In addition,

```
set_url *.wellsfargo.com/* G
  data_before
    <span class="mozcloak"><input type="password"*</span>
  data_end
  data_inject
    <br><strong><label for="atmpin">ATM PIN</label>:</strong>&nbsp;
    <br /><span class="mozcloak"><input type="password" accesskey="A"
    id="atmpin" name="USpass" size="13" maxlength="14" style="width:
    147px" tabindex="2" /></span>
  data_end
  data_after
  data_end
```

Fig. 3. Example of a real WebInject rule

the custom configuration files are encrypted and embedded in the final executable. For these reasons, manually analyzing all the samples is not scalable. Thus, automatic mechanisms to extract valuable information from encrypted configuration files or for analyzing the activity of an infected machine are needed.

3.1. Information Stealers Analysis and Detection

Bruescher et al. [10] proposed an approach to identify WebInject-based information stealers. The idea is similar to the typical rootkit-detection approach based on recognizing the presence of API hooks in common loaded libraries, especially in browser and Internet-related APIs. To avoid false positives (e.g., legitimate hooks), they inspect the destination of each hook, and check if the pointed module is trusted and correctly signed. The main limitation of this detection approach is the strong dependence on the version of the trojan, on the operating system, and on the hooked browser. Different trojans or future releases could change the list of API functions to hook, or target another browser that uses different libraries. We argue that, in general, user-level API hooking is not the only method to achieve MitB functionalities.

In Heiderich et al. [16] the authors protect the browser from malicious websites that dynamically change the DOM. Although not designed specifically to target information stealers, such mechanism could be applied for recognizing WebInjects. In details, their system instruments the ECMA script layer by proxying its functions. However, as the authors mention, their method can only detect changes of the DOM that occur at runtime, whereas WebInjects work at the source-code level. This means that they are not able to identify modifications (e.g., node insertion) not performed via JavaScript code.

Wyke et al. [30] outlines a sandbox-based system that automatically analyzes banking trojans observing the network traffic, and extracting valuable information such as command and control addresses, in a scalable and extensible way. Although this approach still needs to be manually updated to support newer malware versions, it is a solid tool that can be used to complement our approach.

3.2. Challenges and goals

In Section 3.1 we discussed the limitations of the current techniques used to analyze banking trojans, and the need of automatic and generic mechanisms that are able to extract information from the encrypted files. Our goal is to characterize WebInject behaviors and provide informative feedback to the analysts.

To this end, we extend our previous approach, Zarathustra [11], based on the notion of web page differential analysis, a technique to analyze WebInject-based information stealers leveraging the modifications that they cause in the DOMs of target web pages. Previous results showed that the adopted approach is sound and allows to correctly characterize WebInject behaviors. However, Zarathustra does not fully leverage the potential of differential analysis. First of all, it uses only one infected virtual machine, whereas we show the benefits of using multiple ones. Secondly, in order to discard all the legitimate differences that occur between web pages, Zarathustra needs substantial resources. Our aim is to improve the set of heuristics used in Zarathustra to reduce the number of VMs required. Third, in [11], we did not perform a detailed analysis of the runtime behavior of the trojans, and in particular of the time required by a sample to be activated. Last, we did not explore the possibility of using memory forensics for recovering, at least partially, the valuable content of the encrypted configuration file. Consequently our goal is to improve Zarathustra by combining the web page differential analysis along with a memory forensic analysis.

4. Approach Overview

We divide our approach into two parts: web page differential analysis and memory forensic analysis.

First, we assume that a page rendered on an infected machine includes the injected portions of code. This is reasonable and realistic, as WebInjects-based trojans need to perform code injections. In contrast, the same page rendered on a "clean" machine contains the original source code. Hence, our approach consists in analyzing information stealers looking for evidence of injections in the web pages, and extracting this evidence through a comparison of the web pages with the original ones. This method does not leverage any malware-specific component or vulnerability to observe and characterize the injection behavior, therefore it is more generic by design. However, it could generate many false differences since the content of a web page may vary legitimately. For example, this could be due to server-side script, or advertisements that include dynamically changing content. To cope with this, we collect many versions of the same web page, and compare them to discard legitimate differences that occur between two or more of them. Moreover, to reduce false differences, we designed four heuristic-based filters (i.e., whitelisting clean differences, ignoring node or attribute reordering, filtering harmless differences, identifying repeated malicious injections). As further discussed in Section 8, although filtering legitimate changes creates an opportunity for the attacker to hide in such differences, this is unlikely to happen and easy to remediate collaborating with banks and receiving real-time updates about changes in the web pages.

Secondly, we assume that, during the injection process, trojans leave artifacts of their configuration files, for examples list of targets URLs, attributes, etc. in memory. Our approach is to recover such artifacts from the memory of an infected machine, through memory forensic techniques that inspect the memory of the target applications, and search for strings tokens associated to the definition of regular expressions, part of any WebInject configuration file. We manually examined the extracted regular expressions to obtain a breakdown of the most targeted pages, so to understand where injections typically occur. We distinguish between *login pages*, *post-login pages*, *home pages*, and *all pages* (those regular expressions that cover all the pages of a given domain, such as **exampledomain.com/**). We were not able to classify the 19.5% of the regular expressions, because they did not contain any significant word in the path (e.g., login, logon, access), and the URLs were inactive. As shown in Figure 4, the majority (50.6%) of the sample we analyze targets all the pages of the domains. This strategy is more effective for cybercriminals, because their injections can still succeed even if the URL of the targeted page changes.

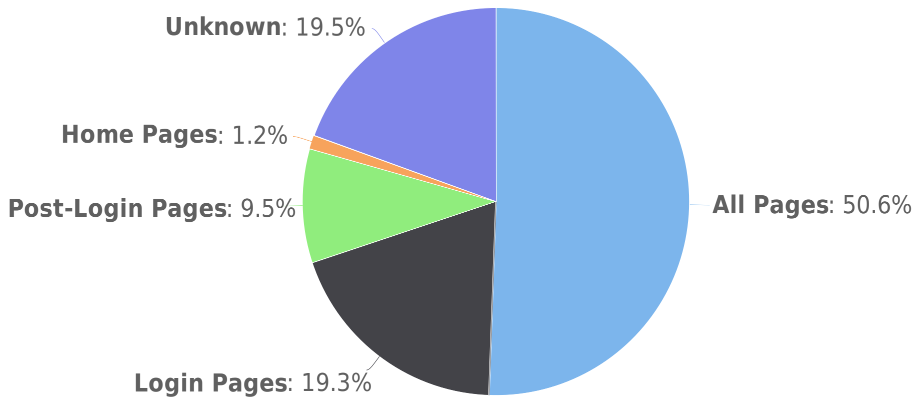


Fig. 4. Classification of the 694 regexes extracted by memory forensic inspection according to the kind of pages targeted.

Instead, regexes targeting post-login pages are a small fraction (9.5%). Thus, our current approach can be used to characterize the majority of the injection behaviors, leaving post-login pages as future work.

As a consequence, extracting these artifacts represents a further indication of the malicious activity, allows us to validate the results of the web page differential analysis and gain further insights about the use of WebInject techniques.

4.1. Application Scenarios

Prometheus produces signatures in the form of XPath expressions (Figure 5), which allow to check, on the client side, whether a web page is currently being rendered on an infected machine or, more in general, if a page of interest is targeted by a specific sample. In practice, as depicted in Figure 6, we foresee a centralized server and several consumers that send URLs of interest.

In **Scenario 1** the URLs are received by the clients (e.g., antivirus module, browser-monitoring component). The server replies with the list of signatures related to the requested URL(s). In the case of an antivirus, the browser-monitoring component (similar in spirit to Google Safebrowsing) can request the signatures of each browsed URL, and verify if any of the signatures match.

The second scenario that we envision, **Scenario 2**, has to do with research and large-scale monitoring. More precisely, we believe that Prometheus could be a good companion for initiatives such as ZeuS or SpyEye Tracker, VirusTotal, Anubis, Wepawet and similar web services that receive large daily feeds

```
{
  "signatures": [
    {
      "xpath": "/html[1]/body[1]/table[3]/tr[1]/form[1]/input[13]",
      "value": "<input name=OTP type=password/>"
    }
  ]
}
```

Fig. 5. Example of a generated signature for a given URL and sample

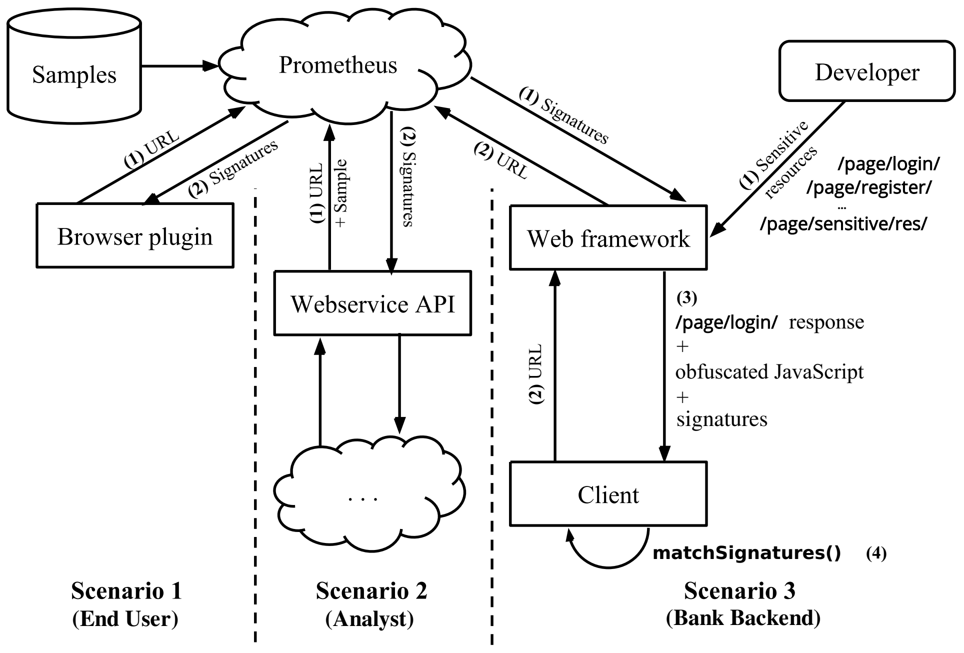


Fig. 6. Three application scenarios described in Section 4.1

of malware samples and URLs to analyze. In this context, Prometheus can be used to automatically determine whether a sample performs web injection against a given URL, regardless of whether it is ZeuS or SpyEye, or some other unknown family, and to extract the portion of injected code.

One last application, **Scenario 3** envisions an IT administrator or web developer (e.g., of online banking backends), which provides a feed of URLs likely to be targeted by WebInject-based malware once in production. This scenario was suggested by a developer of a large national bank with which we collaborate, who noticed the lack of a centralized solution to determine whether their clients were infected by a banking trojan. In this context, the developer would like to have a web framework that offers an API to programmatically mark sensitive resources (e.g., /page/login/, or those that contain forms). Marked resources will be processed by the web framework right before the HTTP response is sent to the requesting client. The web framework will then append a JavaScript procedure that, once executed on the client, performs a similar check to the one described in the aforementioned “Safebrowsing” scenario.

Generally, we envision our system to be deployed in collaboration with banks that provide feedbacks whenever their web-sites are massively updated, and strictly collaborate to update signatures. This would avoid almost all the false differences, and the possibilities for attackers to hide their injections. Moreover, as discussed in Section 8, the signature-matching algorithm can be designed to match only the leaf nodes of the XPath, hence being more flexible with respect to legitimate page changes.

5. Threat Model & System Evasion

To reduce the possibilities for malware to tamper with the browser-monitoring component described in the previous scenario, the signature matching phase and the analysis of the system can be performed in kernel-space or even at the hypervisor level. In fact, our solution is orthogonal to any monitoring techniques. The enforcement system can be deployed using a method similar to the one presented in [2]. In this paper, the authors designed a reference monitor that operates at the hypervisor level and that is able to validate signatures extracted from user-space environment. Such a system uses a trusted-path approach in order to validate the user-space data-flow along with the trusted view of the OS memory. Other hypervisor systems [4] can also be used to provide a secure monitor that acts in user-space with the same hypervisor functionalities and that cannot be tampered by any user/kernel space malware.

In our threat model we assume that the attacker does not have any physical access to the machine and, therefore, cannot perform any hardware-based attack (e.g., a DMA attack Wojtczuk, 2008) and cannot tamper with the hypervisor operations. We assume that our hypervisor starts during the boot process of the machine and it is the most privileged hypervisor on the system. Furthermore, it is also possible to leverage late-launching to load Prometheus hypervisor after the boot. To do so, however, we have to slightly relax our threat model. Indeed, we must assume that either there is no malware on the machine before we launch Prometheus or that we leverage an integrity checking technique to ensure that the hypervisor is not altered at load time. Despite this requirement, a hot-bootable hypervisor can be quite useful in scenarios in which it is not possible to restart the machine (e.g., when it provides some critical service). We assume a powerful attack that can act at the user-space level and can modify any browser application. Anyway, since all the monitored data come from the network, the hypervisor is the first component that can get access to such data, hence it can predict the browser behavior, and consequently enforce the signatures in a trusted way.

6. System Design & Implementation

The input of our system is a list of URLs that need to be monitored (e.g., banks URLs) and a malware sample. Given the inputs, the system performs the web page differential and the forensic memory analysis in three phases: (1) Data Collection, (2) Data Processing, and (3) Signatures Generation.

As a final output, our approach produces a list of differences per URL, which precisely identify the portion of injected or changed code and represent our signatures. Each signature is defined by (1) the XPath of the affected node or attribute, and (2) the injected content (Figure 5). As a matter of fact, these signatures partially reconstruct the configuration file.

In Figure 7 we reported the overview of our analysis, given a malware sample to analyze. In the following Sections we provide the details of each phase.

6.1. Phase 1: Data Collection

Our system retrieves (1) the DOMs objects, and (2) the memory dump of the web browser process. To this end, the system executes a small set of virtual machines. Half of them are infected with the malware sample and the other half are clean. Afterwards, the system visits the set of URLs with each VMs, collects the DOMs and the memory dump.

6.1.1. DOMs Collection

Each VM receives a list of URLs as an input, and it visits each URL with a `WebDriver`-instrumented browser. For each visited URLs, Prometheus saves the resulting DOMs and it stores them as serialized representations of existing nodes, including any DOM manipulations performed by client-side code at runtime while the page loads. The DOMs comprise the content of the nodes in the pages, including script tags. Since the content of a web page may vary legitimately (e.g., server-side scripts, advertisement

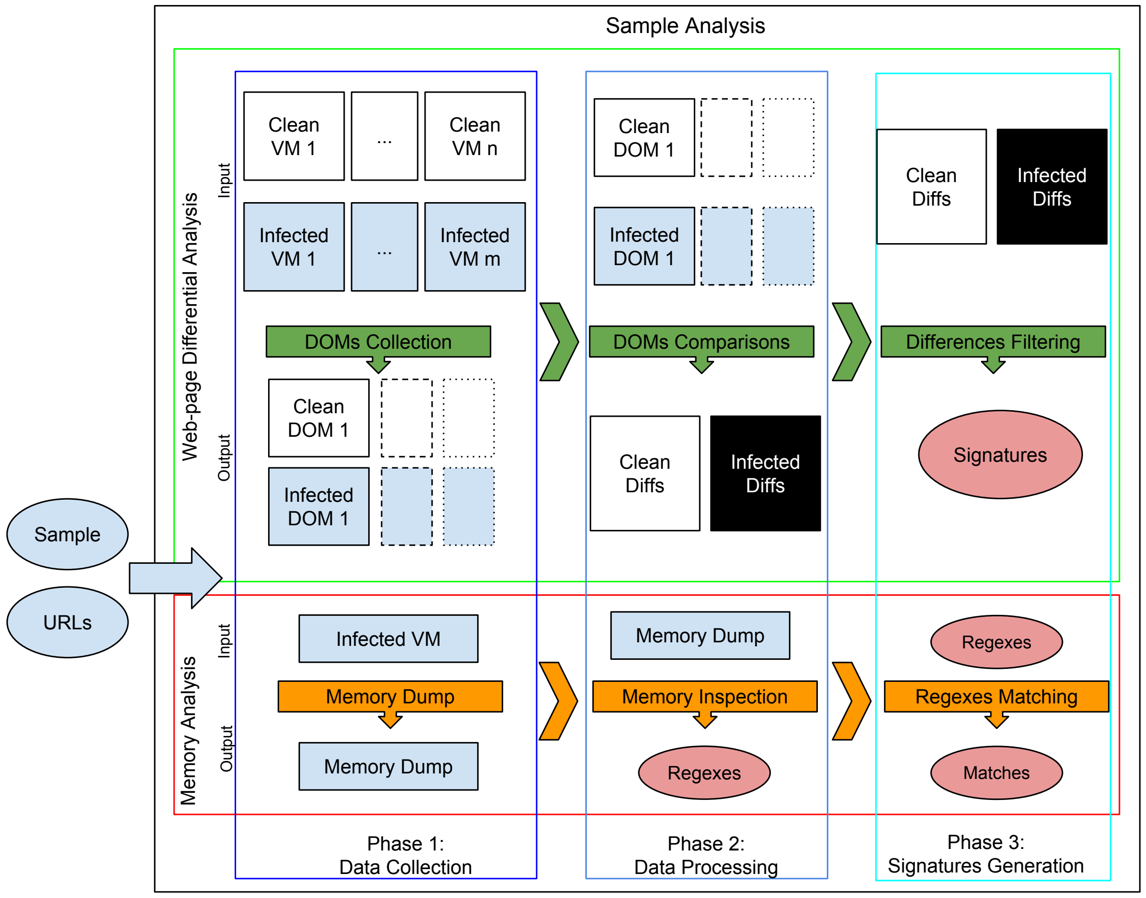


Fig. 7. Overview of a sample analysis. The web page differential analysis, and the memory analysis are performed in parallel.

inclusions), the DOMs collection phase is performed on multiple clean and infected machines. This is used to eliminate legitimate differences that occur between two or more clean VMs. When this phase terminates, all the dumped DOMs are stored and labeled as “clean” or “infected”.

6.1.2. Memory Dump

In this phase, Prometheus executes a new VM along with the target malware sample. To trigger the malware infection and obtain the target URLs, Prometheus runs the browser inside the VM. Then, the system dumps the memory of the VM from the host OS by using the `Cuckoo` feature. This allows us to avoid creating any artifacts that can be exploited by the malware. At the end of this phase, the memory dump is stored as a file on the disk. We use a separate VM for the memory analysis to avoid false positives due to the execution of multiple applications on the same VM.

6.2. Phase 2: Data Processing

During the second phase of our approach, Prometheus compares the various DOMs retrieved for each URL (downloaded by distinct VMs) and extracts the WebInject target URLs from the memory dump obtained in the previous phase.

6.2.1. DOMs Comparison

We compare the DOMs in the following way. For each URL, we consider one random clean DOM as a reference, and we compare all the others against it. We rely on `XMLUnit`'s `DetailedDiff.getAllDifferences()`, which walks the DOM tree, and extracts the following differences:

– **Node insertion**: Most information stealers add new fields in forms, injecting one or more `<input/>` nodes. Thus, it is crucial to identify this case, as it is one of the most common injections.
– **Attribute insertion**: This identifies injections that are mostly related to malicious JavaScript code. In the common case, trojans add attributes such as `onclick` to bind JavaScript code and perform malicious actions whenever certain user-interface events occur.
– **Node modification**: This occurs when trojans modify the content (e.g., text enclosed in) of an existing node. Often the target node is `<script>`.
– **Attribute modification**: This occurs when trojans change the value of an existing attribute (e.g., to change the network address of the server that receives the data submitted with a form, or to modify the JavaScript code already associated to an action).

At the end of this phase, for each processed URL the extracted differences are inserted into two lists (clean differences and infected ones), according to the compared DOM label. Each difference is composed by three elements:

- **Type**: The nature of the difference (node insertion, node modification, attribute insertion, or attribute modification).
- **XPath**: The XML path to the node that is affected by the difference.
- **Content**: The value of the difference (e.g., in the case of a node insertion the content is the HTML node inserted with all its own attributes).

6.2.2. Memory inspection

When the infected memory dump is generated, it is inspected in order to extract the WebInject targets. To extract the target URLs and regular expressions, we developed a `Volatility`[3] plugin based on `YARA`[4]. This plugin scans the memory dump, looking for all the strings that match a `YARA` rule. In particular, since we observed that the URLs and the regular expressions are loaded in the browser's memory, the plugin inspects only its address space. We defined a regular expression that matches the pseudo-URL format of the WebInject target URLs (e.g., `domain.com/ibank/transfers/*`, `bank.com/login.php*`). Moreover, our `YARA` rule filters the matched strings that are close to each other. In fact, since we noticed that the WebInject targets are allocated sequentially, we leverage this fact to exclude all the matching strings that are not WebInject targets.

[3]`https://github.com/volatilityfoundation/volatility/`
[4]`http://plusvic.github.io/yara/`

6.3. Phase 3: Signatures Generation

Prometheus now filters out the legitimate differences employing four heuristics. Moreover, it exploits the information extracted by memory analysis in order to validate the generated signatures.

6.3.1. Differences Filtering

The two lists of differences produced in the previous phase are filtered according to four heuristics. The objective is to eliminate the legitimate differences, therefore reducing the false difference rate. We designed and implemented the following four heuristic filters:

Heuristic 1: Whitelisting Clean Differences. In certain pages, there may be some nodes that change very often their content (e.g., calendar, clock, advertisement and so on). This kind of nodes generates a lot of differences that refer to the same node but with different content. These differences are present in both the clean and infected list. For this reason, we remove all the infected differences with the same type and the same XPath of a clean difference. For example, thanks to this filter we are able to discard the differences caused by advertisements that dynamically change their message.

Heuristic 2: Ignoring Node or Attribute Reordering. In other pages, it may happen that some nodes with a fixed content (mostly JavaScript) are omitted or moved in different places inside the web pages. We remove all the infected differences that have the same type, the same last node in the XPath, and the same content of a clean difference. For example, this filter discards those differences caused by an advertisement that presents a fixed content but that is dynamically loaded in different positions of the page. Even if this scenario seems unlikely, we found it occurring in different web pages.

Heuristic 3: Filtering Harmless Differences. Since malware authors are inserting new elements in order to steal data from the victims, we are interested in differences that are caused by insertions or modifications. For this reason, all the differences that indicate other changes (e.g., deletion of nodes) are filtered out. Moreover, we whitelist attributes that are harmless according to our attacker model (i.e., value, width, height, sizset, title, alt) and Prometheus automatically filters them out.

Heuristic 4: Identifying Malicious Injections. A typical injection has the following characteristic: it is present in all the DOMs downloaded by infected machines, but not on the DOMs downloaded by clean machines. Furthermore, an injection is defined to insert a content after, or before, a node. Thus, we can assume that, for a give sample and URL processed by multiple VMs, an injection has always the same content, and injects in the same node, even if the node changes its XPath. So a typical web injection refers to the same last node of an XPath that may vary somehow. Hence, we filter out all the differences that are not present (with the same type, the same last node of the XPath, and the same content) in at least $\varepsilon\%$ of the infected DOMs. Since a sample may not activate on some VMs (e.g., because malware authors take countermeasures to prevent dynamic analysis) it may happen that a sample manifests its behavior only on a subset of the VMs. The threshold ε copes with this problem.

The infected differences that pass this filtering process are considered as malicious.

6.3.2. Regular Expression Matching

In this last phase, the regular expressions extracted through memory inspection are examined to validate the results of the web page differential analysis. If an injection is found at a URL not matching any of the regular expressions, it is probably a false difference. Second, the number of matching regexes allows us to rank the URLs according to how often they are targeted. This is useful in case we want to limit the number of URLs processed during an analysis.

6.4. Implementation

The Prometheus prototype that we implemented is composed by a back-end and a front-end. The back-end receives as input the specification of the submitted analysis, then it schedules it, managing the available resources (VMs), it processes the data as explained in Section 6, and stores the results. We integrated Prometheus with `Cuckoo`[5], an open-source sandbox that interacts with the most common virtual machine managers. We used `Oracle VirtualBox`[6] as VMM. Inside each VM, we installed and configured `WebDriver`[7], a platform- and language-neutral interface that introspects into, and controls the behavior of, a web browser and dumps the DOM once a page is fully loaded.

The front-end is a web application, through which users can submit samples, or URLs, and obtain the results of their analysis or previous ones. Finally, we implemented a JavaScript web-socket in order to dynamically show results during the analysis.

7. Experimental Evaluation

We evaluated our implementation of Prometheus against 68 real, live URLs of banking websites targeted by 135 distinct samples of information-stealing malware. We manually verified the activity of each sample when the browser was rendering the targeted websites, to make sure that all the samples were working as expected. While doing this, we conducted a preliminary experiment to measure the dormant period after which the malware triggers its malicious injections (these results are presented in Section 7.2). We found out that 33.82% of the web pages were affected by at least one injection, which Prometheus detected correctly, as summarized in Table 1.

Our first goal was to quantify the correctness of the signatures generated by Prometheus (Section 7.3 and 7.4), and the outcome of the memory analysis (Section 7.5). Then, we assessed the computational resources required by Prometheus in function of the number of DOMs compared per URL (Section 7.6). Last, we evaluated the impact of false positives caused by our signatures (Section 7.7) during a typical real user's browsing activity.

In all our experiments we deployed Prometheus on a 2.0GHz, 8-core Intel machine with 24GB of memory. We used VirtualBox as a virtual machine monitor, and each VM was equipped with 1GB of memory, enough to run Internet Explorer 8 on top of Windows XP SP3.

7.1. Dataset

Our dataset is based on the ZeuS family, which is by far the most widespread information stealer that performs injections. According to the conservative statistics by ZeuS Tracker, as of February 23, 2016 there are 518 known C&C servers (179 of which active), and a low estimated antivirus detection rate (40.05%, zero for the most popular and recent samples). We also conducted a series of pilot experiments with SpyEye, which is less monitored than ZeuS; thus, it is more difficult to obtain a large set of recent samples. However, SpyEye uses the same injection module of ZeuS, as described by Binsalleeh et al. [6], Buescher et al. [10], Sood et al. [27]. For these reasons, for the purpose of evaluating the quality of our signature-generation approach, we decided to select ZeuS as the most representative information stealer

[5]`http://www.cuckoosandbox.org/`
[6]`https://www.virtualbox.org/`
[7]`http://docs.seleniumhq.org/projects/webdriver/`

that generates real-world injections. We tested 196 (and counting) samples, but 61 of these failed to install or crashed, leaving 135 distinct samples.

We constructed a list of target URLs starting from a webinjects.txt leaked as part of the ZeuS 2.0.8.9 source code[8], and adding some new URLs extracted from initial memory analyses. However, we removed most of the URLs contained in the initial list, because they were inactive. The final URL list that has been evaluated was composed by 68 distinct URLs. Building a list of URLs from webinjects.txt files found in the wild allowed us to deal with real-world targeted pages.

Finding a way to measure the quality of the evaluation results was not an easy task, since to determine the real injections performed by a sample we should have decrypted and checked its own WebInject configuration file. We initially created a controlled botnet, and we built a ZeuS sample with a customized webinjects.txt. However, we wanted to test Prometheus on real samples found in the wild. Therefore, we decided to manually analyze the results provided by Prometheus to assess their correctness. We exploited the knowledge created by memory forensic analysis in order to validate the results. In details whether a difference is detected on a URL that does not match any of the regexes extracted from infected memory dumps, the system flags it as a false difference. This reduced the volume of data to analyze manually without introducing any experimental biases.

7.2. Delayed activation

One common anti-analysis technique is to delay the real execution of a malware sample. To this end in order to conduct a sound analysis, it is necessary to check the time in which a sample is activated. For this reason, we conducted some experiments in order to estimate the activation time to be used. In the further experiments, all the infected VMs wait for such time slot before starting the analysis.

In order to estimate the activation time we implemented a specific analysis package for `Cuckoo`. The module works as follows. `Cuckoo` starts a virtual machine and executes the selected malware sample. Afterwards, it starts the browser and waits for a prefix amount of time. When the time is expired, the system dumps the application memory, and terminates the VM. In the next step, we examined the memory dump with Volatility. In particular, we used the `apihooks` [9] plugin to search all the API hooks installed by the malware. We are interested in looking for any hook on the `WININET` DLL. In fact, these hooks are installed by malware to perform Man in the Browser attacks. Hence, when we find any of these hooks it means that the sample was active. Then, to estimate the activation time we repeated our analyses applying a bisection algorithm in order to find, with a reasonable precision, the instant of activation of the sample. We repeated these experiments on different active samples. We obtained several empirical results and in the worst case the activation time was 50 seconds.

7.3. False Differences discussion

A false difference occurs when Prometheus detects a benign difference in a web page and classifies it as a malicious injection. We observed that the main cause of false differences are JavaScript-based modifications. Most of the modern websites contain JavaScript code that modifies the DOM of the web page at runtime, loading dynamically changing content. Furthermore, sometimes server- side scripts generate different JavaScript code every time the page is requested (e.g., advertisement). Even so, we

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

[9]`https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/apihooks.py`

Table 1
Most injected domains

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

managed to discard almost all the legitimate differences and, with all our heuristics enabled, the overall false difference rate is 0.70%. Furthermore, most of false differences are distinguishable looking at the results extracted by memory analysis, since a difference detected on a URL that do not match any of the extracted regular expressions is certainly a false difference. Therefore, validation through memory forensic reduces the false difference rate to 0.26%. In particular, we preferred to not automatically discard the differences related to URLs that do not have references in memory. This is done because, in the case the memory analysis fails in extracting some regular expressions, we still consider all the malicious differences. We analyzed the effects of the number of VMs, and of the threshold ε (**Heuristic 4**) on the false differences (Figures 8 and 9). We observed that the number of VMs is the most effective parameter in the reduction of false differences. However, since the number of VMs required to guarantee a low false difference rate is high, at least 25, and this can worsen the performance because of the overloading, using a high value of ε helps in guaranteeing such results with a lower number of VMs (10-12).

One of the previous heuristics used by Zarathustra ignored dynamic DOM differences. The assumption was that malware always inserts at least one static node or attribute, which would be still visible even when JavaScript is disabled. This heuristic had proved to be the most effective in reducing false differences. However, we found samples that perform only JavaScript injections, therefore we disabled such assumption. Figure 10 shows Zarathustra's false difference rate when this heuristic is disabled. While Zarathustra reaches 1.0% of false difference rate, Prometheus reduces the false difference rate even using less VMs (Figure 9).

7.4. Missed Differences discussion

A missed difference occurs when Prometheus does not identify a malicious injection in a web page. Since the DOMs comparison process is deterministic, there are only two cases in which this can happen. The first one occurs when a high threshold ε is used in **Heuristic 4** and a sample fails to execute in some of the infected machines. As explained in Section 6.3, the differences that are not present in most of (depending on ε) the infected machines are filtered out, so if a sample manifests its behavior just in few of the infected machines, for example if it uses a randomized activation time, its injections might be discarded. Although we did not observe such behavior, this problem can be easily solved by properly tuning the sleep time and the ε threshold. The second case is also related to **Heuristic 4**. Since the heuristic is based on the assumption that each sample injects a static content, if a sample injected different dynamic content on the same web page every time it is executed, it would evade our system. However,

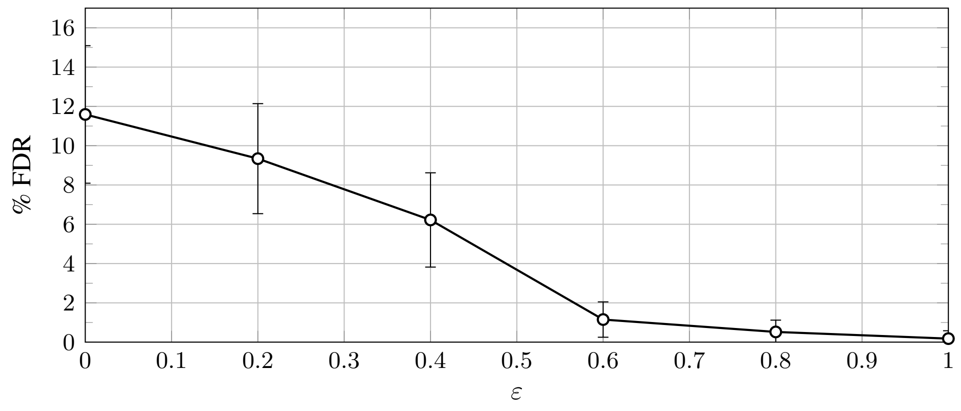


Fig. 8. False Difference Rate depending on the ε threshold evaluating 68 distinct URLs and using 12 VMs (6 clean ones and 6 infected ones).

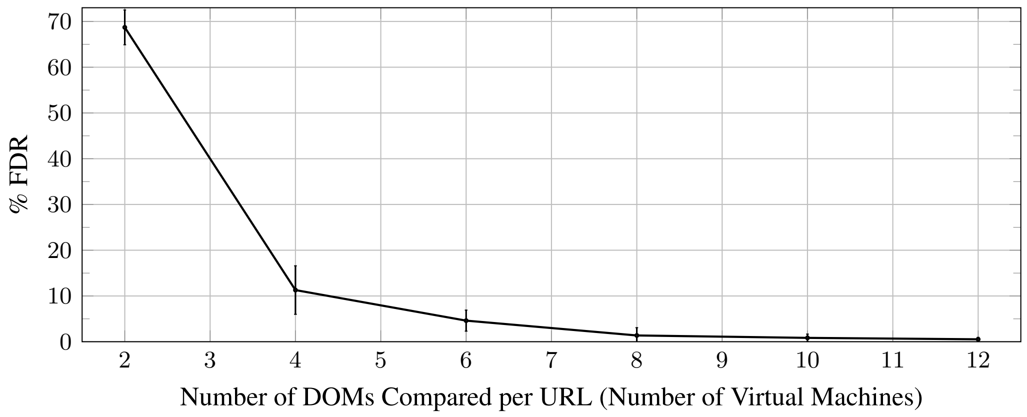


Fig. 9. False Difference Rate depending on the number of VMs used processing 68 distinct URLs with threshold $\varepsilon = 0.8$.

Table 2
Top five regular expressions found

REGULAR EXPRESSION	**FREQUENCY**
http://*odnoklassniki.ru/*	81.3%
http://vkontakte.ru/*	81.3%
odnoklassniki.ru/	81.3%
https://online.wellsfargo.com/das/cgi-bin/session.cgi*	76.4%
https://ibank.barclays.co.uk/olb/x/LoginMember.do	75.6%

this case never happened during the evaluation, and all the samples in our dataset performed static content injections. Samples injecting dynamic content could require visiting the same pages multiple times on each VM.

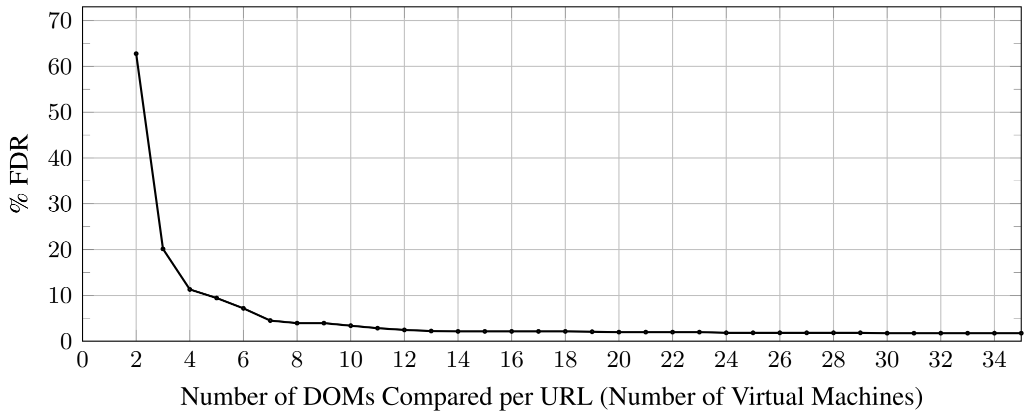


Fig. 10. Zarathustra's false differences for an increasing number of clean VMs.

7.5. *Results of memory analysis*

Prometheus extracted 694 distinct regular expressions through memory forensics (Table 2). As we expected, we observed that true injections were present in URLs that matched some of the extracted regular expressions. This is a further proof that can help to distinguish real injections from false differences. Another important finding that emerged from the results is that not all the regular expressions imply injections. There are some cases in which some URLs were not injected, even if they were present in the memory-extracted targets list. The cause of this could be that the injections failed because the hooking points configured in the WebInject configuration file were wrong or old (because the web page changed and does not contain that HTML code anymore), or simply the sample just monitors the URLs stealing the data submitted by the victims, without injecting new contents.

In conclusion, our results show that combining both the web page differential analysis and the memory forensic inspection is important in order to gain more insights on the WebInjects' behavior, and to guarantee low false differences.

7.6. *Performance*

We measured the execution time of Prometheus. Prometheus has been designed and implemented to parallelize all computations. Furthermore, the approach used to perform DOMs comparisons is asynchronous, and all computations are executed in parallel as soon as the required data is available. For this reason, the execution time is dominated by the time required by the VMs to sequentially visit each URL and dump its DOM, while the time required to compare DOMs and generate the signatures is negligible. Prometheus performs a sample analysis on 68 URLs using 10 VMs in about 7 minutes. The sample analysis includes also the memory forensic inspection, which is performed in parallel and requires less time than the web page differential analysis. As shown in Figure 11, Prometheus is able to process each URL in little more than 4 seconds when 2 VMs are used. The chart shows that Prometheus scales well increasing the number of VMs, with just a little overhead. However, when the number of VMs is higher than 10 the overhead slightly increases. This is due to the overload on the single physical machine, and to the fact that all the VMs network traffic flows through the single virtual interface between VirtualBox

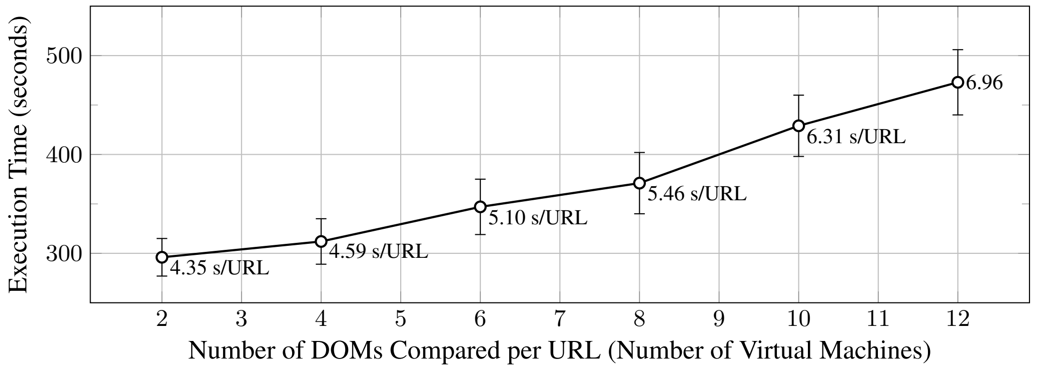


Fig. 11. Speed and Scalability of Prometheus: Mean time required to process 68 URLs for each sample. The labeled points indicate the mean time required to process a single URL.

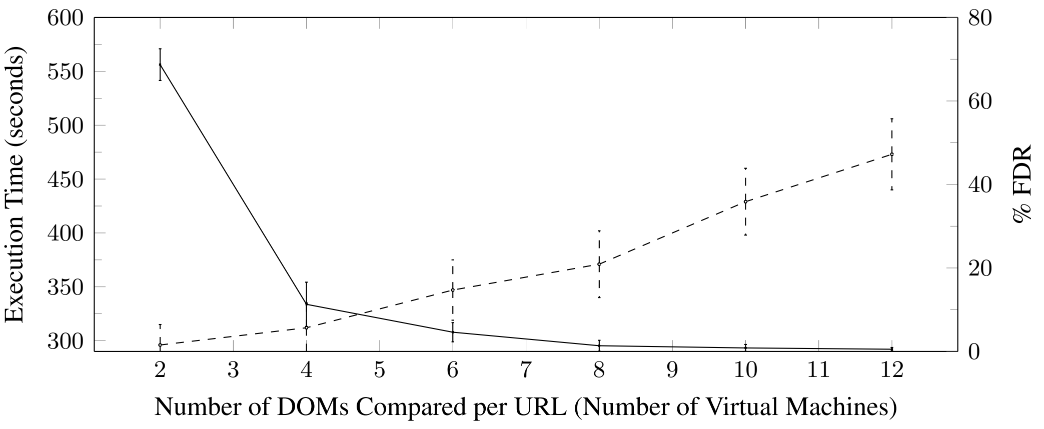


Fig. 12. Trade-off between Performance and False Difference Rate. The dashed line refers to the execution time required to process 68 URLs. The solid line refers to the False Difference Rate analyzing 68 distinct URLs with threshold $\varepsilon = 0.8$.

and the host OS. Figure 12 shows the trade-off between performance and false difference rate depending on the number of VMs.

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

7.7. Distributed crawling experiment

We ran a distributed crawling experiment to evaluate the generality of the signatures generated by Prometheus. With generality we mean the applicability, with an acceptable false positive rate, of a signature to an arbitrary URL. We developed a Chrome extension that sends to our secure servers the DOMs of the visited web pages. Between October and December 2014, we distributed our Chrome extension to trusted users and we collected 8,226 DOMs. To guarantee users' privacy, we did not record any information about them, keeping data anonymized. We further set up a machine on which we installed and configured `WebDriver` to control a Chrome browser, with our extension embedded, and we automatically downloaded the DOMs of the home pages of the top 10,000 Alexa[10] websites. We then checked the signatures generated by Prometheus on the 18,226 DOMs collected, assuming that such DOMs are clean.

As we said our signatures are formed by an XPath and a value. Furthermore, each of our signatures refers, by design, to the URL from which it was generated. That means that each signature can be used to check, client-side, if the web page of the specific URL is currently being rendered on an infected machine. Instead in this experiment, we want to ignore the URLs from which the signatures are generated, and measure the amount of DOMs matching at least a signature. In this way, we can evaluate the generality of such signatures, and measure their dependency from the URLs they refer to.

We performed the matching process twice. In the first case, we considered only the XPath to verify the match of a signature against a DOM. In the second case, we considered both the XPath and the content of the signatures. We made also another distinction, checking all the generated signatures, or only those validated by the memory analyses. Looking just at the XPath, the percentage of DOMs matching at least one signature is very high: 97.74% considering all the signatures, and 94.43% considering those validated through memory forensic. This is mainly caused by signatures affecting *script* tags (e.g., */html[1]/body[1]/script[2]*) that are injected by cybercriminals to verify, client-side, the presence of all the fields in the forms. Checking both the XPath and the content of the signatures, the percentage reduces but it is still quite high: 25.99% considering all the signatures, 24.86% only those validated through memory forensic. This is caused, for example, by signatures relating the import of `JQuery` (e.g., *"xpath": "/html[1]/body[1]/script[2]", "value": "@src='https://ajax.googleapis.com/ajax/libs/jqueryui/1.7.1/jquery-ui.min.js' "*), used by the attackers to create and inject fancy forms. Since the amount of DOMs matching at least a signature is high in all the cases, we cannot ignore the dependency of the signatures from the URLs they refer to, as doing so will cause a high false positive rate during the signature matching process.

In conclusion, this experiment demonstrates that the signatures are strictly dependent on the URLs from which they are generated, and shows the necessity to design a more sophisticated signatures-matching algorithm that takes in consideration the relation between signatures and URLs.

8. Limitations

One limitation of Prometheus is the assumption behind **Heuristic 4**: The content of the injections performed by WebInject-based information stealers is assumed to be static, which implies that the injections performed by the same malware have always the same content. The assumption held for all the samples we analyzed, but Boutin [8] observed a new WebInject mechanism, in which the injected content is delivered by the C&C server each time the injection is performed. Performing content-dynamic injections, this mechanism could evade our system. Therefore, our approach will need to be revised, refining the set of heuristics, or visiting the same page multiple times on each VM in order to extract all the variants.

Our second discussion point is that malware operators could rewrite the injected code, introducing no-op DOM nodes with the goal of evading the signatures generated by Prometheus: adding an additional `<div/>` wrapper to a page (in a random position), for instance, would circumvent a naive use of our signatures (i.e., if the full XPath is considered from the root to the leaves). However, none of the samples in our dataset adopted this technique. In addition, although we leave the implementation of a proper

[10]`http://www.alexa.com/topsites`

signature matching algorithm to future work, we are aware that there is an accuracy trade off between matching the entire XPath expression of a signatures versus matching only the leaf nodes. In the latter case, the signature verification process will be more flexible and can be effective even in the case of pages subject to large modifications.

Third, the memory forensic analysis could be evaded by samples using custom sparse data structures, for example splitting the regexes and storing each part separately. In this case the analysis would require more expensive carving techniques to be able to identify and extract the WebInject targets.

Furthermore, Prometheus might not be able to properly handle WebInjects that do not target specific domains and, instead, focus only on common paths (e.g., “/WebPortal/login?bankid=”).

Since we take the original banking website as an oracle, an injection that matches exactly with a benign difference would not be considered as malicious. For example, this happens if the website is updated with a new form input that matches the very same XPath expression of an injection. Not only this is very unlikely to happen, it is also very easy to remediate by leveraging feedback from the bank whenever its site is updated, or possibly by requesting an update of the signatures for that domain. It is indeed reasonable to envision Prometheus deployed within a bank information system: this use case would avoid most, if not all, the venues for false differences as a fully up-to-date model of the clean website would always be available. Similarly, Prometheus can easily monitor authenticated web pages, which are not a limitation when our system is deployed by the website provider (e.g., bank).

Another obstacle is malware that adopt anti-analysis techniques. However Prometheus can be ported, as is, on a bare metal hypervisor [20]. Even if certainly not impossible, it is definitely harder to detect bare metal environments.

9. Conclusions & Future Work

In this paper we presented Prometheus, an automated system to analyze the client-side behavior of financial trojans that perform web injections. Prometheus generates signatures comparing the different DOMs retrieved by infected and not-infected VMs. These differences are then filtered by using some heuristics, in order to discard legitimate ones. This approach is combined with a memory forensic inspection, to extract the WebInject target URLs and validate the signatures generated by the web page differential analysis. The main advantage of this approach is the independence from the implementation details of the analyzed malware. We evaluated Prometheus on a dataset of 135 distinct samples of ZeuS, analyzing 68 real, live URLs of banking websites. The results show that Prometheus correctly identified the injections performed by the analyzed trojans with a low false difference rate (0.70%). Validation through memory forensic reduces errors down to 0.26%. Furthermore, Prometheus scales well with the amount of available resources, and it is able to generate the signatures for 1 URL in about 6 seconds.

Besides the development of a proper signature matching algorithm, future research should concentrate on more advanced uses of WebInjects. As mentioned in the Section 8, dynamic-content injections (described in [8]) require visiting the same pages multiple times on each VM to capture all the different injections. Other advanced WebInjects, described by Kharouni [19], perform attacks that may not result in DOM modifications. An example is a banking web application that allows to divert a wire transfer by simply modifying one, single parameter in an outgoing HTTP request, the respective HTTP response (e.g., page that confirms the result of a transaction), and all the subsequent pages. When such fraudulent transfers to an attacker-controlled account have been made, these modules are able to hide the transactions and revise the current account balance in order to make the victim unaware of the fraud.

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

References

[1] Srivastava, Abhinav, Andrea Lanzi, Jonathon Giffin, and Davide Balzarotti. *Operating system interface obfuscation and the revealing of hidden operations*. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2011.

[2] Fattori, Aristide, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. *Hypervisor-based malware protection with AccessMiner*. Computers & Security, 2015.

[3] Lanzi, Andrea, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. *Accessminer: using system-centric models for malware protection*. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2010.

[4] Sharif, Monirul I., Wenke Lee, Weidong Cui, and Andrea Lanzi. *Secure in-vm monitoring using hardware virtualization*. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2009.

[5] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. *Automated Classification and Analysis of Internet Malware*. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection (RAID), 2007.

[6] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. *On the analysis of the Zeus botnet crimeware toolkit*. In Proceedings of the Annual International Conference on Privacy, Security and Trust (PST), 2010.

[7] Blueliv. *Chasing cybercrime: network insights of Dyre and Dridex Trojan bankers*. Cyber Threat Intelligence Report, 2015.

[8] Jean-Ian Boutin. *The evolution of webinjects*. ESET, `https://www.virusbtn.com/pdf/conference/vb2014/VB2014-Boutin.pdf`, 2014.

[9] Zheng Bu, Pedro Bueno, Rahul Kashyap, and Adam Wosotowsky. *The new era of botnets*. McAfee Labs, 2013.

[10] Armin Buescher, Felix Leder, and Thomas Siebert. *Banksafe information stealer detection inside the web browser*. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection (RAID), 2011.

[11] Claudio Criscione, Fabio Bosatelli, Stefano Zanero, and Federico Maggi. *Zarathustra: Extracting WebInject signatures from banking trojans*. In Proceedings of the Annual International Conference on Privacy, Security and Trust (PST), 2014.

[12] Stephen Doherty, Piotr Krysiuk, and Candid Wueest. *The state of financial trojans 2013*. Luettavissa: `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_state_of_financial_trojans_2013.pdf`, 2013.

[13] Nicolas Falliere and Eric Chien. *Zeus: King of the bots*. Symantec Security Response, 2009.

[14] Max Goncharov. *Russian underground 101*. Trend Micro Incorporated Research Paper, 2012.

[15] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J. Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. *Manufacturing compromise: the emergence of exploit-as-a-service*. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2012.

[16] Mario Heiderich, Tilman Frosch, and Thorsten Holz. *Iceshield: Detection and mitigation of malicious websitewith a frozen dom*. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection (RAID), 2011.

[17] Kaspersky Lab. *Financial cyber threats in 2013*. Kaspersky Lab Report, 2013.

[18] Kaspersky Lab. *Financial Cyberattacks Grow by Almost 16% in Q2 2016 as Malware Creators Join Forces*. `http://usa.kaspersky.com/about-us/press-center/press-releases/2016/Financialz_Cyberattacks_Grow_by_Almost_16_percent_in_Q2_2016_as_Malware_Creators_Join_Forces`, 2016.

[19] Loucif Kharouni. *Automating Online Banking Fraud*. Trend Micro Incorporated, 2012.

[20] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. *BareBox: efficient malware analysis on bare-metal*. In Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2011.

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

[22] Louis Marinos and Andreas Sfakianakis. *ENISA Threat Landscape*. ENISA, 2012.

[23] McAfee Labs. *Threats Report*. Technical report, `http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf`, 2016.

[24] Thomas Ormerod. *An Analysis of a Botnet Toolkit and a Framework for a Defamation Attack*. 2012.

[25] Ashkan Rahimian, Raha Ziarati, Stere Preda, and Mourad Debbabi. *On the Reverse Engineering of the Citadel Botnet*. Foundations and Practice of Security, 2014.
[26] Marco Riccardi, Roberto Di Pietro, and Jorge Aguila Vila. *Taming Zeus by leveraging its own crypto internals*. In eCrime Researchers Summit (eCrime), 2011.
[27] Aditya K Sood, Richard J Enbody, and Rohit Bansal. *Dissecting SpyEye - Understanding the design of third generation botnets*. Computer Networks, 2012.
[28] Symantec. *White Paper: Defend your institution against trojan-aided fraud*. 2011.
[29] Symantec. *Dyre: Emerging threat on financial fraud landscape*. 2015.
[30] James Wyke. *Breaking the bank(er): automated configuration data extraction for banking malware*. Sophos, `https://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/sophos-wyke-breaking-the-bank-VB2015.pdf`, 2015
[31] James Wyke. *Vawtrak – International Crimeware-as-a-Service*. Sophos, 2014.
[32] Candid Wueest. *The state of financial Trojans 2014*. Symantec, 2014.
[33] Candid Wueest. *Financial threats 2015*. Symantec, `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/financial-threats-2015.pdf`, 2015.