ZARATHUSTRA: Extracting WebInject Signatures from Banking Trojans

Claudio Criscione
Politecnico di Milano and Google Zurich
Email: claudioc@google.com

Fabio Bosatelli
Politecnico di Milano
Email: fabio.bosatelli@mail.polimi.it

Stefano Zanero and Federico Maggi
Politecnico di Milano
Email: stefano.zanero@polimi.it
federico.maggi@polimi.it

***Abstract*—Modern trojans are equipped with a functionality, called WebInject, that can be used to silently modify a web page on the infected end host. Given its flexibility, WebInject-based malware is becoming a popular information-stealing mechanism. In addition, the structured and well-organized malware-as-a-service model makes revenue out of customization kits, which in turns leads to high volumes of binary variants. Analysis approaches based on memory carving to extract the decrypted webinject.txt and config.bin files at runtime make the strong assumption that the malware will never change the way such files are handled internally, and therefore are not future proof by design. In addition, developers of sensitive web applications (e.g., online banking) have no tools that they can possibly use to even mitigate the effect of WebInjects.**

WebInject-based trojans insert client-side code (e.g., HTML, JavaScript) while the targeted web pages (e.g., online banking website, search engine) are rendered on the browser. This additional code will capture sensitive information entered by the victim (e.g., one-time passwords) or perform other nefarious actions (e.g., click fraud or search engine result poisoning). The visible effect of a WebInject is that a web page rendered on infected clients differs from the very same page rendered on clean machines. We leverage this key observation and propose an approach to automatically characterize the WebInject behavior. Ultimately, our system can be applied to analyze a sample automatically against a set of target websites, without requiring any manual action, or to generate fingerprints that are useful to determine whether a client is infected. Differently from the state of the art, our method works regardless of how the WebInject module is implemented and requires no reverse engineering.

We implemented and evaluated our approach against real-world, live online websites and a dataset of distinct variants of WebInject-based financial trojans. The results show that our approach correctly recognize known variants of WebInject-based malware with negligible false positives. Throughout the paper, we describe some use cases that describe how our method can be applied in practice.

I. INTRODUCTION

Information-stealing trojans allows a malware operator to intercept credentials such as usernames, passwords, and second factors of authentication (e.g., PINs or token-generated codes) or to alter how pages are rendered on the client side at their will (e.g., search engine result poisoning, click fraud). These trojans are also referred to as "banking trojans", because they are often used to steal banking credentials when the victim is using an online banking service. However, their flexibility made them easily adaptable to various uses. According to a recent Symantec report [9], in 2012 more than 600 institutions were targeted and a peak of more than 160,000 (October) of computers were compromised with financial trojans.

As we detail in Section II, the typical information stealers implement the interception mechanism through injection modules. An injection module, codenamed "WebInject", manipulates and inject arbitrary content into the data stream transmitted between an HTTP(S) server and the browser. This is implemented through function hooks placed between the rendering engine of the browser and the network-level libraries. Previous work [5] leveraged this observation to detect the hooking libraries as a sign of infection. As a result, WebInject-based trojans are able to circumvent any form of transmission encryption such as SSL. Moreover, a recent incident analysis reported by NASK [2] shows that customized variants of ZeuS are used to create an effective attack scheme involving both a PC and mobile component.

Nowadays, the common practice is that security researchers and professionals exchange samples, as soon as they become available, within private online communities. This makes it easy to obtain and run samples, resulting in quick reaction times, quicker than in the past. However, not all security analysts of targeted institution are equally equipped or skilled to perform accurate reverse engineering. Indeed, the analysis of these malware families, as well as others, require time-consuming reverse engineering, which result in slower reaction, even when samples are readily available. In fact, the detection rates of ZeuS are low. Another method used to extract the trojan configuration files is via memory forensics (e.g., by executing the sample in a sandboxed environment and extracting a memory dump for subsequent carving). The outcome of such analyses normally includes the decrypted webinject.txt file, which is useful for security analysts of targeted institutions, because it allows to verify if and how their website is targeted by an information-stealing campaign. Another interesting use case is the automatic analysis of samples that perform search engine result poisoning. Last, we notice that developers of sensitive web applications (e.g., online banking), possibly targeted by WebInject-based malware, are left with no tools that they can use to mitigate the effect of this threat. For instance, it would be great if a developer could programmatically "annotate" a page as "potentially targeted" to have an automatically-generated JavaScript procedure attached whenever the page is delivered to the client. Once rendered on the client page, such procedure would perform a sanity check to determine the presence of injections from known samples.

Unfortunately, the above mentioned techniques are based on the assumption that the malware will never change or alter

978-1-4799-3503-1/14/$31.00 ©2014 IEEE

the way configuration files are encrypted-decrypted in memory. This inherent limitation makes these methods not future proof, and shows the need for *automatic* methods that characterize the injection behavior of a malware, to tell whether an end host is infected by which known sample, or whether a given website is targeted by some known binary, *before* spending time to reverse engineer it.

The goal of our approach, called ZARATHUSTRA, is to automatically characterize the WebInject-based behaviors regardless of the underlying implementation. In addition, we want to isolate precisely the injected code, as if the configuration files of the malware variant were available. Our key observation is that, regardless of how the hooking mechanism works, the action of an injection module must eventually result in changes to the document object model (DOM). ZARATHUSTRA analyzes samples by first rendering a website page multiple times in instrumented browser instances that runs on distinct, clean machines. ZARATHUSTRA repeats the same procedure on an infected machine, and finally extracts the resulting, malicious differences in the form of an Xpath query along with metadata—which we call "fingerprints". A specific challenge that we tackle is the removal of legitimate DOM differences (e.g., due to ads, A-B testing, cookies, load balancing, anti-caching mechanisms). These differences would otherwise result in false positives. The fingerprint-generation system runs on dedicated machines with no interactions with real clients.

We evaluated ZARATHUSTRA against 213 real, live URLs of banking websites and 56 distinct samples of ZeuS. In all cases, our system generated fingerprints correctly. We analyzed the low fraction of false positives and found that most of them were caused by legitimate differences found in the original web pages, which are tackled by ZARATHUSTRA with specific post-processing heuristics, which can be safely enabled under realistic conditions, as detailed in Section V. ZARATHUSTRA scales well, and can process on average 1 URL in less than 3 seconds even on our limited infrastructure. Furthermore, as fingerprint generation can be performed independently on samples and URLs, the process is fully parallelizable and scalable.

As discussed in Section IV-E, the generality of the generated fingerprints make them suitable for various purposes, beyond malware analysis, that can help at mitigating the threat posed by WebInject-based malware. For example, we ZARATHUSTRA offered as a web service or programming API that, given a database of samples (which are abundant today) and a list of URLs, tells which URLs are targeted by which injection. Fingerprint matching is as fast as evaluating an Xpath query, which is trivial and supported by any XML-based client-side software.

In summary, in this paper we make the following contributions:

- We characterize the WebInject mechanism in an implementation-idependent, forward-looking fashion, without needing a-priori knowledge about the API hooking method, nor on the specific configuration encryption-decryption mechanisms used by the malware.
- We propose an approach to automatically generate fingerprints of the injections, requiring only the binary sample and the target URLs; as a matter of fact, we automatically generate the relevant information that would normally be available only by reversing and extracting the configuration file of the malware with a manual or non-future-proof process.
- We describe and discuss some case studies and how vendors can incorporate our approach in the browser-monitoring components of their antivirus products.

The source code of the ZARATHUSTRA proof of concept is available online at https://code.google.com/p/zarathustra/.

II. WEBINJECT-BASED TROJANS

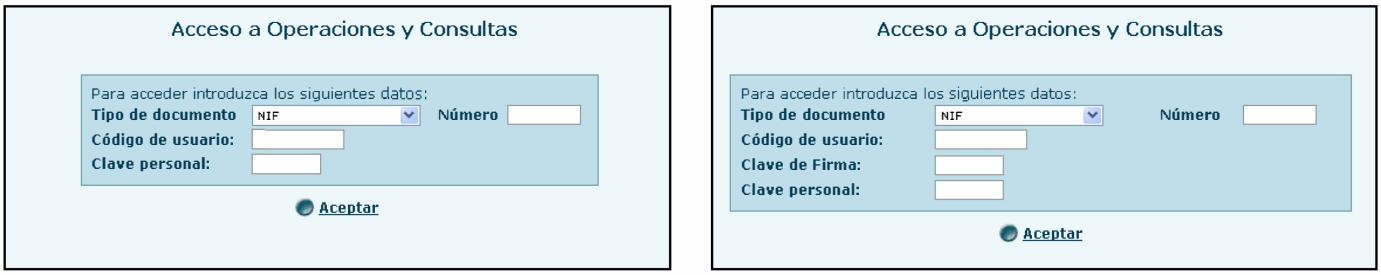
Information-stealing trojans are a growing [4, 11], sophisticated threat. The most famous example is ZeuS, from which other descendants were created. This malware is actually a binary generator, which eases the creation of customized variants. For instance, as of Feb 4, 2013, according to ZeuS Tracker[1], there are 7,457 distinct variants that are yet to be included to the Malware Hash Registry database[2] (these variants were 7,384% six months ago). Notice that this is an under estimate, limited to the binaries that are currently tracked. This high number of variants results in a low detection rate overall (39.17% as of Feb 4, 2013, decreased since six months ago).

State-of-the-art malware is very sophisticated and the development industry is quite mature. Trend Micro [1] reports a 29% increase of financial trojan activity between Q1 and Q2 of 2013 (from 37–39K to 71K infections, and from 113K to 146K targeted institutions worldwide). Lindorfer et al. [10] recently 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. Interestingly, cyber criminals also offer paid support and customization, or sell advanced configuration files that the end users can include in their custom builds, for instance to extract information and credentials of specific (banking) websites. Lindorfer et al. [10] 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 the malware. This also relates to the fact that the source code is sometimes leaked (e.g., CARBERP, ZeuS), which leads to further creation of new (banking trojan) variants [1] to keep up with the never-ending arms race.

A. WebInject Functionality

As part of their functionalities, modern trojans include data-injection and data-stealing capabilities. For instance, since version 1.0.0, SpyEye features a so-called "FormGrabber" module, which can be arbitrarily configured to intercept the data that the victim types into (legitimate) websites' forms.

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




```
set_url https://extranet.banesto.es/npage/OtrosLogin/LoginIBanesto.htm GP
data_before
name=usuario></td>
data_end
data_inject
</tr><tr>
</TR> <TR> <TD align=left><FONT size=+0><B>
Clave de Firma:</B></FONT></TD> <TD align=left colSpan=3><INPUT type=password maxLength=8 align=center size=8 value="" name=ESpass></TD>
data_end
data_after
data_end
```

Figure 1: Example of a real WebInject found on a page of `extranet.banesto.es`, performed by a ZeuS variant (MD5 15a4947383bf5cd6d6481d2bad82d3b6), along with the respective `webinject.txt` configuration file. Injections are not limited to this type of pages but include, for instance, search engine results.

The main goal of money-motivated criminals that rent or operate information-stealing services is to retrieve valid, full credentials from infected systems. In the case of online banking sites, these credentials comprise both the usual username and password, and a second factor of authentication such as a PIN or a token. This (one-time) authentication element is normally used only when performing money transfers or other sensitive operations. As a security measure, many banking websites use separate forms, and do not ask for login credentials along with the second factor of authentication. The goal of the attacker in this scenario is to lure the user into entering the token *up front*, together with username and password. This tactic gives the attacker enough time to use the token.

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

The WebInject module is leveraged to selectively insert the HTML or JavaScript code that is necessary to steal information or to make the targeted pages behave differently (e.g., click fraud, malicious advertising). WebInject allows to do this with surgical precision. For example, as shown in Figure 1, the WebInject module inserts an additional input field in the main login form of an online banking website. The goal is to lure the victim such that he or she believes that the web page is legitimately asking for the second factor of authentication up front. In fact, the victim will notice no suspicious signs (e.g., invalid SSL certificate or different URL) because the page is modified "on the fly" right before display, directly on the local workstation. Another nefarious action implemented through this type of functionality is search engine result poisoning or other forms of illicit content injection (e.g., to perform click fraud or click jacking).

WebInjects allow attackers to modify only the portion of page they need by means of site-specific content-injection rules. More precisely, the attackers can set two hooks (`data_before` and `data_after`) that identify the web page portion where the new content, defined with the `data_inject` variable, is injected. These variables are set at configuration time into a proper file, named `webinjects.txt` in the case of ZeuS, SpyEye, and descendants. Additionally, at runtime, the malware may poll the botnet command-and-control (C&C) server for further configuration options—including new injection rules.

The configuration files embody the actual value of an information stealer. Indeed, these files, and in particular `webinjects.txt` files, are traded[3] or sold[4] on underground marketplaces.

B. Library Hooking

The WebInject module of ZeuS and descendants relies on API hooking. Although distinct families such as ZeuS and SpyEye have a common WebInject module, new builds and other (future) families may implement WebInject differently. In addition, the malware binaries can be packed and obfuscated in various ways (e.g., different packing method or encryption key). Moreover, the custom configuration files are encrypted, and embedded in the final executable. This characteristic, combined with the evolving nature of modern trojans, makes it even more difficult to extract the static and dynamic configuration files—besides through time-consuming reverse-engineering efforts, or in the lucky case that the malware itself exposes some vulnerabilities (e.g., SQL injection, weak cryptography).

III. Goals, Approach and Challenges

The current "solution" against trojans is to use anti-viruses on the client side. Since the host is compromised, we are

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

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

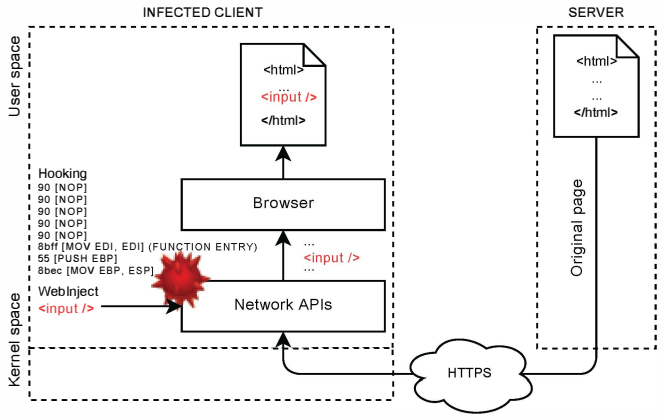


Figure 2: The HTML source code produced by the banking website transits encrypted over the Internet. When it reaches the OS and thus the `Wininet.dll` library, the source code is decrypted and intercepted. ZeuS modifies it on the fly and sends it through the same pipeline, up to the browser rendering engine.

well aware that client-side-only approaches are not an actual solution. There is no solution when the end host is not trusted. However, we believe that research should focus on *mitigation* approaches that (1) capture the inherent behavior of the targeted family (e.g., WebInject trojans) and, based on those behaviors, (2) speedup the generation of signatures. In the case of WebInject-based malware, the competitive advantage is that they exhibit their behavior in the browser. This makes solutions similar to the successful Google Safebrowsing feasible, with the added benefit of centralized deployments such as those described in Section IV-E.

To pursue our two goals, we believe that a good analysis approach should not rely too much on the *implementation* details of a malware. To this end, we observe the behavior of WebInject-based trojans (and other WebInject-based families) from the point of view of the browser. From hereinafter we use the term "WebInject" to refer to any mechanism used by malware to inject arbitrary content in the (decrypted) data that transits between the network layer and the rendering engine of a browser.

A. Approach Overview

Our approach is to fingerprint the behavior of any WebInject-based information stealer by looking for the visible effects of the injection in the targeted websites, regardless of the underlying implementation (e.g., API hooking, DLL patching, other yet unknown techniques). Our approach does not leverage any malware-specific component or vulnerability to observe and characterize the injection behavior. Therefore, it is more generic by design.

Our key observation is that a page rendered on an infected machine unavoidably includes the injected portions of code. In contrast, the same page rendered on a clean machine contains the original source code.

To automatically characterize the WebInject behavior of a given malware sample, our approach requires the malware sample executable and a list of target URLs. For example, in the generic case of an anti-virus company that wants to produce signatures for the top 1,000 online baking applications, the list of target URLs would contain the URLs of the respective websites. Another use case is the security officer of an organization, who receives a daily feed of malware samples and wants to automatically generate a signatures to quickly determine whether the organization website is targeted.

As output, our approach produces one Xpath expression per URL, which precisely identifies the portion of injected or changed code. For instance, for a given URL, the output looks like /html[1]/body[1]/center[3]/table[1]/tbody[1]/tr[1]/form[1]/input[13]. This is, per se, a valuable piece of information for the analyst. The simplicity of its output makes ZARATHUSTRA applicable to many different use cases. For instance, as part of a browser-monitoring component (e.g., based on matching the Xpath expression against the rendered DOM). In the remainder of this paper we focus on the details of characterization process, which is the core part of our contribution.

B. Challenges

Although our approach is conceptually simple when applied at a small scale (e.g., by manual analysis of a handful of target websites and samples, as shown in an example by Ormerod [12]), streamlining it and making it accurate is far from trivial. Indeed, websites may vary legitimately as a consequence of client- and server-side caching or upgrades of the (banking) web application code.

The problem of telling malicious and benign differences apart is hard to solve in general. In fact, a generic solution is beyond the scope of our research. However, in the well-defined case of an attacker that needs to inject at least one DOM node (e.g., `<script />`, `onclick=`, `1<input />`), we can address these challenges with specific heuristics as described in Section IV-B and IV-C.

IV. System Details and Implementation

The implementation of our approach in ZARATHUSTRA is summarized in Figure 3. The input of our approach is a list of URLs that we want to check and a sample. For each URL we repeat the following procedure. Since we want to eliminate false positives due to non-significant differences, we first need to make sure that each URL, if visited from two clean machines, do not exhibit any difference. If they do, then we need to ignore such benign difference, as not caused by the malware behavior. To this end, in the **DOM Collection** phase (Section IV-A), ZARATHUSTRA collects a set of DOMs from a set of identical clean (virtual) machines, and one DOM from the machine infected with the malicious executable. The DOMs are compared in the **DOM Comparison** phase (Section IV-B), which finds the differences between the "clean DOMs" and the "malicious DOM". In the **Fingerprint Generation** phase (Section IV-C), the differences are analyzed to eliminate obvious duplicates (e.g., due to legitimate changes or caching) and other recurring patterns of legitimate differences.

We implemented the **DOM Collection** phase on top of Oracle VirtualBox. We wrote a thin library on top of its API to create, snapshot, start-stop the VMs, and a library based

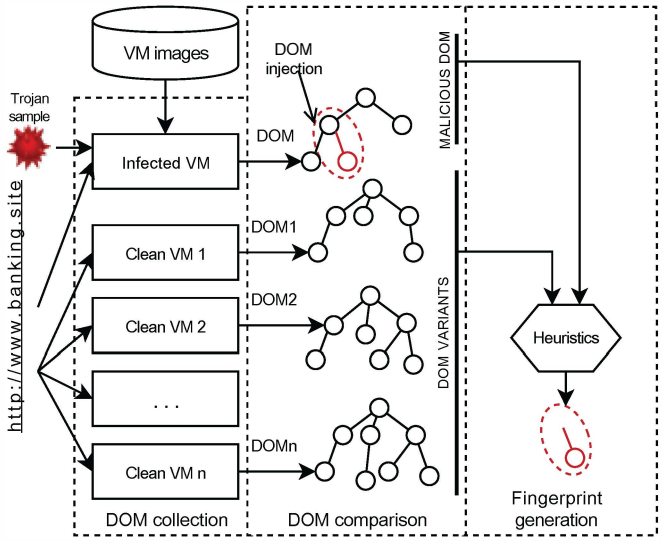


Figure 3: Server side architecture of ZARATHUSTRA, which is in charge of analyzing a given URL against a given trojan.

on WebDriver[5], 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 **DOM Comparison** relies on XMLUnit's `DetailedDiff` class functions. The **Fingerprint Generation** phase does not rely on 3rd-party libraries.

A. Phase 1: DOM Collection

This phase receives a target URL as input. It starts n clean VMs plus one infected VM. Each VM automatically starts the browser with WebDriver, visits the URL, lets the page load completely, and saves the resulting DOM. We then access and store the DOM as computed by the browser, thus including all the manipulations performed by client-side code at runtime while the page loads. The DOM encompasses the content of the nodes in the page, including script tags. This phase outputs the n "clean DOMs" that result from visiting the target URL with the clean VMs, plus one "malicious DOM" for the infected machine. The "malicious DOM" contains the injection that we want to extract.

B. Phase 2: DOM Comparison

This phase compares DOM, the "malicious DOM" against the "clean" $DOM_i \in [1, n]$ to find distinct differences. We rely on XMLUnit's `DetailedDiff.getAllDifferences()`, which walks the tree of DOM and, for each node, walks the tree of DOM_i to look for the following differences:

- **New node:** This catches one of the most common manifestations of information stealers (e.g., new `<input />` fields). This phase takes into account any element type (e.g., forms, scripts, iframes, text).
- **New attribute:** This reveals the presence of possibly malicious attributes such as the `onclick` event, used to bind JavaScript code that performs (malicious) actions whenever certain user-interface events occur.
- **Attribute value modification:** This catches manipulations on existing attribute values (e.g., to change the server that receives the data submitted with a form, or modifies the JavaScript code already associated to an action).
- **Text node content modification:** This occurs when a malware modifies the content of an existing node, for instance to add new code within a script tag, or to change the displayed text.

A pure removal of a DOM node (i..e, not followed by a node insertion) would be against the goals of the malware operator. Substitutions of DOM nodes are accounted for by ZARATHUSTRA as a modification (3rd and 4th case). The output of this phase is a set of DOM differences.

C. Phase 3: Fingerprint Generation

This phase prunes the DOM differences from the **DOM Comparison** and generates a set of fingerprints. First, we remove the differences between each couple DOM_i and DOM_j, $\forall i \neq j$ to take into account the legitimate changes between "clean DOMs", which could cause false positives. In other words, we obtain a pruned set of differences:

$$F = \underbrace{\text{diff}(DOM_i, DOM)}_{\text{all differences}} \setminus \underbrace{\text{diff}(DOM_i, DOM_j)}_{\text{benign differences}}$$

$$\forall i, j \in [1, n], i \neq j$$

where "diff(A, B)" indicates the distinct differences between DOM A and B, and the malicious differences are those obtained in **Phase 2**.

The rationale is that by visiting the same URL multiple times we obtain multiple versions of the same DOM, thus mitigating the effect of legitimate differences caused by session-sensitive content (e.g., caching, cookies). Furthermore, the heuristics 1–4 described in the remainder of this paper eliminate other legitimate differences.

The reader may wonder why, instead of comparing one infected vs. many clean DOMs, we do not compare many infected vs. many clean DOMs. Indeed, this would, in theory, create more variants of both the DOMs (i.e., benign vs. malicious), eliminating the benign differences more effectively. In practice, we would need to visit each URL from multiple *infected* virtual machines and compare the collected "malicious DOMs" against the "benign DOMs". This creates a potential performance problem. Moreover, with a pilot study we noticed that one "malicious DOM" per target URL is sufficient. In fact, visiting the same URL from multiple infected machines results in collecting DOMs that contain elements that are already present in the benign DOMs, so that will be eliminated. In simple words, increasing the number of malicious DOMs beyond one, only increases the required resources without adding any benefit.

[5]http://www.w3.org/TR/webdriver/

An example generated fingerprint for a given URL and sample is:

```
{
  "malicious_node": {
    "parent": "form",
    "value": "input",
    "xpath": "/html[1]/body[1]/table[3]/tbody[1]/tr[1]/form[1]/input[13]"
}
```

which specifies the `<input />` field injected in the real case of Figure 1. The set of fingerprints F already contains valuable information that precisely characterizes if and how an injection takes place. As F is generated in a fully automated way, it may still contain some false differences. These are addressed by two heuristics.

1) Heuristic 1: Ignoring Dynamic DOM Differences: We observe that several legitimate differences are actually due to DOM modifications performed by the browser that executes JavaScript routines while rendering the page. At a first glance, disabling JavaScript may lead to excluding malicious DOM modifications caused by the malware. However, WebInject malware always insert at least one static node or attribute, which would be still visible even when JavaScript is disabled. As we discuss in Section VI, even in the corner case of malware that injects code inside an existing `<script />` tag, by adding static code that performs the actual DOM manipulation, ZARATHUSTRA still generate a fingerprint of the static code injection in the first place.

2) Heuristic 2: Caching Server Responses: By caching server responses—using the URL as the caching key—we reduce the false differences due to dynamic code generation on the server side, which may insert, for instance, a unique element in each response (e.g., to avoid cross-site request forgery or caching). This simple heuristic can be safely enabled and, as showed in Section V-D, helps at effectively reducing the false positives.

D. Post-processing Heuristics

The output of the previous phase is the set F of fingerprints, which is post processed through the following two heuristics that minimize the occurrence of false differences.

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

2) Heuristic 4: Filtering Text Nodes: Text nodes are harmless, because they can only contain pure text. We ignore all the text nodes, unless they are children of `<script />` tags. There are many other ways through which a malware can insert custom client-side code, but ZARATHUSTRA already accounts for these types of WebInjects during **DOM Comparison**.

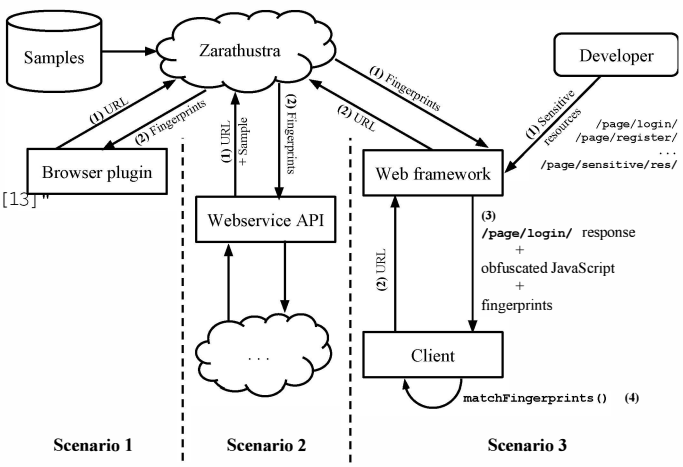


Figure 4: Three application scenarios described in Section IV-E.

E. Application Scenarios

ZARATHUSTRA produces fingerprints in the form of Xpath expressions, which allow to check, on the client side, whether a web page is currently being rendered on an infected machine. The most natural application is the antivirus scenario. In practice, as depicted in Figure IV-E, we foresee a centralized server, which runs **Phase 1–3** on a feed of malware samples and URLs.

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

The second scenario that we envision, **Scenario 2**, has to do with large-scale monitoring. More precisely, we believe that ZARATHUSTRA 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 of malware samples and URLs to analyze. In this context, ZARATHUSTRA 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**) is that of a developer of high-profile web applications (e.g., online banking), which are likely to be targeted by WebInject-based malware once in production. This scenario was suggested to us by a developer working for a large national bank, who noticed the lack of a centralized solution to determine whether their clients where infected with some banking trojan. In this context, the developer would like the web framework to offer an API to programmatically mark sensitive resources (e.g., /page/login/, or those that contain forms) as such.

Marked resources will be processed by the web framework right before the HTTP response is sent to the requesting client. In our vision, the web framework will append a Javascript

procedure that, once executed on the client, performs a check similar to the one described in the aforementioned "Safebrowsing" scenario. The JavaScript can be obfuscated and inserted in randomized positions, so as to make it difficult for the malware author to selectively remove it. The trojan on the client side may still attempt to disable that JavaScript procedure. Note that disabling Javascript completely is against the attacker goal, because it would disrupt the page.

V. Experimental Evaluation

Between January and February 2013 we evaluated our implementation of ZARATHUSTRA against 213 real, live URLs of banking websites and 56 distinct samples of ZeuS (see Appendix). Our main goal was to measure the correctness of the fingerprints, generated with and without the heuristics. Then, we wanted to assess the resource required to analyze a given amount of URLs and samples.

A. Dataset

With the above premises, our decision fell on ZeuS, because it is by far the most widespread information stealer that performs injections: According to ZeuS Tracker, as of Feb 4, 2013 there are 556 known C&C servers (242 active), and an alarmingly low estimated antivirus detection rate (39.17%, zero for the most popular and recent samples). We also conducted a series of explorative experiments with SpyEye, which is less monitored than ZeuS (184% C&C servers, 69 active, and an average detection rate of 27.94%); thus, it is more difficult to obtain an ample set of recent samples. However, SpyEye uses the same WebInject module of ZeuS, as described by Binsalleeh et al. [3], Buescher et al. [5], Sood et al. [14]. For these reasons, for the purpose of evaluating the quality of our fingerprint-generation approach, we decided on ZeuS as the most representative information stealer that generated real-world injections. We downloaded 76 samples, but 20 of these failed to install or crashed, leaving 56 distinct samples.

We constructed a list of target URLs by merging 2 `webinjects.txt` files found on underground forums, plus the `webinjects.txt` leaked as part of ZeuS 2.0.8.9 source code[6]. We so obtained 293 distinct URLs. To make it feasible to manually verify the results in a reasonable time, we selected 213 URLs (143 organizations) among the URLs that were active at the time of evaluation. Building a list of URLs from `webinjects.txt` files found in the wild allowed us to deal with real-world targeted pages. As WebInjects occur on landing or login pages, we concentrated our search on such pages.

We created the ground truth by configuring ZARATHUSTRA with all the heuristics enabled. We then manually analyzed the results to ensure that no false or negatives were found: By design, if there is an injection, ZARATHUSTRA detects it. So, false negatives are not an issue. An alternative approach could have been to craft a proper `webinjects.txt` as the ground truth. However, we wanted to test ZARATHUSTRA on real injections found in the wild.

[6]https://bitbucket.org/davaeron/zeus/

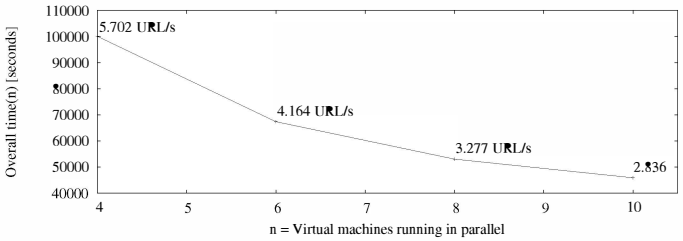


Figure 5: Scalability of ZARATHUSTRA: Time required to process 213 URLs with 76 samples (including crashing samples). the labeled points indicate the time to process 1 URL.

B. Setup and Scalability

We run all our experiments on a 1.6GHz, 4-cores x86-64 Intel machine with 16GB of RAM. We installed Windows XP SP3 (Internet Explorer 6, 7, 8) on each VM and granted outbound and inbound Internet access. ZARATHUSTRA required 256MB of RAM and 2 to 5GBs of disk space per VM.

In our experiments we run **Phase 1 (DOM Collection)** with 2 to 35 VMs to collect the clean DOMs. Figure 5 shows that ZARATHUSTRA scales well overall: With 10 VMs running in parallel we are able to process 1 URL in less than 3 seconds. The architecture of ZARATHUSTRA has no central node, nor any dependency that prevent full parallel operations: As a result, its capacity scales directly with the amount of resources available.

C. Correctness

Table I summarizes the top-ten domains where ZARATHUSTRA correctly recognized injections caused by ZeuS. Some samples perform zero injections, although usually we found around 1 to 9 injections per distinct URL of the same domain.

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

D. False Positives

False positives occur when ZARATHUSTRA confuses benign differences as injections. On the data collected during the experiment described in Section V-C, we obtained zero false positives when using all the heuristics.

In a more detailed analysis, we concentrated on the influence of **Heuristic 1** versus the use of multiple clean VMs.

Table I: Top ten target websites in our dataset. The minimum, maximum, total and average number of injections are calculated over the set of ZeuS 56 samples, and on the URLs within each domain.

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

The rationale is that this heuristic was the most effective at eliminating false positives, as the first row of Table II shows. Thus, after disabling **Heuristic 1** we run ZARATHUSTRA with an increasing number $n \in [1, 35]$ of clean VMs. In this way, we can assess how well ZARATHUSTRA can tell legitimate differences and true positives apart when using a sufficiently large number of emulated clean clients in **Phase 1**.

As Figure 6 shows, without **Heuristic 1** the false positives can still be mitigated by increasing n. The false positive rate approaches almost zero (1%) if at least 35 clean VMs are used. We manually observed that the vast majority of that 1%, at $n = 35$, was caused by JavaScript-based advertisement networks and modifications performed by the browser, which lead to highly-dynamic DOMs. Thus, when deploying ZARATHUSTRA on URLs that have a dynamically-generated DOM, it is recommended that either **Heuristic 1** is enabled, or a large number of VMs is used to create a robust "baseline".

VI. DISCUSSION AND LIMITATIONS

From our experiments we can conclude that ZARATHUSTRA can reach zero false positives when all the heuristics are enabled—or with a decent number of clean VMs in their stead—and has zero false negatives by design. State-of-the-art approaches (e.g., via reverse engineering) may reach have zero false positives. However, considered the time required to generate signatures with these methods, the price is that of missed

HEURISTICS	AVG. CORRECT (± VAR.)	%URLS
2,3,4	**39.58 ± 11.53%**	52.17%
1,3,4	74.98 ± 15.42%	23.48%
1,2,4	97.97 ± 0.069%	22.61%
1,2,3	98.42 ± 0.124%	23.04%
All	100.0%	23.48%

Table II: Contribution of each heuristic on the quality of the fingerprints. The second column reports the fraction of URLs with correctly-identified injections (this fraction is averaged over the set of 56 samples). The last column reports the fraction of URLs where at least one sample was detected while performing an injection, including false differences, which are analyzed separately in Section V-D.

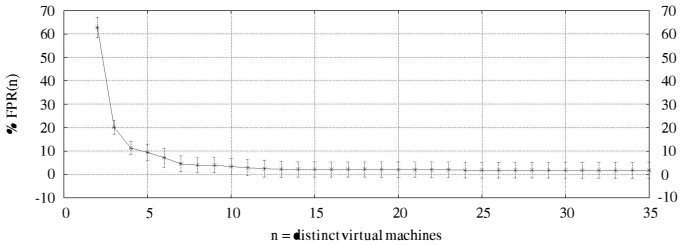


Figure 6: False positives due to legitimate differences decrease for an increasing number, $n \in [2, 35]$, of clean VMs, until it reaches 1.0%. With **Heuristic 1** enabled, we achieve zero false positives.

injections (i.e., false negatives). ZARATHUSTRA, instead, analyzes WebInject-based malware automatically, quicker than a reverse-engineering-based approach, and with the same requirement.

Our second discussion point is that malware operators could rewrite the injected code, introducing no-op DOM nodes with the goal of evading the fingerprints generated by ZARATHUSTRA: Adding an additional `<div />` wrapper to a page (in a random position), for instance, would circumvent a naïve use of our fingerprints (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, and more importantly, modifying the structure of a page can easily result in user-visible, brittle modifications. This is clearly against the malware operators' goal of preserving the look of the page as much as possible. Although we leave the implementation of a proper fingerprints matching algorithm to future work, we are aware that there is an accuracy trade off between matching the entire XPath expression of a fingerprint versus matching only the leaf nodes. However, if the leaf nodes are detailed enough (e.g., they contain attributes), an algorithm that matches only the leaf nodes can achieve good accuracy and high generality.

WebInjects are the only artifacts that we rely on to observe the action of an information stealer. As a result, if a banking trojan succeeds in hiding its behavior (e.g., by injecting content only under certain conditions), ZARATHUSTRA cannot guarantee to extract differences every time a targeted URL is visited. This discussion point relates to malware that adopt anti-emulation techniques. However, we rely on virtual machines solely for ease of implementation and flexibility during evaluation. ZARATHUSTRA works perfectly, and even faster, on bare metal. Hence, this obstacle is easily circumvented by adopting the method proposed by Kirat et al. [8] to obtain virtual-machine-equivalent snapshots on physical hardware. In this way, no malware can possibly recognize that it is running in a controlled environment.

Last, a minor point of our current implementation of ZARATHUSTRA is that we take the (banking) website as an oracle. For reasons that fall outside our attacker model (e.g., client-side malware), an injection may match exactly with a benign difference. For example, this happens if the 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 their site is updated, or

possibly by requesting an update of the fingerprints for that domain. It is indeed reasonable to envision ZARATHUSTRA deployed within a bank information system: This use case would avoid most, if not all, the venues for false positives as a fully up-to-date model of the clean website would always be available. Similarly, ZARATHUSTRA can easily monitor authenticated web pages, which are not a limitation when our system is deployed by the website provider (e.g., bank).

VII. Related Work

Trojans have been studied in the past two years. Sood et al. [14] give a detailed overview of the components of SpyEye, including its development kit, and describe how SpyEye integrates in the whole criminal ecosystem. Binsalleeh et al. [3] performed a similar study on the ZeuS crimeware toolkit.

The key intuition of Buescher et al. [5] is that WebInjects are currently implemented by hooking into the Windows API functions: Since version 2, ZeuS hooks into `Wininet.dll`, used by Microsoft IE to (e.g., `HttpSendRequestA`, `InternetReadFile`). The authors analyzed all the possible hooking mechanisms that could be implemented in the Windows OS (i.e., inline hooks, import address table hooks, export address table hooks, and hooking techniques that manipulate the windows loader mechanism) and, from them, they derived behavioral fingerprints. In practice, they look for extra code sections in the basic Windows libraries, by comparing the version stored on disk against the version loaded in the process memory. Extra code sections are a sign of hooking.

Our approach is similar in spirit to [5], which however focuses on the binary libraries loaded in memory. Instead of focusing on the specific DLL hooking mechanism and functions adopted, we concentrate on the manifestation of such hooking in the browser.

Also [6] is related to our work, since it protects the browser from malicious websites that perform dynamic changes of the DOM. Although not designed specifically to target information stealers, it could be applied to recognize WebInjects. The system instruments the ECMA script layer by proxying its functions so to profile their execution and recognize malicious patterns. However, the authors mention that their method can detect changes of the DOM that occur at runtime, whereas WebInjects work at the source-code level.

Along a different line, Riccardi et al. [13] developed a chosen-plaintext attack against the encrypted stream that flows between ZeuS (1.x and 2.x) and its C&C. The chosen plaintext is a combination of the information from the analysis of the malware toolkit and the data collected while running a sample in a controlled environment (e.g., cookies, user credentials, or computer hostname). These attacks are effective against a specific version of a malware binary. Unfortunately, they require the reverse engineering of the malware.

VIII. Future Work

Besides addressing the discussion points described in Section VI, future research should concentrate on more advanced uses of WebInjects.

As described in Kharouni [7] targeted attacks 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. This threat will require modifications to ZARATHUSTRA, because the injections may occur in pure text nodes. Thus, the set of heuristics will need to be refined to cope with these corner cases.

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.

IX. Conclusions

In this paper we have presented ZARATHUSTRA, an automated system to observe the client side behavior of financial trojans that perform WebInjects. ZARATHUSTRA generates fingerprints of the DOM differences by comparing web pages as they are rendered in an instrumented browser running on clean and infected virtual machines. The first advantage of our system is that of requiring no reverse-engineering effort. Moreover, our approach is future proof by design.

Our evaluation of ZARATHUSTRA against 213 real, live URLs of banking websites and 56 distinct samples of ZeuS show that, in all the cases, our system extracted all the injections correctly. The low rate of false positives (1.0%) were caused by legitimate differences in the original web pages. We have developed specific heuristics, which can reduce such false positives to zero. ZARATHUSTRA scales well, and can generate fingerprints for 1 URL in less than 3 seconds on average, even on our modest infrastructure.

Although simple, our approach has the great advantage of being completely agnostic with respect to the source of the differences: As long as the manipulated data is observable, our approach can be generalized to create further "difference modeling" techniques that can be used to characterize the activity of an information stealer from other observation points.

Acknowledgements

This research was supported by the FP7 project SysSec funded by the EU Commission under grant agreement no 257007, and TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research.

References

[1] Mobile Threats Go Full Throttle. Technical report, Trend Micro Incorporated, 2013.

[2] A powerzeus incident case study. Technical report, NASK - CERT Polska, October 2013.

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

[4] Zheng Bu, Pedro Bueno, Rahul Kashyap, and AdamÂ Wosotowsky. The new era of botnets. Technical report, McAfee Labs, Jan 2013.

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

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

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

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

[9] Piotr Krysiuk and Stephen Doherty. The World of Financial Trojans. Technical report, Symantec Corporation, 2013.

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

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

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

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

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

APPENDIX

FAMILY	MD5	DETECTED INJECTIONS
ZeuS	68ab93087e2bf697e48b912b4546e666	0/0
ZeuS	93895e081e679f8d9760de48b4ad349f	17/17
ZeuS	757f4dcb8fb34e8d168e632f16cebd53	13/13
ZeuS	1a45e46567b84d38ba868f702e795913	4/4
ZeuS	fd622057a281813c32cade7ad54843a5	12/12
ZeuS	9cd8fbd475c088d860bdc1371924dd4f	13/13
ZeuS	9ffe865c925bf06d35aa6b68cdaf3609	0/0
ZeuS	85719c933ccdb42f37e8c4d9b5e6bcfd	0/0
ZeuS	2105082b794ecfa02136e012f5ab4e6b	0/0
ZeuS	15a4947383bf5cd6d6481d2bad82d3b6	13/13
ZeuS	b2a52dabdc8134199cd7858dd8e41013	17/17
ZeuS	b68d88be4d65b29ad17937d8a419d8ba	0/0
ZeuS	bb0c5a0c13682b996f5ab4b5dd79f430	17/17
ZeuS	254712088ab8e08619f20705d7a09cf1	0/0
ZeuS	6ba342b44509215 1d8171a62efe633cb	17/17
ZeuS	71d1a97b5776f3adc7f92ba6e82d162b	13/13
ZeuS	b82eeaf8d5c0ed3d44269196865beb80	13/13
ZeuS	21ef35e6e3f3494d134e9928ca6f38e8	17/17
ZeuS	e54d1b119211907dad7dc33ff087d5be	13/13
ZeuS	56f8a7c7721aa96e543d57b0fef0f98f	0/0
ZeuS	2a12ba5847c0fb58a89ea6b2f6dd1a97	0/0
ZeuS	1ad8e54179e8c2c7767ea3b039d542fc	2/2
ZeuS	9b9951c50e04818c413c8cd1a3096a6b	0/0
ZeuS	d60487f05000160d85db0b354dbdd866	16/16
ZeuS	cdf3bb9c75000fc49c7c148b76c20b45	17/17
ZeuS	31ea03a2a33a75ddf48d52f4605ef0bb	16/16
ZeuS	b1a49aa03fc1a8226ebc1205bdcf5562	13/13
ZeuS	6384e4f1b5eeefbcb99a281ac514078a	0/0
ZeuS	4df1446e8419978a0999ff2fa3fd60a3	17/17
ZeuS	041c17a7b97550fd69d25613d9ef8f46	0/0
ZeuS	9bc0e3d19af915c608a784fda63b0076	13/13
ZeuS	a4aa162745adcb84373e6a623125c650	12/12
ZeuS	22788996e2381bdb97480b8de141ec2c	0/0
ZeuS	5e26d372feb7d085b752fffa931fc156	0/0
ZeuS	39ad78a889a2b40a94dd700d67f1a5ed	2/2
ZeuS	b2c82ffe10763cdc241c7fa8d97097ae	13/13
ZeuS	bf45f27a403acfd3847fbbae88a8375f	0/0
ZeuS	9abaffda80841aa87c9f5786e0db639e	0/0
ZeuS	029d4f8dcf43837f773116439b07e980	1/1
ZeuS	08e01221186cf82952c25d995176561b	0/0
ZeuS	6436032a3d5bf53c6273ddd0ffab80be	40/40
ZeuS	fd12f0d2e2bbef953ac87d4dca32c15d	0/0
ZeuS	3ba3149213e6b9091c727104dbb26ea6	41/41
ZeuS	b62dbd301f130487dfbc1473dced8aad	17/17
ZeuS	f75e3fa05762072e5e6471f3fb982087	13/13
ZeuS	c04fddfaab6b879a25b036980a34908e	12/12
ZeuS	ffcaf8a2f2f59e0f7b165d085842bd17	16/16
ZeuS	70dfde201f6a9a66730d9ae6b69450f8	42/42
ZeuS	ecc0a5bdf5174efcd9d292e815de06f4	11/11
ZeuS	5298f1fd6b300223f6bcdbc1fa89c2c0	0/0
ZeuS	7f280b73093e5b61ab2eec7b6ebda420	17/17
ZeuS	21248f3752c84ee5866a95992dba0813	17/17
ZeuS	51eef801f614a0278c8b79f7be9d2fdf	12/12
ZeuS	be4f416d394b4e305fd0e11d40a4242c	17/17
ZeuS	99646549006435d73efeddbbbcf4313f	13/13
ZeuS	c4ba4d84e5b40132e82b403469eb13ca	0/0