2014 IEEE Symposium on Security and Privacy

Hunting the Red Fox Online: Understanding and Detection of Mass Redirect-Script Injections

Zhou Li[†‡], Sumayah Alrwais[†], XiaoFeng Wang[†], Eihal Alowaisheq[†]

[†]Indiana University at Bloomington
[‡]RSA Laboratories

***Abstract*—Compromised websites that redirect web traffic to malicious hosts play a critical role in organized web crimes, serving as doorways to all kinds of malicious web activities (e.g., drive-by downloads, phishing etc.). They are also among the most elusive components of a malicious web infrastructure and extremely difficult to hunt down, due to the simplicity of redirect operations, which also happen on legitimate sites, and extensive use of cloaking techniques. Making the detection even more challenging is the recent trend of injecting redirect scripts into JavaScript (JS) files, as those files are not indexed by search engines and their infections are therefore more difficult to catch. In our research, we look at the problem from a unique angle: the adversary's strategy and constraints for deploying redirect scripts quickly and stealthily. Specifically, we found that such scripts are often blindly injected into both JS and HTML files for a rapid deployment, changes to the infected JS files are often made minimum to evade detection and also many JS files are actually JS libraries (JS-libs) whose uninfected versions are publicly available. Based upon those observations, we developed JsRED, a new technique for the automatic detection of unknown redirect-script injections. Our approach analyzes the difference between a suspicious JS-lib file and its clean counterpart to identify malicious redirect scripts and further searches for similar scripts in other JS and HTML files. This simple, lightweight approach is found to work effectively against redirect injection campaigns: our evaluation shows that JsRED captured most of compromised websites with almost no false positives, significantly outperforming a commercial detection service in terms of finding unknown JS infections. Based upon the compromised websites reported by JsRED, we further conducted a measurement study that reveals interesting features of redirect payloads and a new Peer-to-Peer network the adversary constructed to evade detection.**

I. Introduction

For years, the Internet community has been haunted by increasingly sophisticated and organized cybercrimes, ranging from exploits on vulnerable systems (e.g., drive-by downloads) to all kinds of frauds and social engineering. Such criminal activities have developed into mass underground businesses, costing the world hundreds of billions of dollars every year and victimizing hundreds of millions of Internet users [44]. Crucial to their operations is the existence of a large number of vulnerable websites, which can be easily compromised on a large scale and converted into web redirectors. These redirectors serve as doorways for a complicated web infrastructure that delivers malicious payloads to victims [42], playing a critical role in hiding more expensive criminal assets (e.g., exploit servers) in the shadow.

The "Red Fox". Timely detection and recovery of those compromised websites deprives cybercriminals of their major resources for luring visitors, and can potentially disrupt this portion of the underground business. Development of effective techniques for this purpose, however, is a daunting tasks in fighting cybercrimes. Different from the web servers hosting malicious content such as exploit kits, those redirectors are just ordinary websites with a few injected redirect scripts, which can also appear on legitimate sites. Existing ways to detect them rely on tracking a redirection chain that ultimately hits malicious content providers [27], a process that is often interrupted by cloaking [47]. Further complicating this effort is the trend that the criminals increasingly place their redirect scripts within JavaScript (JS) files on a compromised site, which are different from other web documents like HTML, are not indexed by Google and other search engines [15], and thus the infections on them are even more difficult to find. Most importantly, those redirectors are easy to collect and often expendable to the attackers, rendering any heavyweight detection technique hard to catch up with the pace that new sites are recruited. As a result, even though progress continues to be made in analyzing and detecting malicious content hosts [27], compromised web redirectors remain to be an elusive "red fox" difficult to hunt down.

In our research, we looked at the problem from a new perspective - the strategies those criminals adopt to inject redirect scripts, which underline the constraints they face. Through inspecting 436,869 infected files collected recently, we found that the "red fox" indeed has several unique, previously unknown features. In particular, to deploy his redirect scripts quickly, the attacker tends to inject them *blindly* into various files (JS files as well as HTMLs) on a compromised site. This needs to be done carefully, avoiding any interference with the website's normal operations to hide the presence of the malicious code. Also, a significant portion of infected JS files are public JS libraries (JS-libs), due to their dominant presence on legitimate websites (at least 60% web sites use JS-libs [46]), which web developers either do not change at all or modify in a way completely different from what the attacker does.

Detection and findings. Leveraging those unique features, we developed a new, lightweight technique for catching this red fox. Our solution, called *JsRED*, is designed to automatically detect *unknown* redirect scripts on a large scale. Our idea is based upon the observation that in a mass redirect-injection campaign, similar scripts are blindly inserted into JS-libs, other JS files, HTMLs, etc. on compromised web servers. Among them, the clean versions of the JS-libs are publicly available,

DOI 10.1109/SP.2014.8

often unchanged by the website developer or customized by adjusting just a few parameters. Therefore, we can compare a JS-lib file (e.g., jQuery [38]) crawled from a website with its clean references[1] to extract the difference, and further analyze it statically and dynamically to determine whether the difference is actually a redirect script. Given the fact that it is extremely rare in a legitimate customization of a JS-lib to add just redirect code, a script identified this way is almost certain to be malicious. With the blind injection strategy the attacker takes to make his campaign scalable, the detected code can then be generalized into a template for scanning non-lib JS files, HTMLs and other content to catch their infections. In this way, we can identify infected websites on a large scale, even when their infections have never been seen before.

We implemented JsRED and evaluated it over 1,129,988 JS and HTML files we collected. The new approach was found to be highly effective: it outperformed Microsoft Security Essentials [31], a commercial malware detection service, by nearly 100% in terms of detected JS-file infections, and did not result in any false positives on data collected recently over three months. The approach has also achieved high performance and is capable of analyzing 255,082 JS files to generate signatures within one day and scanning all 1,129,988 files in only two hours using the signatures, on a single desktop machine. We further conducted a measurement study on the infected JS and HTML files JsRED detected, which reveals the attacker's strategies such as the effort they made to conceal their redirect scripts. Of particular interest is the discovery of a structured peer-to-peer (P2P) redirection network built entirely on compromised sites: a visitor to any of such compromised doorways will be redirected to other sites, which are also compromised legitimate websites, before being forwarded to attack hosts. This network provides further cover for the web redirectors and we studied this network and report its unique features like dynamic selection of redirect targets, cloaking strategies and long life time (over 285 days) in Section V-B.

Contributions. We summarize the contributions of the paper as follows:

- *New technique*. We developed a lightweight yet effective technique for fully-automated detection of unknown redirect scripts. Our approach leverages new observations of the attacker's strategy and exploits his limitations, identifying new redirect code through a simple differential analysis. Our study shows that the technique works particularly well on infected JS files, outperforms commercial tools and incurs almost no false positive.

- *New discoveries*. We performed an in-depth measurement study on compromised web redirectors. Our study helps better understand how the attacker covers injected code and deploys it on a large scale. We also looked into the P2P redirect networks we discovered, which is a new strategy the attacker utilizes to protect compromised doorways.

Roadmap. The rest of the paper is organized as follows: Section II presents the background information about the redirect-injection attack and its unique features we observed; Section III elaborates the design, implementation and evaluation of JsRED; Section V reports the findings of a measurement study on the compromised websites caught by JsRED; Section VI discusses the limitations of our technique and potential future research; Section VII reviews related prior research and Section VIII concludes the whole paper.

II. Mass Redirect-Script Injections

A. Background and Adversary Model

As discussed before, compromising a large number of vulnerable websites is the adversary's dominant strategy to find potential victims and deliver malicious web content. As evidence, WebSense recently reports that 85% of malicious links detected this year have been found on compromised hosts [49]. On these hosts, web pages were altered to either directly serve malicious content, which attacks visitors through drive-by downloads or phishing, or redirect the visitors to other hosts set up by the adversary. By comparison, the redirection approach is much stealthier, as it does not directly expose the web content involving attack vectors (e.g., exploiting a vulnerability within the visitor's browser) and thus is less likely to be identified. Indeed, according to a recent study by Sucuri [42], over 70% of such compromised sites are used as redirectors, referring their visitors to other compromised or malicious servers. The redirections here are performed through injecting HTML tags like <iframe> to HTML files or redirect scripts into JS or HTML files. Such scripts later set the client's browser location or dynamically create HTML tags to cause malicious web content to be downloaded to the browser. Given the fact that JS files are increasingly utilized by websites and not indexed by search engines (which make it harder to locate them in the first place), they are becoming a popular targets for the redirect-code injections. We focus on this new type of threats.

Redirect-script injection. Here we show how such attacks work using an example in Figure 1, which illustrates a redirect-script injection campaign [43]. After compromising a vulnerable host and acquiring its root access, the adversary uploads an encoded redirect script to `/usr/share/php/a.control.bin` on the host and modifies its system file `/etc/httpd/conf.d/php.conf`, which configures all the PHP applications on the Apache server, to intercept requests delivered to the server. As a result, the adversary gets a chance to inject the script each time when the server responds to a request. Specifically, whenever a client asks for a JS file from the compromised site, the redirect script kept under `/usr/share/php/a.control.bin` is decoded and attached to the original JS file, and then executed within the browser. The injected code, once executed together with the JS file within the client's browser, creates hidden iframes within an HTML file to redirect the browser, which finally reaches the exploit server installing *RedKit* [19], a popular exploit kit for drive-by downloads.

Detecting this type of attacks on a large scale is challenging. On the one hand, through port scanning or searching with Google dorks [29], the adversary can easily discover hundreds of thousands of vulnerable web servers and quickly turn them into web redirectors using automatic tools [12]. Therefore, an effective control on the injection campaign cannot rely on

[1] There can be many versions for a specific library: e.g., jQuery has 38 versions.

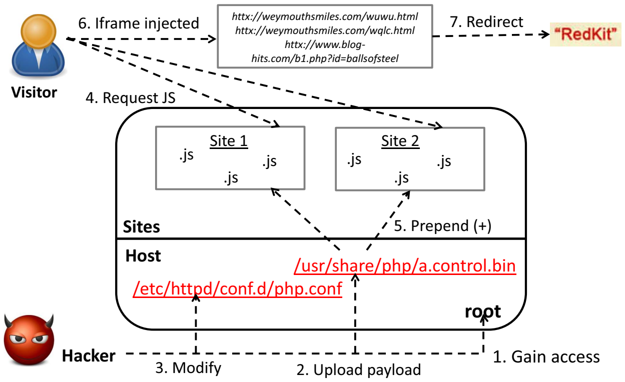

6. Iframe injected
httx://weymouthsmiles.com/wuwu.html
httx://weymouthsmiles.com/wqlc.html
httx://www.blog-hits.com/b1.php?id=ballsofsteel
7. Redirect
"RedKit"
Visitor
4. Request JS
Site 1
.js
.js
.js
Site 2
.js
.js
.js
Sites
5. Prepend (+)
Host
/usr/share/php/a.control.bin
/etc/httpd/conf.d/php.conf
root
Hacker
3. Modify
2. Upload payload
1. Gain access

Fig. 1. An example of malicious campaign that compromises web servers and injects redirect scripts.

heavyweight, slow approaches such as tracking down redirection chains to expose exploit servers, which often requires a lot of effort when infected or malicious hosts cloak, moving onto the next hop on the chain only if certain conditions (e.g., "IE only") are satisfied. On the other hand, the parties that perform the detection, such as Google, typically do not have access to the server-side code of hosts and have to rely on the content they export to the client to find out infections. Such information can be scant, as the redirection code on the client side can be very simple when cloaking is performed on the server side and also popular on clean, legitimate websites. Actually, without a well-thought-out approach, it can be difficult to detect such infected redirectors even manually (see the example in Figure 2), not to mention extracting their signatures for an efficient online scan.

```
var temp = ””,
        i, c = 0,
        out = ””;
var str = ”60!102!...116!62!”;
l = str.length;
while (c <= str.length − 1) {
    while (str.charAt(c) != '!')
                temp = temp + str.charAt(c++);
    c++;
    out = out + String.fromCharCode(temp);
    temp = ””;
}
document.write(out);
```

(a) A redirect script discovered in the wild.

```
document.write(
        '<frameset rows=”100%” cols=”100%”>
        <frame src=”http://b−muj.ru/?k=3258” noresize/>
        </frameset>'
        );
```

(b) The redirect script after deobfuscation.

Fig. 2. Redirect script sample.

As an example, let us look at a redirect script in Figure 2, which we discovered in our research. The script was injected into HTML pages and redirected a visitor's browser to `http://b-muj.ru/?k=3258` through `frame` tag, a web page that was malicious at the time we crawled. The original code was extensively obfuscated. After deobfuscation, we found that the script includes a single JS API `document.write`, which is very common and frequently used by clean, legitimate web sites. In this case, even a manual inspection of the code (without further analyzing the redirection target) may not be able to conclusively identify the infection. To address this problem, we came up with a different approach based upon unique features of the threats, as elaborated in Section II-B and Section III.

Adversary model. We consider an adversary who intends to extensively deploy his redirect scripts to a lot of vulnerable hosts within a short period of time and also wants such code to operate stealthily, without undermining the functionalities of the original websites. This is exactly what a real-world attacker does. To attain these goals, the adversary has to work under some constraints. Particularly, he cannot deliberately avoid JS libs, as they are used by the majority of web sites (above 60%). He cannot even modify the name of a JS lib, whose references are scattered across the whole website. Therefore, any name change will force the adversary to touch many files and run the risk of being caught or disrupting the way the website works.

B. *Features*

To find a better way to detect redirectors and thwart a large-scale injection campaign, we need an in-depth understanding of the attack to identify its weak spots. To this end, we analyzed a large number of web files (both malicious and legitimate) crawled from the web. Our study brought to light a set of interesting features that uniquely characterize the attack, including (1) the adversary's blind injection strategy, (2) the prevalence of JS-libs among all JS files and (3) the way that the adversary modifies a JS-lib. Below we first explain how we collected the data for the study and then get to the details of our findings.

Data collection. As discussed above, the data used in our study was crawled from the web. For this purpose, we implemented a crawler as a Firefox extension and deployed it to 20 Virtual Machines (VMs) . The crawler is designed to explore all URLs it finds, starting from a set of "feeds". For each URL it visits, it renders the web page the URL points to and executes all the dynamic content on the page, like JS code. Then, it dumps all the HTML and JS files discovered during the visit to a database shared among all crawlers. This approach works much more effectively than a static crawler, which just collects web content but never runs it, as new redirections and new web content are increasingly generated through execution of dynamic content such as scripts.

In our study, we ran our crawler over two data feeds. Specifically, we generated the *Alexa feed* by collecting the list of Alexa top one million sites from 2012/07/16 to 2012/07/17. Also we got the *bad feed* from Microsoft on a daily basis between 2012/07/15 and 2012/08/30. The bad feed was derived from the pages indexed by Bing search engine and was confirmed by Microsoft. The web pages discovered through crawling those feeds were further processed and classified into two datasets, *Bad set* and *Good set*, as described below:

• *Bad set:* This dataset contains infected redirector pages. They were crawled from the bad feed containing 1,558,690 doorway URLs and further scanned using Microsoft Security Essentials [31], a leading malware detection service, to identify those infected with redirect scripts, for example, those marked with `Trojan:JS/Iframe.AA` [2]. In this way, we gathered 436,869 unique compromised files (associated with 474,600 URLs), composed of 70,119 JS files (113,729 URLs) and 366,750 HTML files (360,871 URL).

• *Good set:* The dataset contains 396,223 files (associated with 491,171 URLs), including 151,188 JS files (319,269 URLs) and 245,035 HTML files (171,902 URLs) considered to be clean. They were crawled between 2012/07/16 and 2012/07/17, using 69,864 doorway URLs randomly selected from the Alexa feed. To remove infected web content, all the pages we crawled were scanned with Security Essentials in May 2013, which was supposed to filter out the vast majority (if not all) of the files infected one year ago.

Over those datasets, we analyzed the redirect-script injection attacks, as follows.

Blind injection. We first studied the adversary's script injection strategy using the Bad set. All 436,869 files in the set were classified by Security Essentials into 316 different classes of redirect payloads. Among them, some contained only a few URLs discovered by our crawler. To avoid drawing any conclusion based upon such a small sample size, we ignored those with less than 10 unique URLs, which left us 213 types. We found that more than half of them, 115 out of 213 (53.99%), infected both HTML and JS files, 24 (11.27%) only appeared within JS files and 74 (34.74%) were HTML only. Since each malware type labeled by Security Essentials is a cluster of similar scripts, the above result indicates that the adversary tends to blindly inject similar redirect scripts to both HTML and JS files. Actually, even on one compromised host, oftentimes multiple files (HTML or JS) were injected with same redirect scripts. Figure 3 shows one such site, `i-globalsolutions.com`, which was compromised and both its HTML page (`?page_id=146`) and JS files (e.g., `cufon-yui.js`) included the same redirect payload. Apparently, the adversary utilizes this strategy to broadly disseminate his redirect code within a short period of time and make it more likely for a visitor to trigger the redirections.

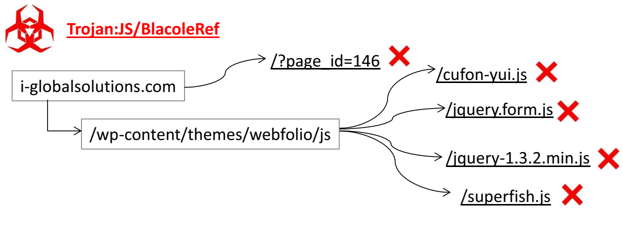


Fig. 3. An example showing a malicious redirect script injected into multiple files on one compromised site.

JS-libs. We further inspected the infected JS files. Some of them appeared to be JS-libs, such as `jQuery`. To understand how pervasive such JS-libs are, we need to identify them from the Bad set. To this end, we first normalized JS file names through removing all their version numbers (consecutive numbers separated by '.') and descriptive terms such as 'min', 'compress' and 'pack'. For example, `jquery-1.3.2.min.js` was converted to `jquery.js`. Then, we clustered all the files with the same normalized names and ranked these clusters according to the numbers of URLs they included. On such a ranked list, top 20 clusters (i.e., normalized names) account for 33.29% of all the URLs within the Bad set. Through manual check (including search for them on the Internet and inspection of their file content), we found that 18 of them are third party JS-libs. Figure 4 illustrates the top 20 clusters and the JS-libs we discovered. These libraries are also extensively used by clean, legitimate websites: from the Good set, 99,140 (31.11%) URLs are associated with the 18 libraries. Moreover, the site owners prefer to use the local copies instead of linking to the remote copies maintained by library developers: among the 99,140 URLs, 70,749 (71.1%) pages use the libraries served by the legitimate sites themselves. While the remote copies save the site owners from maintaining the library code, they could cause performance overhead and incompatibility issues if the code is updated.

JS-file infections. To understand how the adversary alters files and implants redirect code, we took a close look at the infected JS-libs. We focused on those libraries due to the availability of their clean copies, which can be used to compare with the compromised versions to identify their infections (i.e., malicious code). Note that this cannot be done using commercial-off-the-shelf malware detectors like Security Essentials, since they just raise alarms and do not pinpoint malicious code. Specifically, we performed this differential analysis on 100 JS files randomly sampled from the Bad set. Each of these files was within one of the aforementioned 18 name clusters: that is, it was supposed to be a JS-lib. However, we found that 11 of them turned out to be HTML documents[3]. Among the remaining 89 files, 4 contained the redirect code that was used to replace some of the original code within the libraries, 3 had the scripts placed right in front of their library code and 82 carried the malicious payload appended to their legitimate programs. It becomes pretty clear that the adversary tends to carefully arrange his script around the original library code to avoid interfering with a JS-lib's normal operations.

On the other hand, legitimate web developers could also adjust the content of those JS-libs. The question is how such changes differ from what the adversary makes. In our research, we randomly sampled from the Good set 100 files apparently to be the JS-libs, according to their file names (within the 18 clusters). Among these samples, 4 were HTML files (similar to what we found from the Bad set) and 17 utilized very old versions of JS-libs whose original copies were no longer available on their official websites. For the remaining 79 files, 47 were identical to their original copies, 24 contained only very small changes (revised comments, one additional statement, etc.), and only 8 had been modified significantly. We further looked into those 8 samples, and found that 6 of them were just original versions of the libraries compressed by known packers (e.g., [13]) and only 2 files carried some serious semantic changes. Most importantly, all

[2] We did not use VirusTotal [45] to scan the files because there is a limit on the number of files that can be uploaded in a day. In our case, we have around one million files and therefore we could not use this service for the initial labeling. Instead, it is used to verify the new findings (see Section IV-B).

[3] All these files were full of error messages, which we suspect could be used to respond to the request for the JS-libs the host did not serve.

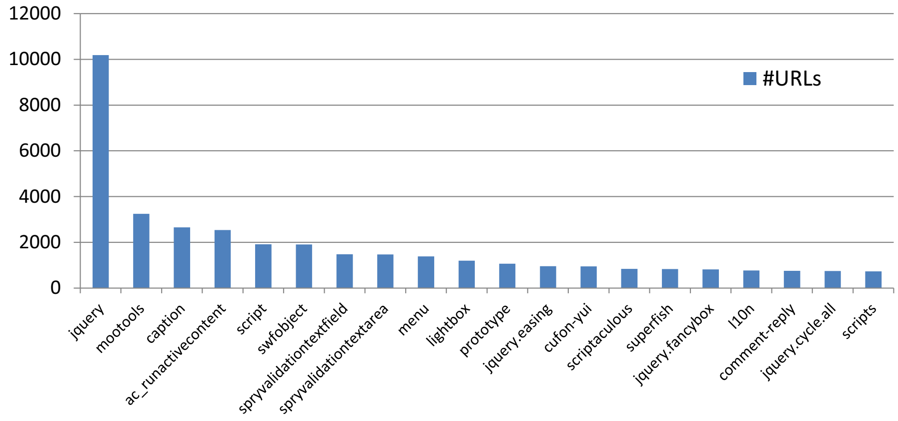


Fig. 4. The top 20 file names discovered in Bad set. Except “script” and “scripts”, all the remaining file names are related to 3rd-party JS libraries.

the content adjustments observed from those clean, legitimate JS-libs are very different from the infection code injected to compromised files: we did not find that any of them would lead to redirections.

Summary. Our study brings to light some intriguing observations related to the redirect-script injection attacks. First, we found that there are a significant portion of these attacks aiming at either HTML and JS files or JS only. For those working on HTML and JS, the adversary tends to inject malicious redirect scripts blindly to both types of files, presumably for the purpose of deploying redirect payloads widely and efficiently. Second, many JS files are JS-libs, whose original versions can be found from their official websites. From the adversary’s viewpoint, deliberately avoiding these libraries not only is time-consuming (for identifying them) but also makes his attack much less effective, given the fact that such libraries make up an important portion of all JS files. Third, legitimate users of those JS-libs tend to keep their original versions and when they have to change those files, they only make minor adjustments most of time and rarely introduce any redirect code. This is completely different from what the adversary does, whose sole purpose is to inject redirect scripts.

III. Automatic Detection of Unknown Redirect-Script Infections

The features we discovered from mass redirect-injection campaigns (Section II-B) offer an opportunity to detect those attacks in a lightweight and effective way. In this section, we present the design, implementation and evaluation of such a new technique, called *JsRED*, for automatic identification of unknown redirect-script infections.

A. Overview

The idea. As described in Section II-B, the adversary tends to blindly inject same or similar redirect scripts into both JS and HTML files in a campaign. Since a significant portion of these JS files are third party JS-libs and their clean versions are publicly available (e.g., on their official websites), our approach exploits those relatively “soft” targets through a *differential analysis*, extracting the script code from the difference between a JS-lib and its clean reference (i.e., its official version), and then *extends* what we learn to other JS and HTML files, using the detected script code to catch their infections. This makes it possible to quickly detect a large number of compromised websites, even a whole campaign, even when the malicious redirect script involved has never been seen before.

More specifically, unknown redirect code can be revealed by checking the output of the differential analysis. Given the observation that a legitimate customization of JS-libs rarely brings in just redirect code, JsRED detects infections by simply determining whether the difference part is a redirect script (*redirector*[4]), based upon a combination of static and dynamic analyses. All the redirectors captured in this way are further generalized into signatures for scanning other suspicious JS and HTML files.

Design. In Figure 5, we illustrate the design of JsRED. It has a mechanism that gathers a set of clean JS-libs (which is meant to be as complete as possible) as references. Each of such references is then compared with every JS file crawled from the web, using a scalable Bloom-filter *inclusion analysis* that measures the proportion of the reference present in the JS file (Section III-B). When most part of the reference is found, we further run google-diff-match-patch [14], a code-diff tool, to extract the difference part of the code (*diff* for short) from the JS file. The diff obtained this way is sent to a *verifier* module that analyzes the code both statically and dynamically: once it is found to be a redirector, we believe that a malicious script is detected (Section III-C). All such scripts are then grouped using the Hierarchical Clustering algorithm [21] and a signature is generated for each such cluster. Those signatures are used to scan other crawled web content, HTML as well as JS, to identify other infected files.

JsRED works fully automatically and is designed to detect zero-day redirect infections. With the lightweight technique it is built upon, Bloom-filter based differential analysis in particular, the approach can efficiently analyze a large number of suspicious web content, detecting most infected JS files with

[4] For simplicity of presentation, here we overload the term, which also refers to compromised websites doing redirections.

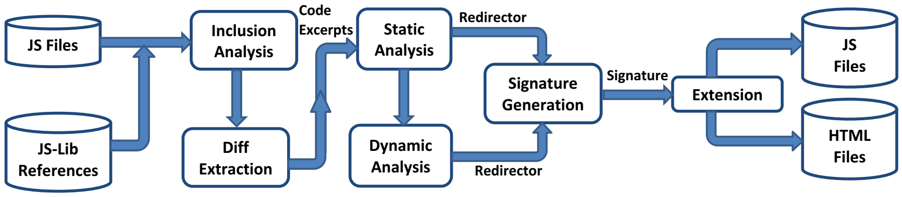


Fig. 5. The framework of JsRED.

almost no false positive (Section IV). In the following, we present the design of individual components.

B. Suspicious Content Extraction

As discussed before, JsRED detects zero-day redirect scripts through a differential analysis on suspicious JS files. To this end, we need to build a reference set, perform an automatic similarity analysis on references and suspicious JS files, and extract the diffs from a subset of them for further analyses, as elaborated below.

Reference collection. Finding clean versions for JS-libs turns out to be more complicated than it appears to be. The challenge here is how to make the list of references as complete as possible. Although one can always enumerate a few most famous libraries such as *jQuery* [38], there are thousands more less popular ones, not to mention even more *plug-ins* for individual JS-libs (over a thousand for jQuery, such as jQuery slider, jQuery selecter, etc.) designed to enrich the functionalities of the original libraries. In our research, we utilized the JS-lib repository maintained by Google (*Google hosted libraries* [17]) to get popular references, which gave us totally 249 libraries.

To acquire clean references for less popular library files and a large number of JS-lib plug-ins, we crawled 602,243 doorway URLs randomly selected from Alexa top one million sites between 2012/07/01 and 2012/07/15 and 382,814 JS files were collected in this process. Note that all these files were scanned by Security Essentials one year later, which ensured that they were all clean. Then, we picked up references from those JS files based on whether individual files were associated with at least two different doorway hosts. In other words, this means that any JS file used by at least two different websites was treated as a JS-lib. The rationale here is that non-lib JS files are rarely utilized by two different sites. Also, even when this aggressive strategy indeed brings in some non-lib files, they could cause the inclusion analysis (see below) to happen when it is unnecessary but will not affect JsRED's effectiveness in detecting malicious scripts. All together, we got 53,339 references (including 249 from Google) in this way.

Inclusion analysis. With the reference set, we are ready to perform a differential analysis to identify malicious scripts included in JS files. A problem is that simply comparing every single JS file crawled from the web (which we call *suspicious* file) with every single reference using a precise code-diff tool turns out to be too heavyweight. A single run of google-diff-match-patch on two files could take seconds or even minutes, while here we are talking about 50,000 reference files and hundreds of thousands of suspicious JS files. To make this analysis scalable, JsRED first utilizes file names to pair a suspicious file with its references. As discussed before, the names for JS-libs are less likely to be changed by the adversary on a compromised website, as this requires modifications of all the code linked to these libs on the site. Directly searching the names across the site files and replacing the occurrences does not work, simply because legitimate websites often dynamically generate the JS-lib name (e.g. using *eval*) when referring to it. Therefore, a heavyweight code analysis is needed here, which does not scale well. We can map suspicious JS files to their corresponding references based on their normalized file names, as described in Section II-B. We have to normalize file names because a lot of web developers change the names of the JS-libs they download, for example, from `jquery-1.3.2.min.js` to `jquery.js`, for convenience of use on their websites. By matching the normalized names, our approach automatically categorizes suspicious files to different subsets of references they need to be compared with.

Even with this classification, we may still need to work on too many references for each suspicious file. As a prominent example, our reference set contains hundreds of different jQuery versions, all of which have to be analyzed against a `jquery.js` crawled from the web. To efficiently go through all these references, our idea is to quickly identify the reference closest to the suspicious file: when these two files are identical, the suspicious JS-lib is exonerated; otherwise, when most part of the reference has been included in the suspicious file, our approach extracts their diff for a further analysis (Section III-C). Note that in the case that the JS file fails to contain a significant portion of any matched reference (according to their normalized names), most likely we do not have its right reference or it has been compressed by an unknown packer. When this happens, JsRED can either skip the file (which may cause a false negative) or send a notice to the website, suggesting an inspection of the file's integrity.

Our design quickly identifies unmodified copies of the references from the set of suspicious JS files by checking their MD5 checksums and removes them from the set. More challenging here is a lightweight inclusion analysis that determines the proportion of reference code within a suspicious file. Intuitively, when the file contains most or all of the reference content but is still different from the reference, the reference's complement part within the suspicious file (the code not in the reference) should be carefully checked to determine whether it is an injected redirect script. In our research, we implemented a simple inclusion-ratio measurement based on n-grams, as elaborated below:

I For both suspicious files and references, our approach first removes comments, new lines and redundant spaces using an open-source tool JSCompressor [50]. Note that during this normalization step, we still preserve non-ASCII characters, which the adversary may use to encode his malicious payload [39], and restrain from lowering letters, which may break JS syntax.

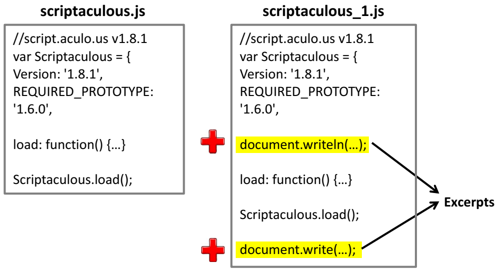
II Then, JsRED breaks the code within individual files into *tokens* using a set of delimiters, including ‘'’, ‘"’, ‘;’, whitespace, ‘\n’, ‘\t’, ‘\r’, ‘(’, ‘)’, ‘{’, ‘}’, ‘[’, ‘]’. Over each token stream, we slide a window of size n to extract n-gram token sequences. In our implementation, $n = 4$.

III After that, our approach compares the n-gram token sequences from a reference and a suspicious file to calculate their *inclusion ratio*: given the set of n-grams for the reference R and that for the suspicious JS file S, we have their ratio $d(S, R) = \frac{|S \cap R|}{|R|}$.

Since the references are used to scan all suspicious files crawled online, we normalized them beforehand in our implementation and built *Bloom filters* to store their n-gram token sequences for high-performance online comparisons. Specifically, for each reference, we ran k random hash functions on its n-grams, mapping them onto a bit array. Given a suspicious file that needs to be compared with the reference, all its n-grams are then tested using the hash functions to determine their memberships within the reference. Based upon this membership test, our approach calculates the inclusion ratio between the file j and the reference i, $d(S_j, R_i)$. If for all the references associated with the file, $\max_i d(S_j, R_i) > \theta$, the diff between the file and every reference whose inclusion ratio goes above this threshold θ is extracted for a further analysis. The threshold θ can be adjusted to strike a balance between the coverage of the detection and its performance (the lower it becomes, the more files we need to check). Also note that though Bloom filter is known to introduce one-sided error (false positive), this will not be an issue for our approach, since JsRED further analyzes the diff to confirm that it is indeed redirect code (Section III-C).

Diff extraction. As discussed above, when a suspicious JS file is not identical to a reference but contains most or all of its content, JsRED needs to inspect the diff from the file (with regards to the reference). To this end, we incorporated google-diff-match-patch [14], an open-source code diff tool, into our implementation to identify all the code within the suspicious JS files not present in the reference. This tool utilizes the classical Myer's diff algorithm [34] to report a list of code segments related to the following editing operations: *INSERT* that injects a block of new code somewhere within the original program (the reference) and *DELETE* that removes part of the original code. JsRED then extracts the code blocks added to the suspicious file and after preprocessing them, runs its verification module to analyze these blocks. Our current implementation does not inspect the relations among different code blocks (called *excerpts*) and instead checks them separately. This treatment is based upon our observation that the code of an injected script tends to stay together without mixing with legitimate code, since otherwise the adversary needs to make efforts to understand each JS he compromises to avoid messing up its original program logic. Of course, we can always enhance our current technique, using more heavyweight information-flow analysis to link different excerpts together when there is a need for doing that. Figure 6 illustrates an example for this diff extraction operation.

Our approach further processes those code excerpts before handing them over for a more heavyweight analysis (Section III-C). Specifically, it first drops those most likely caused by legitimate customizations (by the website's developer). As discussed in Section II-B, such customizations typically lead to only minor changes to the reference (the original version of a JS-lib). On the other hand, a malicious redirect script needs to include enough code for doing its job stealthily. Leveraging this observation, JsRED simply removes all short code excerpts whose size are below α bytes. For the remaining long excerpts, we further search them in the reference dataset and take out those found in any clean, legitimate JS-lib. This operation avoids further inspections of the customizations that merge partial code from two JS-libs together. Finally, our approach discards repeated code excerpts and for every block that consists of repeated strings, we just keep one of them. This cleans individual excerpts of repeated infections (in which the adversary injects the same code multiple times to a JS file).



```
//script.aculo.us v1.8.1
var Scriptaculous = {
Version: '1.8.1',
REQUIRED_PROTOTYPE:
'1.6.0',

load: function() {…}

Scriptaculous.load();
```

```
//script.aculo.us v1.8.1
var Scriptaculous = {
Version: '1.8.1',
REQUIRED_PROTOTYPE:
'1.6.0',

document.writeln(…);

load: function() {…}

Scriptaculous.load();

document.write(…);
```

Fig. 6. An example showing how diff is extracted.

C. Verification and Extension

For an excerpt taken from a suspicious JS file, we need to find out whether it is indeed a redirect script. If so, the code is considered to be an infection instance and a signature will be generated from it and other similar instances. Such a signature is then "extended" to other files for detecting other instances within non-lib JS files and HTML files. In this section, we elaborate how these operations are performed by JsRED.

Redirector identification. As demonstrated by our study on redirect scripts (Section II-B), a legitimate customization of JS-libs, which itself does not happen often, rarely introduces redirect scripts. Therefore, whenever a suspicious excerpt is confirmed to be a redirector, it is almost certain to be an infection. Based on this observation, what JsRED does is a combination of static and dynamic analyses on every suspicious excerpt, in an attempt to catch its redirection operations.

To perform a redirection, a script must either directly set some fields within DOM objects to a target URL or invoke JS APIs (`document.write` or `document.writeln`) to change these fields or inject HTML tags. Table I lists all such standard APIs and DOM object fields. Therefore, the first thing JsRED does to an excerpt is checking its code statically to detect the presence of any of these APIs or fields. Specifically,

No.	Type	API or Field	Input Parameter Pattern
1	Invoke API	document.write	<script|iframe|frame src=*> or
2		document.writeln	<meta http-equiv=”refresh” content=”\d+; url=*”>
3	Set field	window.location	*
4		document.location	
5		[script element].src	
6		[iframe element].src	
7		[frame element].src	

TABLE I. Standard APIs and DOM objects’ fields relevant to redirection.

our implementation parses the code into an Abstract Syntax Tree (AST), using Mozilla’s SpiderMonkey JS engine [33], and searches for the occurrences of the APIs and objects in Table I over the AST. If any of them has been found, JsRED further inspects its parameters to detect the patterns that conform with those of a redirect operation (see Table I). This step helps us quickly identify the valid redirector that has not been obfuscated.

A potential concern is that this treatment will miss the attackers’ injected excerpts that contain syntactic errors and therefore cannot be parsed properly. However, such code cannot be executed correctly either and therefore will not cause any damage to the user. Another trouble comes from google-diff-match-patch, the diff tool our implementation was built upon. This simple tool extracts all the diffs between two documents without looking at their syntactic correctness. For example, given a statement `x=3` in a JS-lib and its counterpart `x=4` in the reference, a fragment “3” will be reported as part of the code excerpt identified. This only happens when the JS-lib has been customized by a website’s developer, as all the injected infections we observed just include complete *new* statements and never touch existing statements. As a result, such fragments will not be introduced during an analysis on the library files that do not include any legitimate modifications. In the case that a customized JS-lib gets infected, which is rare (due to the fact that websites tend to keep such library files intact), some fragments produced could still be parsed by our JS engine (such as “3” in the above example), and therefore an AST can still be correctly built and the follow-up dynamic analysis can still proceed. On the other hand, when a fragment indeed brings in a serious syntactic error that cannot be managed by our current implementation, the analysis can be disrupted. In our experiment (see Section IV), we found that among all the 10,901 excerpts discovered in the bad set, 2,080 (19.08%) triggered syntactic errors during the static analysis. However, a close look at the those problematic excerpts, based upon 50 samples randomly selected, reveals that the vast majority of them (90%) were related to the syntactic problems already there, within either malicious injections or legitimate customization code, and only 10% of them were caused by the fragments introduced by the diff tool. This indicates that even the simple diff tool works well in practice. Of course, we can always improve the accuracy of this diff extraction by switching to a more sophisticated diff tool such as SemanticMerge [5], which allows us to include the whole statement into an excerpt whenever part of it differs from its reference counterpart.

Certainly, static analysis alone is insufficient for catching all the redirect scripts, as it can be circumvented by different obfuscation tricks. Examples include programmatically specifying the objects for the `src` attribute (`script`, `iframe` or `frame`) or even constructing the attribute itself (e.g., `document["wr"+"ite"]()`), and utilizing `eval` and `setTimeout` to unfold redirect code on the fly [11]. When this happens, we need to perform a dynamic analysis to determine the presence of redirect behavior. For this purpose, JsRED uses an instrumented Firefox browser to run each suspicious JS excerpt, in an attempt to find out what it does. Specifically, this dynamic analyzer closely monitors the operations on the `src` attribute under the DOM objects `script`, `iframe` and `frame`, and the `location` property of the global DOM objects `document` and `window`, and reports any changes to their content, which is considered to be a redirection (3-7 in Table I). Also, it intercepts all calls to the DOM functions `document.write` and `document.writeln` to inspect their parameters (which can be formed programmatically, during the excerpt’s runtime) against the redirection patterns (1-2 in Table I).

This static and dynamic combination works effectively against redirect scripts (Section IV), which typically do not contain complicated program logic. However, there are situations where a redirect script could pull some cloaking stunts, stopping its execution when certain conditions are not satisfied. Most prominent examples for such conditions are the client’s user agent and the party that refers the client to a compromised website. To address the cloaking, JsRED configures the dynamic analyzer in a way that most likely triggers the script’s redirect activities: particularly, the user-agent of the analyzer is set to IE-6, the most popular target of web attacks, a Referral of “`http://www.google.com/`” is given to the excerpt whenever it queries the content of this field, and the cookie of each execution is always cleaned before running the next script to detect those that only work on new visitors.

This dynamic analysis can be heavyweight. However, we do not need to use it all the time, given the fact that most websites are legitimate and our techniques are designed to drop the vast majority of them through the differential analysis and preprocessing. In the end, the chance that excerpts move to the verification stage and go through the dynamic analysis is relatively low. In Section IV, we show that only 4.62% of the JS files inspected by JsRED were analyzed within the Firefox browser.

Extension. Once a redirect script is found, it can then be used to identify the infections in other files, including non-lib JS and HTML content. An issue here is the adjustments the adversary may make on different instances of an infection. To catch such instances, JsRED automatically generates a signature for a group of redirectors with similar structures. Specifically, our approach first breaks the code of each script into tokens in the way described in Section III, and then clusters them based on the *Jaccard distance* between each pair of the scripts. Given the token set T_1 for one script and the set T_2 for the other, the distance is calculated as $1-\frac{|T_1 \cap T_2|}{|T_1 \cup T_2|}$. An issue here is duplicated tokens: for example, a small script can look similar to a large one in terms of this Jaccard distance, when the latter contains many duplicated tokens also in the former. To handle this complexity, we also consider the total number of tokens in each script during the clustering and require that the difference in the token number between two scripts does not exceed a given threshold (set to 2 in our experiment). Alternatively, we could use *edit distance* for the clustering purpose, which, however, is more heavyweight. Based upon such pair-wise distances, JsRED further runs the *single-linkage hierarchical clustering algorithm* [21] to merge scripts into clusters using a distance threshold β. Over each cluster, we identify the longest common token sequence across all its members: that is, the largest set of JS statement tokens (see Section III-B) in a given order that are included in every script within the cluster. This can be done through dynamic programming. If this sequence is sufficiently long (no less than 4) and does not match any text in the JS-libs from reference dataset, it then serves as a signature for detecting other infection instances. Figure 7 illustrates an example.

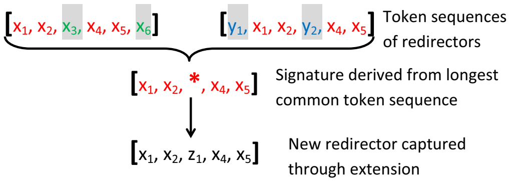

[x1, x2, x3, x4, x5, x6]
[y1, x1, x2, y2, x4, x5]
Token sequences of redirectors
[x1, x2, *, x4, x5]
Signature derived from longest common token sequence
[x1, x2, z1, x4, x5]
New redirector captured through extension

Fig. 7. An example about extension step.

With the signature, we can easily extend what we discover from JS-libs to other files: JsRED simply searches for the token sequence (the signature) within non-lib JS files and HTMLs crawled from the web to find out whether they are also infected. Note that such a scan is a linear-time operation. With its simplicity, this approach works effectively against real-world redirect infections: in Section IV, we show that the extension stage helps to detect around 60% more JS files, without introducing any false positive.

IV. Evaluation

A. Experiment settings

In our experiments, we ran JsRED on a few datasets crawled from the web to detect their infected web content. Such analyses and detection were conducted on a desktop with intel i7-4770 3.40GHz CPU and 32 GB memory. Here we describe the data used in this evaluation and the parameters of our prototype system.

Datasets. Our experiments were conducted on three datasets: the Bad set and the Good set as described in Section II-B and an *Unknown* set collected recently. The Unknown set was used to evaluate JsRED's effectiveness in catching unknown infections. To build this set, we crawled all Alexa top one million sites during 05/01/2013 and 08/30/2013 and then ran blacklist-based filtering on the web content discovered to identify a smaller group of websites more likely to be compromised or malicious than a randomly selected site. This is a standard way to collect a test dataset with a high "toxicity" level [20] for a content analysis under the constraint of limited computing power. In our study, we utilized Google Safebrowsing [16] for this purpose, as did in prior research (*Rozzle* [24]). From the blacklisted websites, we got 33,775 unique JS files (41,311 URLs) and 263,121 unique HTML files (49,879 URLs) for the Unknown set.

Parameter settings. As discussed in Section III, JsRED includes a set of parameters, which were configured as follows in our study:

- *Inclusion ratio threshold (θ)*. This parameter determines when the proportion of a reference file included in a suspicious JS file is significant enough to trigger the differential analysis and its follow-up dynamic verification. Making it too high misses the infected JS files with only small changes to original JS-lib code while making it too low increases the computation burden for analyzing a large number of suspicious files. In our experiments, we set this parameter according to the distribution of the inclusion ratios between confirmed infected JS files and their closest references (see Figure 8). Using the files sampled from the Bad set, we found that a cutoff of 0.95 covers 45.4% of all compromised files.

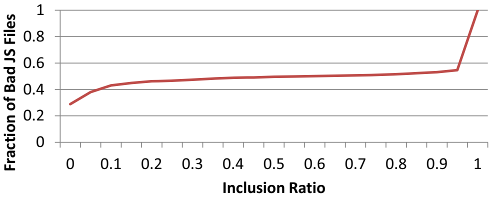


Fig. 8. CDF of inclusion ratio.

- *Minimum size of code excerpt (α)*. As explained in Section III-B, the JS excerpts from suspicious files need to have enough content to be considered as a possible redirect script. We checked the lengths of the redirect code extracted from 100 files randomly sampled from the Bad set (see Section II-B) and found their sizes vary from 83 bytes to 27175 bytes. We chose 50 as the threshold, as all the samples were longer than that.

- *Maximum distance for clustering(β)*. To make the clustering approach (Section III-C) work, a distance threshold needs to be determined. In our study, this parameter was set to 0.12, based upon the distances between the malicious scripts of the same type (classified by Security Essentials) and similar sizes randomly sampled from the Bad set. Figure 9 illustrates the distribution of such distances. We found that a larger threshold significantly increases the chance of including unrelated scripts in a cluster.

B. Effectiveness

Coverage. We first studied JsRED's coverage over the Bad set (see Table II). From the JS files within the dataset, our implementation extracted 10,901 excerpts suspected of causing malicious redirections and performed the dynamic analysis on them. Among these excerpts, 3,514 were confirmed to

Source	Target	#Exceprts	#Redirectors	#JS Files I	#JS URLs I	#Signatures	#JS Files II	#JS URLs II	#HTML Files II	#HTML URLs II
Bad	Bad	10,901	3,514	30,141	52,812	1,394	47,735	70,796	63,912	70,476
Good	Good	8,980	31	37	40	30	42	45	0	0
Bad	Good	-	-	-	-	1,394	6	6	6	6

TABLE II. DETECTION RESULTS ON BAD AND GOOD DATASETS. "SOURCE" DENOTES THE DATASET WE USE TO GENERATE SIGNATURES. "TARGET" DENOTES THE DATASET ON WHICH WE EXTEND THE SIGNATURES. "#JS FILES I" AND "#JS URLS I" MEANS THE NUMBER OF JS FILES/URLS CONTAINING REDIRECTORS IDENTIFIED THROUGH DIFFERENTIAL ANALYSIS. "#JS FILES II", "#JS URLS II", "#HTML FILES II" AND "#HTML URLS II" MEANS THE NUMBER OF ALL DETECTED JS/HTML FILES/URLS.

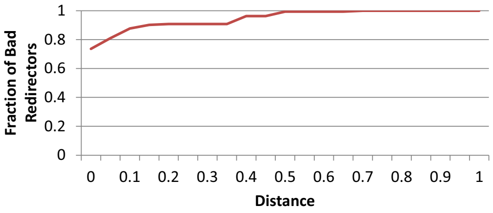


Fig. 9. CDF of distances between different bad redirectors.

be redirectors, contained in 30,141 JS-lib files. Clustering these detected, our approach produced 1,394 signatures. These signatures were found to be very effective in detecting redirect infections: scanning non-lib JS files with them, we captured another 17,594 infections, which increases our total detection counts to 47,735 JS files (70,796 URLs) covering 68.07% of all the JS files within the Bad set (62.24% URLs). This "extension" step also netted 63,912 HTML files (70,476 URLs) infected by redirect scripts, 17.81% of the total HTML files (19.53% URLs) within the Bad set. The results indicate that some attackers indeed inject similar malicious redirectors into both JS and HTML files to efficiently deploy their infection vector in a large scale. This blind injection strategy is exploited by our approach to catch the infections within both types of files. On the other hand, its coverage on the HTML files can be limited, given that some attacks aim specifically at HTML files, using `iframe` and other redirection tags instead of JS code, which JsRED cannot detect. With JS-based injections becoming increasingly popular (due to their stealthiness under search engines), JsRED can serve to complement existing detection techniques that target at HTMLs (e.g., [6]).

False positive. Over the Good set, we assessed JsRED's false positive rate (FPR). False positives can be brought in by the differential analysis on JS-libs and the extension of the signatures generated thereby on other files. In our study, we analyzed the FPRs at both stages. Specifically, from 151,188 JS files (319,269 URLs) within the Good set, JsRED only reported 37 of them as infections through inspecting JS-libs, and generated 30 signatures. Interestingly, even though all these 37 files were supposed to be false positives, given they were found from the Good set, 11 of them turned out to be true positives. They actually contained infections that slipped under the radar of Security Essentials when we scanned the dataset, but were later caught by VirusTotal [45]. Those 30 signatures further led to the discovery of 5 more JS files, 2 of them were confirmed to be true positives. Therefore, JsRED only caused a FPR of 0.019% (29 out of 151,188) among all JS files and a negligible 0.0073% among JS and HTMLs together, as our implementation was never found to mistakenly incriminate any HTMLs. Such a low FPR comes from the fact that legitimate users tend to avoid modifying JS libs and even when they do, they rarely introduce redirections.

We further applied the 1,394 signatures generated from the Bad set to the Good set, in an attempt to understand the FPR of those signatures. In the end, 6 JS files and 6 HTML files, with one URL each, were reported. A close look at them, however, discovered that all of them were actually true positives. Again, those 12 scripts were missed by Security Essentials but were caught by our approach. In other words, JsRED did not cause any false positives in this case.

New detections. Finally, we studied JsRED's potential to detect previously unknown malicious redirectors. To this end, we tested it against the Unknown set, which includes 33,775 JS files (41,311 URLs) and 263,121 HTML files (49,879 URLs) crawled recently. From the JS-lib files among them, our implementation identified 1,562 suspicious excerpts and confirmed the presence of redirect scripts in 266 of them. These findings were further utilized to generate 90 signatures, which matched 143 additional JS files. Altogether, our approach captured 409 infected JS files (with 277 URLs). All of them were confirmed through VirusTotal and manual analysis, without any false positives. By comparison, Microsoft Security Essentials only detected 207 JS files (141 URLs). Most of them (169 JS files associated with 118 URLs) had also been caught by JsRED, while most infections alarmed by our approach (240 JS files associated with 159 URLs) were not found by Security Essentials. In other words, JsRED outperformed Security Essentials by nearly 100% in detecting unknown JS redirect infections[5]. We also scanned the HTML files in Unknown set using the 90 signatures and 264 HTML files (39 URLs) were detected, among which 78 files (9 URLs) were new findings. In the meantime, Security Essentials detected 1677 HTML files (205 URLs). Again, we have to point out here that JsRED is not designed to replace existing anti-virus systems. Instead, it is meant to serve as a complement to them, helping them fare better against the emerging large-scale JS file injection campaigns.

	#JS Files	#JS URLs	#HTML Files	#HTML URLs
JsRED	409	277	264	39
SE	207	141	1677	205
JsRED -SE	240	159	78	9
SE - JsRED	38	23	1491	175

TABLE III. COMPARISON OF THE DETECTION RESULTS: JSRED VS. SE (SECURITY ESSENTIALS).

C. *Performance*

Latency. To understand the performance of JsRED, we measured the time it spent on suspicious content extraction and

[5] Note that in the coverage part, we ran Security Essentials to detect the infections one year before (Section II-B), which were already known.

verification, which involves the differential analysis as well as the static and dynamic analyses, and extension of detection outcomes to find other infections. The results are presented in Figure IV. For the first stage, excerpt extraction and static analysis only took 17.48 ms and 2.12 ms on average, respectively, to process one JS file, but dynamic analysis is much more expensive - 5042 ms on average for working on one excerpt. However, this heavyweight operation does not need to be invoked often. In our research, we studied the frequency with which the dynamic analysis needs to be run using the Unknown datasets. Among all the JS files, only 4.62% of them were analyzed dynamically, while the others all went through the fast channel. Also for the extension stage, scanning each file with all signatures generated from the Bad set (1,394 in total) took 6.15 ms on average.

Stage	Avg Latency (ms)	Std Deviation (ms)
Excerpt Extraction	17.48	89.22
Static Analysis	2.12	1.63
Dynamic Analysis	5042	5860
Extension	6.15	12.31

TABLE IV. LATENCIES AT DIFFERENT STAGES.

Throughput. We further measured the throughput of JsRED using the single desktop described in Section IV-A. JsRED was found to take 19.5 hours to analyze all 255,082 JS files from the Bad, Good and Unknown sets together and generate signatures, and just 2.1 hours to scan all 1,129,988 JS and HTML files using the signatures. With this throughput (roughly 16 file per second), our analyzer can catch up with the web crawler in processing web files. Again, all those results were got from a single desktop. This throughput will go up linearly with the number of machines added to the system.

V. MEASUREMENT AND DISCOVERIES

Based upon what was discovered by JsRED, we further conducted a measurement study that sheds light on the adversary's techniques and strategy in a large-scale redirector injection campaign. Specifically, our study reveals extensive use of blind injection, the evasion tricks the adversary plays on their redirect scripts and the infrastructure of such an injection campaign. Most interesting here is our discovery of a structured P2P redirect network built entirely upon compromised websites in a unique hierarchical way.

A. Analysis of Injected Redirectors

Altogether, JsRED generated 1,541 signatures from the Bad, Good and Unknown sets. Those signatures were used to identify 5,071 unique redirect scripts from 129,997 unique URLs (on 29,881 hosts). We further studied the target URLs the scripts redirected the browser to: all the infected URLs led to 4,923 redirection targets with 3,771 hostnames.

What the data tells us is that not only did the adversary inject similar scripts to many websites, but oftentimes he simply copied the same code to compromised websites, without being bothered to change its content. For example, a redirect script captured by JsRED appeared across 19,819 URLs. Also, clearly a large number of websites were compromised in a mass injection campaign to serve just a handful of malicious hosts, as indicated by the ratio between the infected URLs and the target URLs.

We further analyzed the detected redirect code, in an attempt to better understand the techniques the adversary typically employs to construct redirections and evade detection. Here are what we found:

Redirections. We measured different types of redirect strategies, as illustrated in Table V, built into the injected scripts. The results are presented in Table V. Apparently, injecting HTML tags was much more popular than changing DOM's `location` field. This is possibly due to the former's stealthiness to the web user: a new URL set to the `location` field will show up in the browser, which could attract unwanted attention to the redirection, while an HTML tag silently moves the browser to a malicious website without causing any visual effect. Among such tags, `script` was used more often than others, as the code injected to the hosting page is granted unlimited access to the web content on the page. Also interesting here is the way hidden iframes were used by the adversary. They have been considered to be a major avenue to bring in malicious content and even serve as a detection feature [6][6]. However, among all 821 iframes identified in our research, 340 were not hidden. This suggests that the adversary might have adjusted their strategy to evade such hidden-iframe-based detection.

Redirection Techniques	# Redirectors
Change Location	583
Inject frame tag	148
Inject iframe tag	821
Inject script tag	3837

TABLE V. THE NUMBER OF REDIRECTORS USING DIFFERENT REDIRECTION TECHNIQUES.

Also, we found that a few redirectors made a lot of redirections: among all 5,071 scripts, 863 triggered 2 to staggering 105 redirections each. This turned out to be the result of repeated infections. In these cases, the script detected by JsRED is actually a collection of the redirectors left by multiple injections.

Obfuscation and evasion. Prior research shows that obfuscation has been extensively used by malware to evade detection [18]. In our research, we also took a look at the ways those detected redirect scripts were obfuscated. Specifically, we considered a script to be obfuscated if its redirection URL was not present together with its code in plaintext. Among all 5,071 redirectors, 1,815 did not fit such descriptions. In other words, they exposed their redirection targets in their code. Such an observation is interesting, as apparently, the adversary was not bothered to protect more than a third of the malicious redirectors we detected. For those obfuscated, we found that 22 scripts utilized a standard packer developed by Dean Edwards [13]. Interestingly, one of these scripts was first caught by Security Essentials in July 2012 and later repacked. This obfuscated version remained undetected (by VirusTotal) until October 2013.

In addition to obfuscation, which works against a static analysis, those redirect scripts also employed an array of

[6] They classify an iframe as hidden as if its width and length are no more than 15 pixels.

cloaking tricks to impede a dynamic analysis, as illustrated in Table VI. Most popular here is hiding code within exception handlers [23], possibly because malware detection often happens when exception handling is turned off for performance reasons. We also discovered a new evasion technique, called *parseInt0*, which has never been reported before. The trick here is to exploit an unexpected behavior of the JS API `parseInt` in IE6: when the API's input is a string starting with 0, IE 6 will parse it with the octal radix, while other browsers still use the default decimal radix. As a result, the script can call `parseInt` to parse an input to determine whether it is running inside IE6 before unleashing its redirect payload.

Keywords	# Payload	Description
try/catch	2253	Payload is carried in catch block
onmousemove	446	Payload is not executed until the user moves mouse
navigator	887	Cloaking to specific user agent
cookie	916	Using cookie to prevent revisiting
referrer	1237	Cloaking to certain referrer fields
parseInt0	500	Profiling IE6
setTimeout	715	Delayed execution of malicious code

TABLE VI. THE EVASION TECHNIQUES.

In total, 2,883 scripts (56.9%) have evasion techniques built in. Furthermore, we found that many redirect scripts were armed with multiple evasion techniques. Figure 10 illustrates the number of the techniques discovered in individual scripts: 27.8% utilized more than one trick and 14.2% even employed as many as five techniques. A dynamic analyzer needs to be carefully designed to capture their redirection activities.

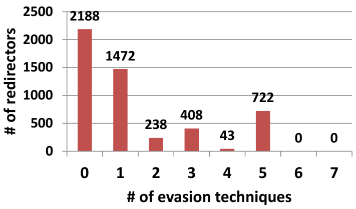


Fig. 10. The number of redirectors Vs. the number of used evasion techniques.

B. P2P Redirect Infrastructures

The most surprising finding of our measurement study is a P2P redirect infrastructure the adversary built through injecting scripts into compromised sites. A malicious web infrastructure typically contains entities of different roles, including compromised sites (i.e. point of entry), dedicated malicious redirectors (e.g., Traffic Direction Systems [27]) and target destinations (e.g. drive by download sites). In our study, we observed a new twist, in which the full malicious infrastructure up to and including the target destinations was built on top of a network of compromised websites. The compromised sites here are utilized not just for their traditional role, that is, serving as a traffic source, but rather a more advanced one in which they perform multiple layers of redirections to other compromised sites until leading the victim to the final destination which could be another compromised site used to deliver malware. Such layered redirections through compromised sites make the infrastructure much harder to detect due to the absence of dedicated malicious sites. Up to our knowledge, only a recent online post reports a similar observation [19] based upon the data apparently collected later than what were used in our research. Most importantly, our research brought to light several key features of this new attack network that have never been reported before, including its dynamic target selection, cloaking strategy and the lifetime of the network. Below we elaborate this study and our findings.

Constructing redirect networks. Our study is based upon a dataset of HTTP traffic collected by our dynamic crawler (Section II-B). This dataset contains the redirection paths discovered by crawling a feed of suspicious URLs provided by Microsoft, which covers the period between April 1st, 2012 and February 28th, 2013 (11 months in total) [7]. Comparing this dataset with the list of compromised URLs and hosts discovered by JsRED (Section IV-B), we extracted redirection paths that include at least one compromised URL (from JsRED) and one other compromised host (the one serving the URLs and files detected by JsRED)[8] to examine the compromised networks they form. This process resulted in a subset of redirection paths that cover 1,760 (5.89%) of the compromised hosts found by JsRED (Section IV-B) as illustrated in Table VII.

# Unique URL to URL paths	47,363
# Host to host paths (host paths)	5,238
Average path length	4
# Compromised hosts	1,760 (5.89%)
# Clusters by payload signatures	63

TABLE VII. OVERVIEW OF COMPROMISED NETWORKS DISCOVERED.

We further clustered those redirection paths using the signatures generated from the malicious scripts caught by JsRED (Section IV-B), which resulted in 63 clusters with the most prevalent cluster covering 68% of all the 1,760 compromised hosts. Those hosts were found to be infected with the scripts part of *RedKit* [3], a drive-by download toolkit. Therefore, we call the whole attack network the "*RedKit network*". We further inspected this network by studying its structure, evolution and lifetimes of the compromised sites.

Network structure. We found that this RedKit network was built through script-based iframe injections. An example of the redirect payload (i.e., the script) is shown in Figure 11, where `ifrm.src` points to another compromised site. We observed this redirect network to mimic the behavior of a P2P network where the compromised hosts had one of three roles: *relay* nodes (i.e., redirectors), *exit* nodes and *target* nodes.

In the network, relay nodes bounced a visitor either to another relay node or to an exit node. Exit nodes, which were also relay nodes, had an additional functionality of leading the victim out of the network by dynamically selecting a target destination and subsequently directing the victim to it, as depicted by Figure 12. The target destinations were also compromised sites but used by the network to deliver malware.

Using discovered redirection paths, we found that relay hosts often replied to a visitor with an HTTP response

[7] The details of the data feed and the technique for generating redirection paths are described in [27].

[8] Here we did not require a path to have two compromised URLs, since this can be too specific. In our research, we manually sampled those paths and confirmed that they were all malicious.

```
function frmAdd() {
        var ifrm = document.createElement('iframe');
        ifrm.style.position='absolute';
        ifrm.style.top='−999em';
        ifrm.style.left='−999em';
        ifrm.src = ”http://www.scuolaartedanza.net/wp−admin/
                template.php”;
        ifrm.id = 'frmId';
        document.body.appendChild(ifrm);
};
window.onload = frmAdd;
```

Fig. 11. Example of a payload used by the RedKit Network.

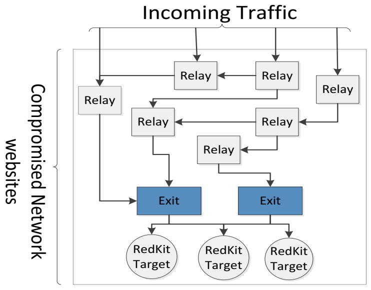


Fig. 12. A diagram depicting the operation of the RedKit Network.

containing the aforementioned iframe injection, while exit hosts when acting as exits always responded with an HTTP redirection (302 or 303). This strategy could serve the purpose of protecting the exit nodes, as the Referral field observed to the visitor only showed the relay node.

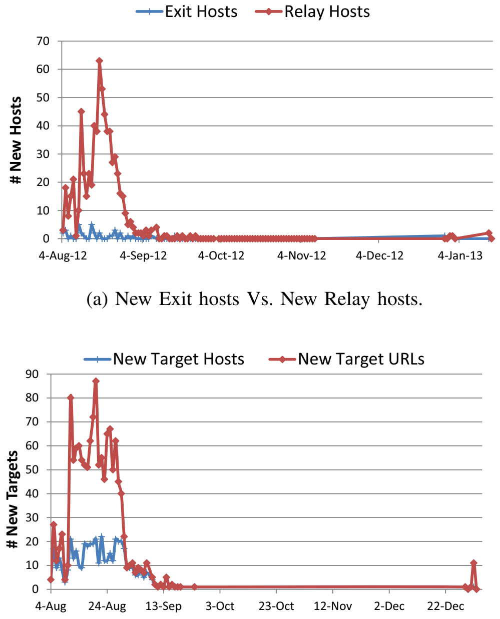
In our study, we were able to identify 37 exit nodes that cover 60% of this network's paths in our dataset. On the other hand, since the compromised hosts here (Section IV-B) were all caught by their injected redirect scripts, it is possible that our dataset failed to include the exit hosts that did not serve as relays (i.e., redirectors), which could contribute to the remaining 40% of the paths.

Network evolution and redirection strategies. We further investigated the evolution of this RedKit network and its redirection strategies. Figure 13a illustrates the number of new exit hosts and relays appearing every day during a five-month period. As we can see here, in the first two months, the network was pretty dynamic, with new members joining it on daily base. Interestingly, the set of exit nodes were rather stable, barely changing during the network's whole life time. By comparison, those relay nodes apparently came and left quickly. This indicates that the exits indeed served as the center of the network with relay nodes all directing traffic to them. Also, apparently the network did a good job in protecting those exit hosts, given the fact that their total number was relatively stable during the period.

We then took a close look at the patterns of the URLs that served as target destinations, that is, to where the exit hosts redirect the visitors. Those URLs were all found to belong to the RedKit exploit kit [3]. In total we discovered 1,340 such target URLs associated with 473 host names with an average of 3 URLs per host. The evolution of these target hosts and URLs is illustrated in Figure 13b. This time, the redirection targets (the URLs and the hosts) change dramatically in the figure, which indicates that the exit hosts frequently modified the URLs and hosts they led the visitor to. Table VIII shows the top 5 exit hosts in the network and their coverage of the target URLs and hosts. All the observations point to a strategy that leverages a large set of expendable relay hosts to funnel traffic to a set of relatively stable, well protected exits, which further dynamically select exploit servers for individual visits from the victims. We also found that the target hosts were shared among the exit nodes while the same target URL was never reused by another node.

Also, part of the adversary's strategy is the way those compromised hosts cloak. Figure 13 shows the network activities slowed down dramatically around the middle of September and then started up again late December in 2012. By analyzing the redirection paths during this period (September to December), we found the exit nodes redirecting our crawler to `google.com` or `google.com/robots.txt`. This clearly indicates that the network blacklisted our crawler and attempted to cloak. Interestingly, our crawler again received attack payloads in late December, which coincides with the network's structural change, as discussed below.

From December 29th, 2012, the RedKit network started moving onto a new structure and strategy. We found that new redirect scripts were introduced and as a result, only the first relay responded with a redirect script to the visitor while the subsequent relay and exit nodes all replied with an HTTP redirection command.



(a) New Exit hosts Vs. New Relay hosts.

(b) New Target URLs Vs. New Target hosts.

Fig. 13. The RedKit Network activity.

Lifetime. To understand how successful such a network was in surviving detection, we investigated its lifetime. Our data set shows that the RedKit network had been there at least from August 4th, 2012 to Jan 16th, 2013 (around 5.5 months). We further estimated the lower bounds for its individual hosts' *compromise lifetimes*, that is, the time period during

#	Exit Host	% Host Paths	% Target URLs	% Target Hosts
1	scuolaartedanza.net	19.9%	21.27%	44.82%
2	mamagre.it	11.5%	13.36%	26.64%
3	olm-torino.it	8.9%	9.33%	20.93%
4	santantonionovoli.it	7%	8.28%	15.22%
5	gynetech.it	7.6%	8.21%	17.97%

TABLE VIII. TOP 5 EXIT HOSTS AND THEIR COVERAGE OF PATHS AND TARGETS.

which those hosts contained infections. For this purpose, we utilized the results of daily scans by Security Essentials and Safebrowsing: a host's lifetime here was estimated by simply counting the number of days in which it was alarmed by either Security Essentials or Safebrowsing as being infected. Note that this estimate is considered to be a lower bound because most hosts in question had not been visited by our crawler on a daily basis. From the dataset, we found an average compromise period of 15 days with a standard deviation of 20 days, a median of 8 days and the maximum of 162 days. Similarly, a study on search-redirection attack [26] shows that the infection lifetime of compromised sites can be as long as 192 days.

Apparently, the adversary built the redirect network to protect compromised hosts from being detected. To find out the effectiveness of this strategy, we compare those on the RedKit network with other compromised hosts, in terms of the lengths of their infections, as observed in our research. The results are presented in Figure 14, which shows that those on the RedKit network indeed stayed longer than those not.

Finally, we re-scanned all the compromised hosts we detected on October 29th, 2013 and found that 9.4% of those on the RedKit network still had not been cleaned up. In contrast, only 1.9% of the compromised hosts not participating in the P2P network were still infected. This indicates that such a network indeed protects compromised hosts against detection. By combining the re-scanning result, we found that the infection period for some hosts (i.e., compromise lifetimes) can be as long as 285 days (8.5 months).

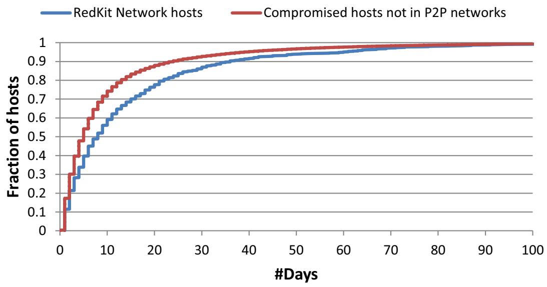


Fig. 14. CDF comparison of the compromise lifetimes of the RedKit Network hosts Vs. compromised hosts not participating in P2P networks.

Hosting providers. We inspected the IP addresses and the corresponding Autonomous System Numbers (ASN) of the compromised hosts. Table IX illustrates the top 5 ASNs observed, which include cloud and hosting providers. Our research shows that many compromised sites were hosted on the same autonomous systems. Upon inspecting the compromised sites, we found that most of those websites were built with a number of website generation platforms known to be vulnerable, such as Wordpress [4], Joomla [2] and Plesk [35].

To discover more compromised sites associated with those ASNs, we performed a reverse lookup on a passive DNS database [1] using those ASN's IP prefixes. This search gave us over 1.5 million hostnames. Scanning these hostnames with Safebrowsing further discovered 2,696 compromised websites, many of which were found to be still accommodated by the same hosting provider on October 29th 2013 as shown in Table IX.

VI. DISCUSSION

Redirect script detection. Our research shows that JsRED works effectively on detecting redirect script injection campaigns. This new technique is designed to complement, rather than replace, existing techniques for identifying compromised HTML files [6]. Although the adversary can always choose to avoid JS files in his campaign, this strategy can make the attack less stealthy, given the fact that HTML and other files are often indexed by search engines and thus are easier to find, while JS files are not. Also, attackers can force compromised sites to redirect visitors through HTTP redirection (302 or 303), without leaving any compromised file (JS or HTML) to crawlers. Again, this strategy is less stealthy as the location of browser is changed. In fact, we found this type of redirection is not prevalent after sampling the Bad set.

Our approach is based on the observation that the adversary tends to blindly inject redirect scripts to JS files, with most of them being JS-libs, and wants to avoid the complexity in substantially modifying those files. To evade our detector, the adversary may choose not to infect JS-libs. However, this renders the adversary more difficult to get traffic, as many JS files are actually JS-libs (Section II-B). Also, substantial changes to the infected JS files require more in-depth understanding of their content, which increases the adversary's cost and slows him down in propagating his attack payload. If the customized packers are used for this purpose, the obfuscated JS-lib file will look very different from what it is supposed to be, which could raise the attention from the website owner JsRED contacts. On the other hand, further research is needed to better understand JsRED's capability to handle those evasion attacks.

P2P redirect network. Our study reveals a P2P redirect infrastructure built entirely upon compromised sites, and made a first step toward understanding its unique features. However, what has been done just scratches the surface of this new attack strategy. Many important issues remain unclear (e.g., how the adversary controls/manages the network), and need further research effort.

Ethical issues. The large number of compromised sites detected (in our case, around 30k) and the challenge in finding right parties to talk to make it difficult for us to inform the owners of the websites found to be compromised in our research, as also happened in related prior research [22], [6]. Actually, most of the compromised sites (99.76%) are part of a ground truth set provided by Microsoft and as such we believe that the affected parties have already been notified when necessary. The remaining sites (0.24%) were all found to be on the blacklist provided by Google Safebrowsing at the

#	ASN#	ASN Description	Country	# Compromised Hosts	# Compromised Hosts Identified by Reverse Lookup
1	31034	Aruba - Shared Hosting and Mail services	IT	70	35
2	26496	GoDaddy	US	47	332
3	21844	ThePlanet.com Internet Services	US	47	2
4	36351	WEBSITEWELCOME.com	US	36	64
5	16276	OVH Dedicated servers	US	34	17

TABLE IX. Top 5 ASNs hosting the RedKit compromised hosts.

time when they were identified in our research. The site owners should already be aware of the infections if they queried the blacklist.

VII. Related Work

Detecting compromised websites. A lot of work has been done to understand and detect compromised websites. Prior research investigated the activities of exploiting vulnerable websites using a network of honeypots [7], and web hosting providers' capability to detect those malicious activities [8]. Also studied were how vulnerable websites are exploited for the purposes of phishing [32] and black-hat search engine optimization [22]. Different from those prior studies, which mainly work on the infected websites delivering attack payloads (e.g., malware, phishing content, etc.), our research focused on compromised sites serving as redirectors, which are known to be hard to detect. Parallel to our work, a recent study [6] reports a new technique that monitors the changes made in compromised sites and further identifies malicious content injected in HTML files using external oracles (i.e., detection services provided by others). By comparison, our research aims at understanding redirector injections that happen to JS files, a much stealthier attack, and our differential analysis on JS-lib files is shown to be capable of detecting zero-day malicious scripts automatically.

Redirection chain analysis. Redirection chain analysis has been performed in multiple prior research to understand or detect malicious web infrastructures [27], [28], [30], [25], [41]. However, those approaches are not designed to capture compromised redirector websites, which are very difficult to differentiate from legitimate websites. Also, once those websites cloak, the whole redirect chain is interrupted and such analyses can no longer move forward. Our approach, however, is built to detect compromised redirect websites and can identify them even in the presence of cloaking.

Code analysis. A typical way to detect compromised websites is through program analysis, such as static analysis [10], [9], dynamic analysis [48], [40] or their combination [11], [24], [36]. These prior approaches are found to work well on drive-by downloads, phishing, etc., but less effective on compromised redirect websites. As described before (Section II), without inspecting redirection targets, it is very difficult to determine whether a piece of redirection code is malicious. We addressed this problem by exploiting unique features of redirect injection attacks and the adversary's constraints, using a simple differential analysis to extract redirect scripts.

Detecting changes in web files. Our detector identifies the injected payloads by comparing the crawled web files with their clean references, which is apparently similar to *Web Tripwires* [37], a technique for detecting changes that happen to web files. However, Web Tripwires needs to be deployed by a website's owner, who knows the clean versions of the files under protection. While JsRED is meant to be operated by a third party, who has no idea about the clean versions of each website, except the JS-libs that ordinary users rarely change. Our new approach is about how to leverage this type of references to detect redirection code injection, without incriminating authorized changes made to those sites.

VIII. Conclusion

Detecting compromised websites that serve as malicious redirectors is challenging, due to the generality of redirect scripts injected into these sites and the cloaking techniques they may utilize to disrupt a redirection chain before it hits exploit servers. In our research, we studied this problem by looking at the adversary's strategy and constraints. Particularly, he has to inject redirect scripts blindly into a large amount of web content, JS files as well as HTMLs, for a rapid deployment and avoid substantial modifications on the compromised files to evade detection. Also, many infected JS files are actually JS libs, whose clean copies are publicly available. Based upon those observations, we developed JsRED, a new detection technique that automatically identifies unknown redirect scripts. JsRED compares a JS-lib file with its clean reference to extract their diff, which is almost certain to be malicious if it is a redirector. From those scripts, we further generate a signature and run it against other JS files and HTLM files to detect infected files. This simple technique turns out to be highly effective, detecting most compromised websites with almost no false positive. It also outperformed commercial malware scan service in terms of detected JS infections. Using the compromised sites JsRED reported, we further performed a measurement study on the properties of such script injection attacks, and discovered a new P2P redirect network built upon compromised websites. Our study brought to light a few important features of such a network that have never been known before.

Acknowledgements

We thank our shepherd Nicolas Christin and anonymous reviewers for their insightful comments and suggestions. We thank Yinglian Xie and Fang Yu from Microsoft Research for providing us bad feed and their valuable feedbacks. This work is supported in part by NSF CNS-1017782, 1117106, 1223477 and 1223495. Alrwais and Alowaisheq are funded by the College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia.

References

[1] Farsight security information exchange. https://api.dnsdb.info/.

[2] Joomla! the cms trusted by millions for their websites. http://www.joomla.org/.

[3] Malware domains list. http://www.malwaredomainlist.com/.

[4] Wordpress, blog tool, publishing platform, and cms. http://wordpress.org/.

[5] SemanticMerge - The diff and merge tool that understands C# and Java. http://www.semanticmerge.com/, 2013.

[6] K. Borgolte, C. Kruegel, and G. Vigna. Delta: Automatic Identification of Unknown Web-based Infection Campaigns. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS '13. ACM, 2013.

[7] D. Canali and D. Balzarotti. Behind the scenes of online attacks: an analysis of exploitation behaviors on the web. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[8] D. Canali, D. Balzarotti, and A. Francillon. The role of web hosting providers in detecting compromised websites. In *Proceedings of the 22nd international conference on World Wide Web*, WWW '13, pages 177–188, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[9] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 197–206, New York, NY, USA, 2011. ACM.

[10] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[11] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*. USENIX Association, 2011.

[12] D. Danchev. Diy commercially-available 'automatic web site hacking as a service' spotted in the wild webroot threat blog. http://www.webroot.com/blog/2013/07/31/diy-commercially-available-automatic-web-site-hacking-as-a-service-spotted-in-the-wild/, July 2013.

[13] D. Edwards. /packer/. http://dean.edwards.name/packer/, 2013.

[14] N. Fraser. google-diff-match-patch - diff, match and patch libraries for plain text - google project hosting. https://code.google.com/p/google-diff-match-patch/, 2013.

[15] Google. Can google site search index javascript content on my pages? https://support.google.com/customsearch/answer/72366?hl=en.

[16] Google. Safe browsing api google developers. https://developers.google.com/safe-browsing/.

[17] Google. Google hosted libraries - developer's guide - make the web faster - google developers. https://developers.google.com/speed/libraries/devguide, 2013.

[18] F. Howard. Malware with your Mocha? Obfuscation and antiemulation tricks in malicious JavaScript. Technical report, Sept. 2010.

[19] F. Howard. Lifting the lid on the redkit exploit kit. http://nakedsecurity.sophos.com/2013/05/03/lifting-the-lid-on-the-redkit-exploit-kit-part-1/, May 2013.

[20] L. Invernizzi, S. Benvenuti, M. Cova, P. M. Comparetti, C. Kruegel, and G. Vigna. Evilseed: A guided approach to finding malicious web pages. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 428–442, Washington, DC, USA, 2012. IEEE Computer Society.

[21] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, Sept. 1999.

[22] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. deseo: combating search-result poisoning. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association.

[23] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: an automated approach to the detection of evasiveweb-based malware. In *Proceedings of the 22nd USENIX conference on Security*, SEC'13, pages 637–652, Berkeley, CA, USA, 2013. USENIX Association.

[24] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 443–457, Washington, DC, USA, 2012. IEEE Computer Society.

[25] S. Lee and J. Kim. WarningBird: Detecting suspicious URLs in Twitter stream. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[26] N. Leontiadis, T. Moore, and N. Christin. Measuring and analyzing search-redirection attacks in the illicit online prescription drug trade. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 19–19, Berkeley, CA, USA, 2011. USENIX Association.

[27] Z. Li, S. Alrwais, Y. Xie, F. Yu, and X. Wang. Finding the linchpins of the dark web: a study on topologically dedicated hosts on malicious web infrastructures. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 112–126, Washington, DC, USA, 2013. IEEE Computer Society.

[28] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 674–686, New York, NY, USA, 2012. ACM.

[29] J. Long, E. Skoudis, and A. v. Eijkelenborg. *Google Hacking for Penetration Testers*. Syngress Publishing, 2004.

[30] L. Lu, R. Perdisci, and W. Lee. Surf: detecting and measuring search poisoning. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 467–476, New York, NY, USA, 2011. ACM.

[31] Microsoft. Microsoft security essentials. http://http://windows.microsoft.com/en-us/windows/security-essentials-download/, 2013.

[32] T. Moore and R. Clayton. Financial cryptography and data security. chapter Evil Searching: Compromise and Recompromise of Internet Hosts for Phishing, pages 256–272. Springer-Verlag, Berlin, Heidelberg, 2009.

[33] Mozilla. Spidermonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/, 2013.

[34] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[35] Parallels. Parallels plesk panel - full-featured control panel experience for hosting management. http://www.parallels.com/products/plesk/, 2013.

[36] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[37] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 31–44, Berkeley, CA, USA, 2008. USENIX Association.

[38] J. Resig. jquery. http://jquery.com, 2013.

[39] K. Security. Incognito exploit kit redux. http://www.kahusecurity.com/2011/incognito-exploit-kit-redux/, 2011.

[40] J. W. Stokes, R. Andersen, C. Seifert, and K. Chellapilla. Webcop: locating neighborhoods of malware on the web. In *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[41] G. Stringhini, C. Kruegel, and G. Vigna. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 133–144, New York, NY, USA, 2013. ACM.

[42] Sucuri. Sucuri 2012 web malware trends report. http://blog.sucuri.net/2013/03/2012-web-malware-trend-report-summary.html, 2012.

[43] Sucuri. Apache php injection to javascript files. http://blog.sucuri.net/2013/06/apache-php-injection-to-javascript-files.html, 2013.

[44] Symantec. 2013 Norton Report. http://www.symantec.com/about/news/resources/press_kits/detail.jsp?pkid=norton-report-2013, 2013.

[45] VirusTotal. Virustotal - free online virus, malware and url scanner. https://www.virustotal.com/, 2013.

[46] W3techs. Usage statistics and market share of javascript libraries for websites, november 2013. http://w3techs.com/technologies/overview/javascript_library/all, November 2013.

[47] D. Y. Wang, S. Savage, and G. M. Voelker. Cloak and dagger: dynamics of web search cloaking. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 477–490. ACM, 2011.

[48] Y. Wang, D. Beck, X. Jiang, and R. Roussev. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'06)*, 2006.

[49] WebSense. Websense 2013 threat report. http://www.websense.com/assets/reports/websense-2013-threat-report.pdf, 2013.

[50] E. Woodruff. A javascript compression tool for web applications. http://www.codeproject.com/Articles/4617/A-JavaScript-Compression-Tool-for-Web-Applications, 2006.